R_GIS 01

Introduction aux géotraitements vectoriels avec R

Septembre 2022

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.

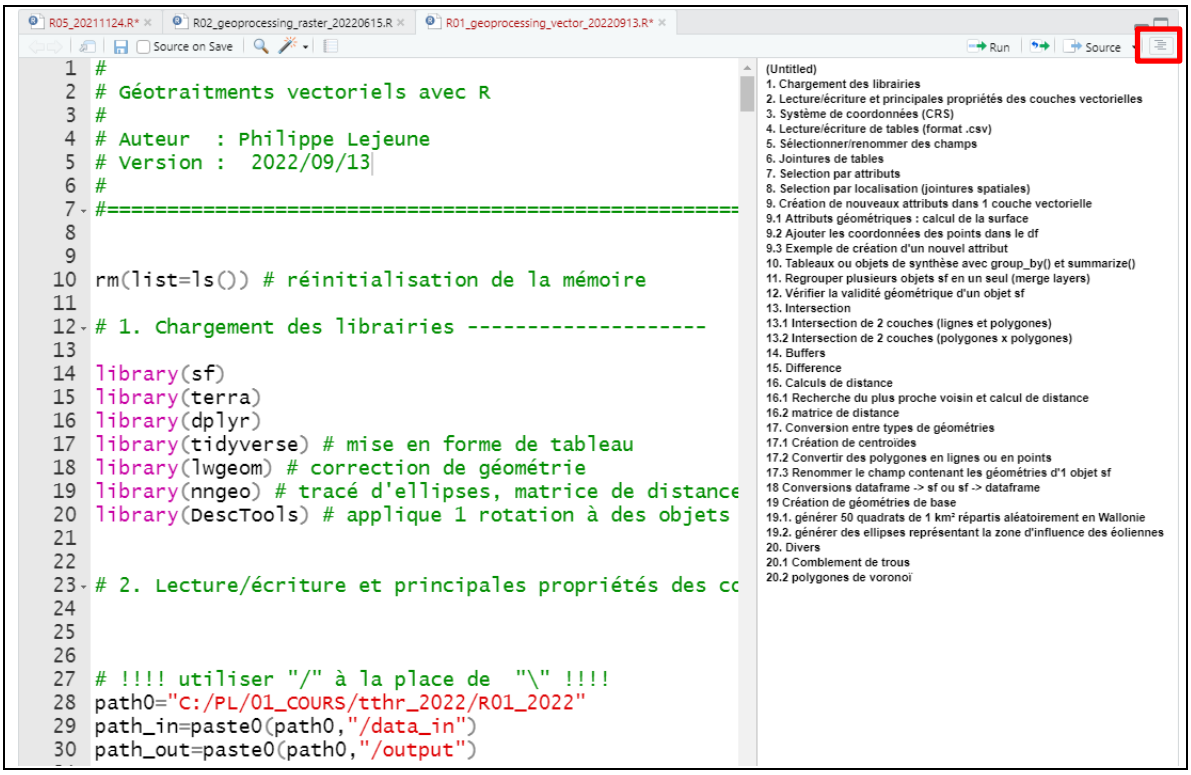
Auteur

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

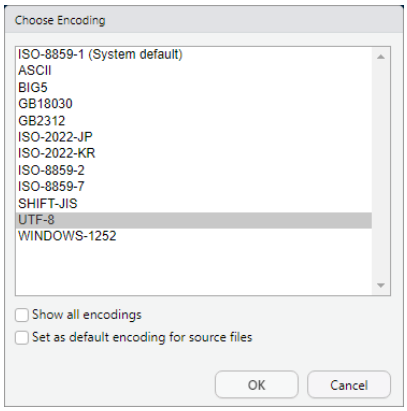
Licence de ce document

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.
- 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 : **sp** et **sf** (gestion des objets vectoriels), **raster** (gestion des objets raster), **rgeos** (géotraitements vectoriels), **rgdal** (géotraitements raster)…
- Désormais, la plus grande partie des traitements vectoriels peut être réalisée avec le package **sf**, pour les géométries vectorielles et le package **terra** pour les données vectorielles ou raster. Le package **dplyr** est pour sa part utilisé pour la gestion des tables attributaires (sélection, jointures, agrégations…).
- Ce tutoriel se focalise sur les principaux géotraitements vectoriels à l'aide des librairies **sf** et **dplyr**. Pour certaines applications spécifiques, il met en œuvre quelques autres packages forts utiles (**nngeo**, **lwgeom**…).
- L'ensemble des opérations présentées dans cet exercice sont rassemblées au sein d'un script **R_GIS_01.r** disponible dans le jeu de données qui accompagne le présent document. Les numéros des paragraphes de ces notes d'exercices peuvent être utilisés comme point de repère pour retrouver les lignes de code dans le script.
- La liste des paragraphes est accessible avec le bouton  accessible dans le bandeau de l'interface de R Studio



- Les manipulations présentées dans ce tutoriel ont été testées dans l'environnement de R Studio, avec la version 4.0.3 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, terra, dplyr, tidyverse, lwgeom, nngeo et DescTools.

```
# 1. Chargement des librairies --------------------

library(sf)
library(terra)
library(dplyr)
library(tidyverse) # mise en forme de tableau
library(lwgeom) # correction de géométrie
library(nngeo) # tracé d'ellipses, matrice de distance
library(DescTools) # applique 1 rotation à des objets
```

3.2 Lecture/écriture et principales propriétés des couches vectorielles

- Classiquement l'accès aux sources de données s'opère après avoir défini le répertoire de travail avec la fonction ***setwd().***
- Nous recommandons de travailler différemment en définissant de manière complète les noms des fichiers, tant à la lecture qu'à l'écriture. Cela implique de définir des variables qui contiennent l'adresse de répertoire contenant les données d'entrée et celle relative au répertoire qui recevra les données de sortie. Cette approche permet notamment d'accéder facilement à des données situées dans différents répertoires.

```
# !!!! utiliser "/" à la place de  "\" !!!!
path0="C:/tmp/R01"  # adapter le chemin en fonction du réprtoire utilisé
path_in=paste0(path0,"/data_in")
path_out=paste0(path0,"/output")
```

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

```
# Lire une couche vectorielle (-> objet sf)
f_comm=paste0(path_in,"/communes.shp")
comm=st_read(f_comm,stringsAsFactors = F)
```

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

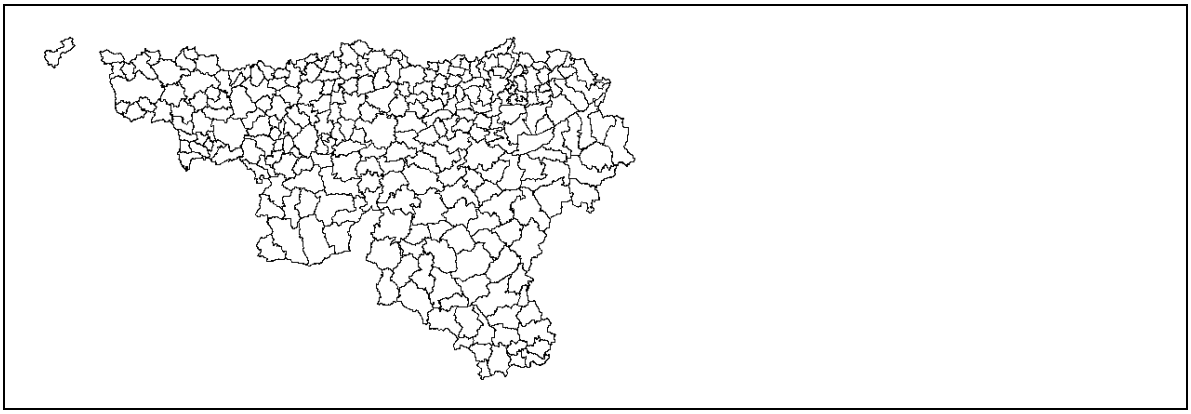
- La fonction ***class()*** renvoie la classe de l'objet comm. Celui-ci est constitué d'une collection d'éléments simples (**S**imple **F**eatures) comprenant des attributs et des géométries sous la forme d'un dataframe. On notera que la géométrie des objets est stockée dans un champ du dataframe baptisé par défaut « geometry ».

```
# Classe de l'objet
class(comm)
# Structure de l'objet comm
str(comm)
```

```
> class(comm)
[1] "sf"         "data.frame"
> str(comm)
Classes 'sf' and 'data.frame':  262 obs. of  11 variables:
 $ OBJECTID  : num  1 2 3 4 5 6 7 8 9 10 ...
 $ ADMUKEY   : chr  "82036" "82037" "82038" "83012" ...
 $ ADPRKEY   : chr  "8" "8" "8" "8" ...
 $ ADMULG    : chr  "F" "F" "F" "F" ...
 $ ADMUNAFR  : chr  "Vaux-sur-Sûre" "Gouvy" "Sainte-Ode" "Durbuy" ...
 $ ADMUNADU  : chr  "Vaux-sur-Sûre" "Gouvy" "Sainte-Ode" "Durbuy" ...
 $ ADMUNAGE  : chr  "Vaux-sur-Sûre" "Gouvy" "Sainte-Ode" "Durbuy" ...
 $ HIGHLIGHT : num  0 0 0 0 0 0 0 0 0 0 ...
 $ SHAPE_Leng: num  66685 77101 57450 95520 64042 ...
 $ SHAPE_Area: num  1.36e+08 1.65e+08 9.79e+07 1.57e+08 7.89e+07 ...
 $ geometry  :sfc_MULTIPOLYGON of length 262; first list element: List of 1
  ..$ :List of 1
  .. ..$ : num [1:4727, 1:2] 239885 239888 239889 239889 239889 ...
  ..- attr(*, "class")= chr [1:3] "XY" "MULTIPOLYGON" "sfg"
```

- L'affichage de la couche s'opère avec la fonction ***plot()***.

```
# Afficher la couche
plot(comm$geometry)
```



- Nous utiliserons très peu les fonctions d'affichage de couches cartographiques dans R, préférant l'environnement QGIS plus interactif et plus simple d'utilisation.

- La fonction ***names()*** affiche la liste des champs contenus dans le dataframe.

```
# Noms des champs contenus dans le df
names(comm)
```

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

- La fonction ***head()*** permet d'afficher les premiers enregistrements contenus dans l'objet comm. Elle donne également certaines informations sur la couche cartographique comme le nombre d'éléments (6), le nombre de champs (10), le type de géométrie (« MULTIPOLYGON »), l'emprise spatiale (Bounding box), et le système de coordonnées.

```
# Premiers enregistrements
head(comm)
```

```
> head(comm)
Simple feature collection with 6 features and 10 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 219270.7 ymin: 62395.48 xmax: 268893.3 ymax: 124947.4
Projected CRS: Belge 1972 / Belgian Lambert 72
  OBJECTID ADMUKEY ADPRKEY ADMULG      ADMUNAFR      ADMUNADU      ADMUNAGE HIGHLIGHT
1        1   82036       8      F Vaux-sur-Sûre Vaux-sur-Sûre Vaux-sur-Sûre         0
2        2   82037       8      F         Gouvy         Gouvy         Gouvy         0
3        3   82038       8      F    Sainte-Ode    Sainte-Ode    Sainte-Ode         0
4        4   83012       8      F        Durbuy        Durbuy        Durbuy         0
5        5   83013       8      F        Erezée        Erezée        Erezée         0
6        6   83028       8      F        Hotton        Hotton        Hotton         0
  SHAPE_Leng SHAPE_Area                       geometry
1   66684.88  135723301 MULTIPOLYGON (((239885 7854...
2   77100.70  165376490 MULTIPOLYGON (((264773.7 10...
3   57449.82   97942306 MULTIPOLYGON (((230440.9 85...
4   95520.37  157096824 MULTIPOLYGON (((224828.5 12...
5   64042.02   78894959 MULTIPOLYGON (((236690.2 11...
6   53087.57   57021128 MULTIPOLYGON (((224040.8 11...
```

- La fonction ***st_bbox()*** renvoie les coordonnées de l'emprise spatiale de la couche cartographique.

```
# Bbox (bounding box = emprise spatiale)
st_bbox(comm)
```

```
> st_bbox(comm)
     xmin      ymin      xmax      ymax
 42244.56  21153.97 295167.07 167684.24
```

- La fonction ***st_write()*** est utilisée pour sauvegarder un objet de la classe sf dans 1 fichier (ou groupe de fichiers dans le cas du format shapefile). L'option « delete_layer=T » permet d'écraser un fichier pré-existant.

```
# Sauvegarder sous forme de shapefile
f_out=paste0(path_out,"/communes.shp")
st_write(comm,f_out,delete_layer=TRUE) # delete_layer=T : overwrite existing file

Writing layer `communes' to data source
  `C:/PL/01_COURS/tthr_2022/R01_2022/output/communes.shp' using driver
ESRI Shapefile'
Writing 262 features with 10 fields and geometry type Multi Polygon.
```

- Le format geopackage est de plus en plus utilisé en remplacement du format shapefile.

```
# Sauvegarder au format geopackage
f_out=paste0(path_out,"/communes.gpkg")
st_write(comm,dsn=f_out,layer="communes",delete_layer=TRUE)

> st_write(comm,dsn=f_out,layer="communes",delete_layer=TRUE)
Deleting layer `communes' using driver `GPKG'
Writing layer `communes' to data source
  `C:/PL/01_COURS/tthr_2022/R01_2022/output/communes.gpkg' using driver `GPKG'
Writing 262 features with 10 fields and geometry type Multi Polygon.
```

- Il présente plusieurs avantages : les données sont stockées dans 1 seul fichier contre 4 pour les shapefile (.shp, .shx, .dbf, .prj). En outre 1 shapefile peut accueillir plusieurs couches cartographiques dans 1 seul fichier. Dans l'exemple qui suit, le polygone correspondant à la commune de Gembloux est extrait de la couche *« comm »* et sauvegardé dans 1 couche séparée nommée « gembloux ».

- La fonction ***st_layers()*** est utilisée pour lister les couches contenues dans 1 geopackage.

```
# 1 fichier geopackage peut contenir plusieurs couches
gembloux=comm[comm$ADMUNAFR=="Gembloux",] # select the polygon for Gem
gembloux
st_write(gembloux,dsn=f_out,layer="gembloux",delete_layer=TRUE) # add

# Afficher la liste des couches contenues dans 1 geopackage
st_layers(f_out)

> st_write(gembloux,dsn=f_out,layer="gembloux",delete_layer=TRUE) # add a second la
er in the gpkg file
Deleting layer `gembloux' using driver `GPKG'
Writing layer `gembloux' to data source
  `C:/PL/01_COURS/tthr_2022/R01_2022/output/communes.gpkg' using driver `GPKG'
Writing 1 features with 10 fields and geometry type Multi Polygon.
>
> # Afficher la liste des couches contenues dans 1 geopackage
> st_layers(f_out)
Driver: GPKG
Available layers:
  layer_name geometry_type features fields
1   communes Multi Polygon      262     10
2   gembloux Multi Polygon        1     10
```

- Pour lire 1 couche dans 1 fichier geopackage qui en contient plusieurs, il faut préciser le nom de la couche souhaitée.

```
# Lecture d'une couche dans 1 gpkg
f_in=paste0(path_out,"/communes.gpkg")
gembloux = st_read(dsn = f_in, layer="gembloux",stringsAsFactors = F)
```

```
> gembloux = st_read(dsn = f_in, layer="gembloux",stringsAsFactors = F)
Reading layer `gembloux' from data source
  `C:\PL\01_COURS\tthr_2022\R01_2022\output\communes.gpkg' using driver `GPKG'
Simple feature collection with 1 feature and 10 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 168172.4 ymin: 131741 xmax: 181041.5 ymax: 144587.7
Projected CRS: Belge 1972 / Belgian Lambert 72
```

- Nous avons évoqué en introduction qu'il était aussi possible d'exploiter les données vectorielles à l'aide de la librairie **terra**. Dans ce cas, la couche est chargée dans un objet de la classe SpatVector.

```
# Lire les données vectorielles en format "spatvecor" (terra)
comm=terra::vect(f_in,layer="gembloux")
class(comm)
```

```
> comm=terra::vect(f_in,layer="gembloux")
> class(comm)
[1] "SpatVector"
attr(,"package")
[1] "terra"
```

- En cas de besoin la conversion entre ces 2 classes d'objet peut être réalisées avec les fonctions ***sf::st_as_sf()*** et ***terra::vect()***.

```
# Conversion spatvector <-> sf
comm=st_as_sf(comm) # spatvector -> sf
comm=vect(comm) # sf -> spatvector
```

3.3 Système de coordonnées

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

```
# Identifier le CRS
st_crs(comm)
```

```
> 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["unnamed",
        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",east,
            ORDER[1],
            LENGTHUNIT["metre",1]],
        AXIS["northing",north,
            ORDER[2],
            LENGTHUNIT["metre",1]],
    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 caractères contenant le nom du système de coordonnées et (2) une seconde chaîne de caractères 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.

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.

```
# Définir le CRS quand il est manquant ou incorrect
f_loc=paste0(path_in,"/localites.shp")
loc=read_sf(f_loc,stringsAsFactors = F)
st_crs(loc)
# attribuer le CRS epsg:31370 à la couche "loc"
st_crs(loc)=31370
st_crs(loc)[1] # on affiche uniquement le code EPSG
```

```
> loc=read_sf(f_loc,stringsAsFactors = F)
> st_crs(loc)
Coordinate Reference System: NA
> # attribuer le CRS epsg:31370 à la couche "loc"
> st_crs(loc)=31370
> st_crs(loc)[1] # on affiche uniquement le code EPSG
$input
[1] "EPSG:31370"
```

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 ».

```
# reprojeter une couche vectorielle
loc_wgs84 = st_transform(loc, 4326)
st_crs(loc_wgs84)[1]
```

```
> loc_wgs84 = st_transform(loc, 4326)
> st_crs(loc_wgs84)[1]
$input
[1] "EPSG:4326"

# comparer les emprises spatiales des 2 couches
st_bbox(loc)
st_bbox(loc_wgs84)

> 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
```

3.4 Lecture et écriture de données tabulaires (dataframe)

- 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.table()***
- Lire le fichier **stat_pop_2018.csv** qui contient les statistiques de population par commune pour l'année 2018 (source : www.statbel.fgov.be).

```
# 4. Lecture/écriture de tables (format .csv) --------------

f_stat=paste0(path_in,"/stat_population_2018.csv")
stat_pop =read.table(f_stat,header=T,sep=";",stringsAsFactors = F)

names(stat_pop)
head(stat_pop, n = 10)
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.

```
f_stat=paste0(path_in,"/stat_agricole_2018.csv")
stat_agri=read.table(f_stat, 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()***.

```
# Sauvegarder un df dans un fichier csv
f_out=paste0(path_out,"/stat_agricole_2018.csv")
write.table(stat_agri,file=f_out,col.names = T,
            row.names = F,sep=";",dec=".")
```

3.5 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.

```
f_comm=paste0(path_in,"/communes.shp")
comm=st_read(f_comm,stringsAsFactors = F)
names(comm)
```

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

- La fonction ***dplyr::select()*** est utilisée pour sélectionner certains champs et, éventuellement, renommer certains de ceux-ci.
- Les exemples présentés ci-dessous montrent différents cas d'utilisation de cette fonction.

- Remarque importante : le champ [geometry] est conservé par défaut, sans devoir le faire apparaître dans la liste des champs à conserver.

```
# exemple 1 : conserver les 5 premiers champs de la couche "comm"
comm1 = dplyr::select(comm,OBJECTID:ADMUNAFR)
names(comm1) # le champs "geometry" est conservé par défaut
```

```
> # exemple 1 : conserver les 5 premiers champs de la couche "comm"
> comm1 = dplyr::select(comm,OBJECTID:ADMUNAFR)
> names(comm1) # le champs "geometry" est conservé par défaut
[1] "OBJECTID" "ADMUKEY"  "ADPRKEY"  "ADMULG"   "ADMUNAFR" "geometry"
```

```
# exemple 2 : sélectionner certains champs
comm1 = dplyr::select(comm,ADMUKEY,ADMUNAFR,SHAPE_Area)
names(comm1)
```

```
> # exemple 2 : sélectionner certains champs
> comm1 = dplyr::select(comm,ADMUKEY,ADMUNAFR,SHAPE_Area)
> names(comm1)
[1] "ADMUKEY"    "ADMUNAFR"   "SHAPE_Area" "geometry"
```

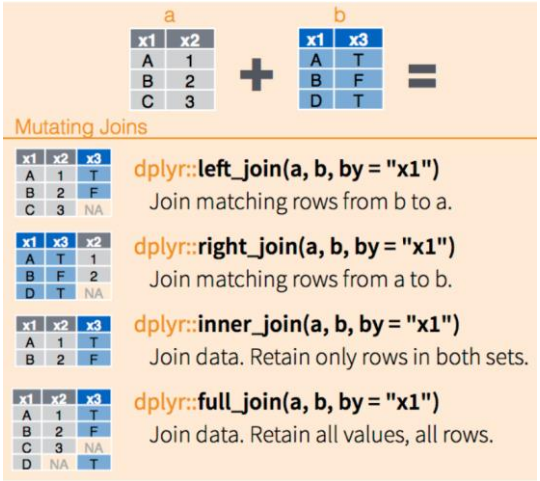
```
# exemple 3 : sélectionner certains champs et renommer ceux-ci
comm1 = dplyr::select(comm,INS = ADMUKEY, numprov= ADPRKEY, nom = ADMUNAFR)
names(comm1)
```

```
> # exemple 3 : sélectionner certains champs et renommer ceux-ci
> comm1 = dplyr::select(comm,INS = ADMUKEY, numprov= ADPRKEY, nom = ADMUNAFR)
> names(comm1)
[1] "INS"      "numprov"  "nom"      "geometry"
```

3.6 Jointures de tables

- Les opérations de jointure constitue un outil de première importance de la gestion de données dans un SIG !
- 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.



- Le cas de figure le plus souvent rencontré est une jointure gauche où des données tabulaires sont ajoutées au dataframe d'un objet de la classe sf.

☞ 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].
- Remarque : 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].

```
# 6. Jointures de tables ----------------------

names(comm1) # créé au § 5
names(stat_pop) # créé au § 4

# Les champs utilisés dans la jointure doivent être du même type
class(comm1$INS)==class(stat_pop$INS)

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

head(comm2)
```

```
> names(comm1) # créé au § 5
[1] "INS"      "numprov"  "nom"      "geometry"
> names(stat_pop) # créé au § 4
[1] "INS"    "Lieu"   "Hommes" "Femmes" "Total"
>
> # Les champs utilisés dans la jointure doivent être du même type
> class(comm1$INS)==class(stat_pop$INS)
[1] TRUE
>
> comm2 = left_join(comm1,stat_pop, by = c("INS" = "INS"))
> names(comm2)
[1] "INS"      "numprov"  "nom"      "Lieu"     "Hommes"   "Femmes"
[7] "Total"    "geometry"
> head(comm2)
Simple feature collection with 6 features and 7 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 219270.7 ymin: 62395.48 xmax: 268893.3 ymax: 124947.4
Projected CRS: Belge 1972 / Belgian Lambert 72
    INS numprov           nom          Lieu Hommes Femmes Total
1 82036       8 Vaux-sur-Sûre Vaux-sur-Sûre   2823   2828  5651
2 82037       8         Gouvy         Gouvy   2608   2598  5206
3 82038       8    Sainte-Ode    Sainte-Ode   1294   1263  2557
4 83012       8        Durbuy        Durbuy   5645   5729 11374
5 83013       8        Erezée        Erezée   1654   1574  3228
6 83028       8        Hotton        Hotton   2728   2803  5531
                       geometry
1 MULTIPOLYGON (((239885 7854...
2 MULTIPOLYGON (((264773.7 10...
3 MULTIPOLYGON (((230440.9 85...
4 MULTIPOLYGON (((224828.5 12...
5 MULTIPOLYGON (((236690.2 11...
6 MULTIPOLYGON (((224040.8 11...
```

```
[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**.

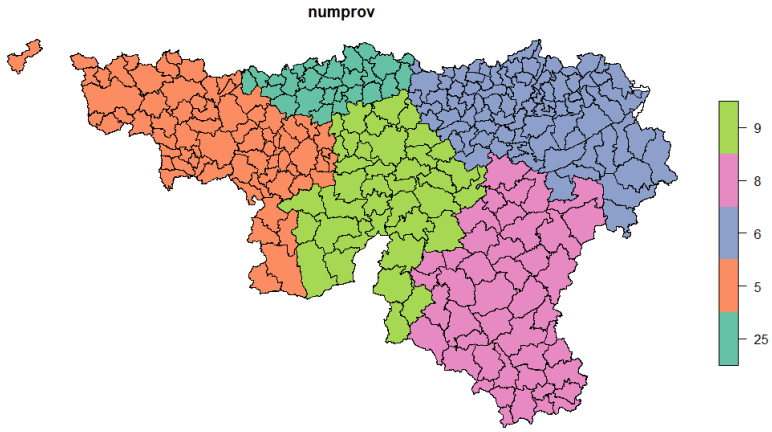
```
> summary(comm2)
    OBJECTID          INS              province           langue              nom
 Min.   :  1.00   Length:262         Length:262         Length:262         Length:262
 1st Qu.: 66.25   Class :character   Class :character   Class :character   Class :character
 Median :131.50   Mode  :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
```

3.7 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

```
# 7. Selection par attributs ----------------

names(comm2)
plot(comm2[,"numprov"])

# province of Namur -> "9"
```



- Créer un nouvel objet baptisé ***« comm_nam »*** reprenant uniquement les communes de la province de Namur.

```
# Sélectionner les communes de la province de Namur
comm_nam = comm2[comm2$numprov == 9,]
plot(comm_nam$geometry)
```

3.8 Sélection par localisation

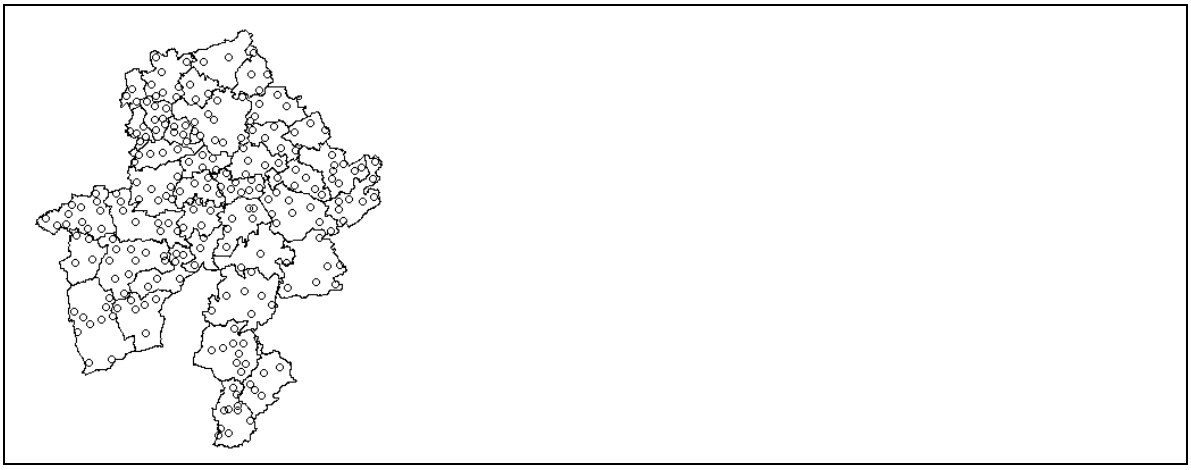
- Les fonctions de construction de requêtes spatiales comparent 2 listes de géométries (par exemple 2 objets sf) de dimension n1 et n2. Elles renvoient une matrice de dimension n1 x n2 qui contient des valeurs VRAI/FAUX traduisant le fait que la relation spatiale testée est VRAI/FAUX pour la paire d'objets 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.

```
# 8. Selection par localisation (jointures spatiales) ------------

# Sélectionner les localités de la province de Namur (inclusion)
prov_nam=summarize(comm_nam)
loc_nam = loc[st_intersects(loc,prov_nam,sparse=F),]
plot(comm_nam$geometry)
plot(loc_nam$geometry,add=T)
```



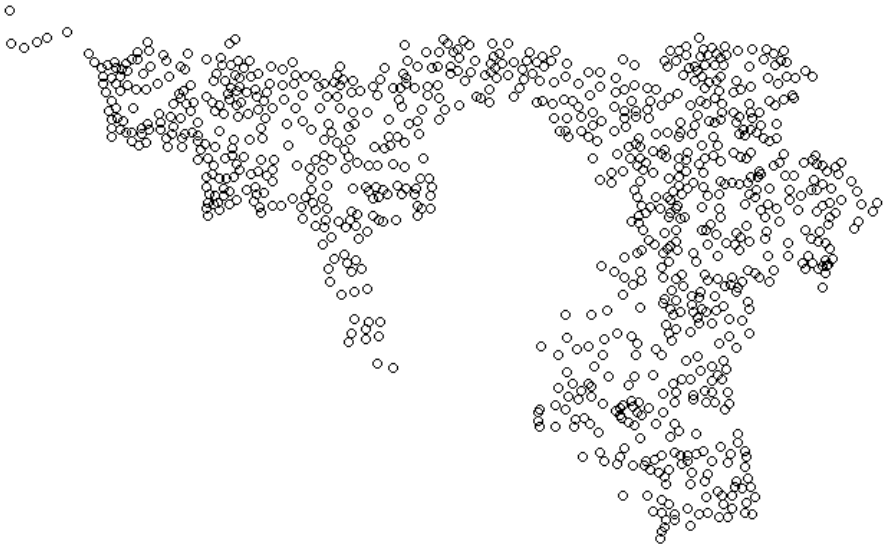
- **Remarque** : l'option « sparse=F » est obligatoire lorsque la fonction « filter » utilise un critère de sélection de type spatial.

☞ 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])
```

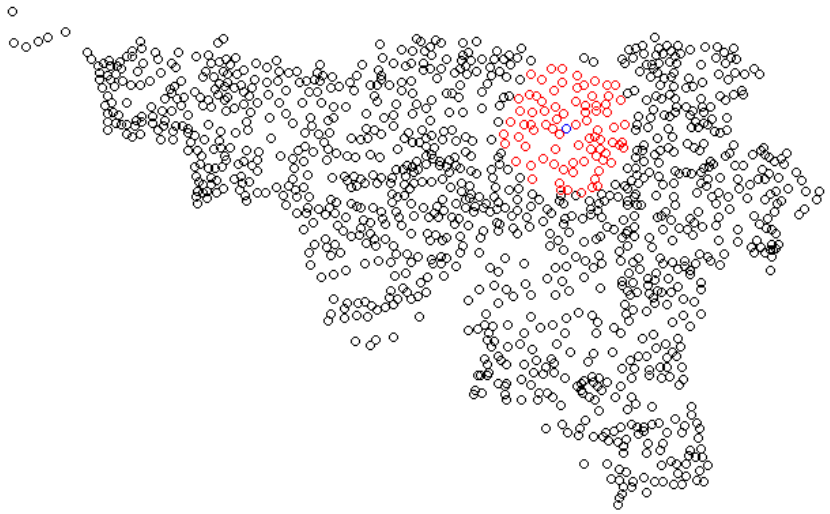
☞ Sélectionner les localités situées **en dehors** de la province de Namur

```
# Sélectionner les localités qui sont hors de la province de Namur
loc_out_nam = loc[st_disjoint(loc,prov_nam,sparse=F),]
plot(loc_out_nam$geometry)
```



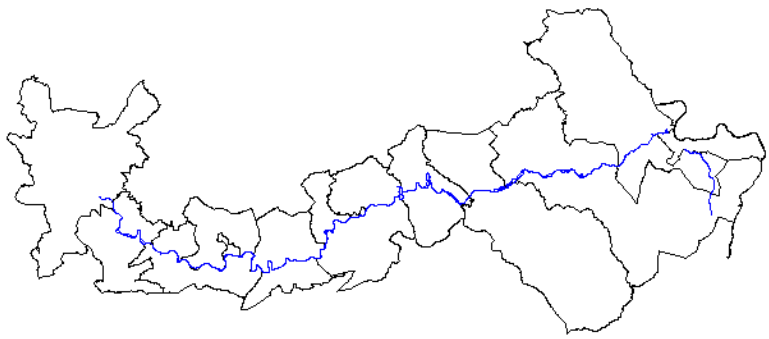
☞ Sélectionner les localités situées à moins de 20 km de la centrale nucléaire de Tihange.

```
# Sélectionner les localités qui sont à moins de 20 km de la Centrale nucléaire
tihange=st_read(paste0(path_in,"/tihange.gpkg"))
loc_prox_tihange = loc[st_is_within_distance(loc,tihange,20000,sparse=F),]
plot(loc$geometry)
plot(loc_prox_thiange,add=T,col="red")
plot(tihange,add=T,col="blue")
```



☞ Sélectionner les communes wallonnes traversées par la Vesdre.

```
# Sélectionner les communes qui sont traversées par la rivière "Vesdre"
vesdre=st_read(paste0(path_in,"/vesdre.gpkg"))
vesdre=summarize(vesdre)

comm_vesdre = comm[st_crosses(comm,vesdre,sparse=F),]
plot(comm_vesdre$geometry)
plot(vesdre$geom,add=T,col="blue")
```



3.9 Création de nouveaux attributs dans un objet sf

3.9.1 Calculer la surface de polygones

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

```
# 9.1 Attributs géométriques : calcul de la surface ---------

# Surface en m²
comm_nam$surf=st_area(comm_nam)
head(comm_nam$surf)

class(comm_nam$surf)
```

```
> # Surface 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"
```

- 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).
- Dans certains cas, on préfèrera travailler en données numériques « pures » et s'affranchir des contraintes liées à la classe « units ». Pour cela, il suffit de transformer les résultats en valeurs numériques avec la fonction ***as.numeric()***.

```
# Convertir le résultat en km² avec 1 format "numeric"
comm_nam$surf_km2 = as.numeric(st_area(comm_nam)/1000000)
head(comm_nam$surf_km2)
class(comm_nam$surf_km2)
```

```
> 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
> class(comm_nam$surf_km2)
[1] "numeric"
```

3.9.2 Ajouter 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()***.

```
# 9.2 Ajouter les coordonnées des points dans le df -
f_loc=paste0(path_in,"/localites.shp")
loc=read_sf(f_loc,stringsAsFactors = F)
loc$x=st_coordinates(loc)[,1]
loc$y=st_coordinates(loc)[,2]
head(loc)
```

```
> head(loc)
Simple feature collection with 6 features and 5 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 210756.8 ymin: 74509.14 xmax: 218055 ymax: 97809.87
Projected CRS: Belge 1972 / Belgian Lambert 72
# A tibble: 6 x 6
     ID NOM          one            geometry       x      y
  <int> <chr>     <dbl>         <POINT [m]>   <dbl>  <dbl>
1     1 Rochefort     1 (211025.3 94425.68) 211025. 94426.
2     2 Hargimont     1 (217108.3 97809.87) 217108. 97810.
3     3 Wavreille     1 (212918.7 90334.98) 212919. 90335.
4     4 Arville       1   (218055 80514.48) 218055. 80514.
5     5 Libin         1 (213752.4 74509.14) 213752. 74509.
6     6 Tellin        1 (210756.8 85898.09) 210757. 85898.
```

- Pour ajouter les coordonnées géographiques des localités, on va procéder à un changement de CRS (voir § 3) et extraire les coordonnées de la couche ainsi produite.

```
# Ajouter les coordonnées géographiques
loc_wgs84 = st_transform(loc,4326)
loc$long=st_coordinates(loc_wgs84)[,1]
loc$lat=st_coordinates(loc_wgs84)[,2]
head(loc)
```

```
> head(loc)
Simple feature collection with 6 features and 7 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 210756.8 ymin: 74509.14 xmax: 218055 ymax: 97809.87
Projected CRS: Belge 1972 / Belgian Lambert 72
# A tibble: 6 x 8
     ID NOM          one            geometry       x      y  long   lat
  <int> <chr>     <dbl>         <POINT [m]>   <dbl>  <dbl> <dbl> <dbl>
1     1 Rochefort     1 (211025.3 94425.68) 211025. 94426.  5.22  50.2
2     2 Hargimont     1 (217108.3 97809.87) 217108. 97810.  5.31  50.2
3     3 Wavreille     1 (212918.7 90334.98) 212919. 90335.  5.25  50.1
4     4 Arville       1   (218055 80514.48) 218055. 80514.  5.32  50.0
5     5 Libin         1 (213752.4 74509.14) 213752. 74509.  5.26  50.0
6     6 Tellin        1 (210756.8 85898.09) 210757. 85898.  5.22  50.1
```

3.9.4 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 dans lequel se situe la commune.

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

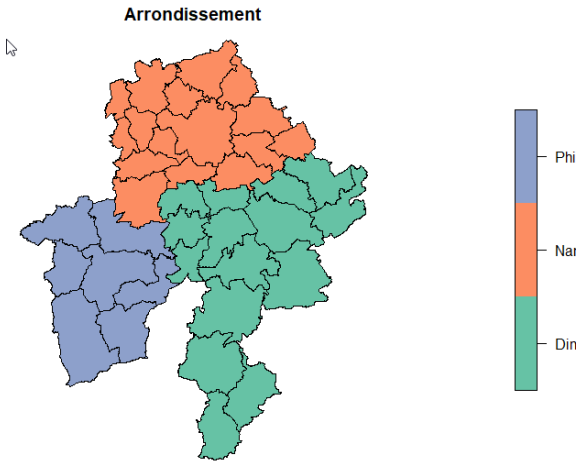
```
# 9.3 Exemple de création d'un nouvel attribut -------------
#     Compléter la couche "comm" avec le nom de l'arrondissement
#     dans lequel se trouve chaque commune
#     Les données sont insérées avec 1 jointure

# Lire le fichier arrondissement_2018.csv
f_arrond=paste0(path_in,"/arrondissements_2018.csv")
arrond=read.csv(f_arrond, 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
comm_nam$id_arr

# step 2 : créer 1 jointure pour insérer le nom de l'arrondissement 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"])
```



3.10 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.

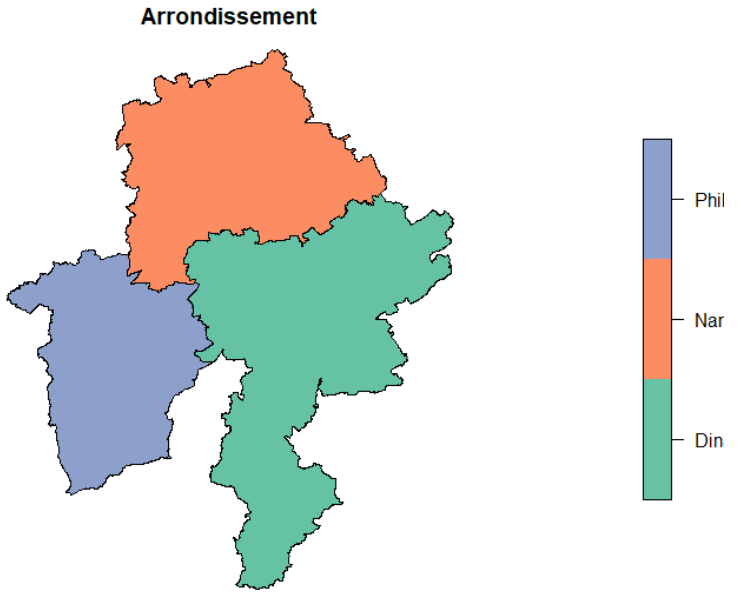
```
# 10. Tableaux ou objets de synthèse avec group_by() et summarize()
f_comm_nam=paste0(path_out,"/comm_nam.gpkg")
comm_nam=st_read(f_comm_nam,stringsAsFactors = F)

names(comm_nam)

# calculer la densité de population 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 par arrondissement
t1=Sys.time()
arr_nam = comm_nam %>%
            group_by(Arrondissement) %>%
            summarize(nb_comm = n(),
                      pop_tot = sum(Total),
                      pop_dens_moy = mean(pop_dens))
t2=Sys.time()
(t2-t1)
class(arr_nam)
names(arr_nam)
plot(arr_nam[1]) # les polygones ont été agrégés par arrondissement
```

```
> # aggréger les données par arrondissement
> t1=Sys.time()
> arr_nam = comm_nam %>%
+             group_by(Arrondissement) %>%
+             summarize(nb_comm = n(),
+                       pop_tot = sum(Total),
+                       pop_dens_moy = mean(pop_dens))
> t2=Sys.time()
> (t2-t1)
Time difference of 0.9015839 secs
> class(arr_nam)
[1] "sf"         "tbl_df"     "tbl"        "data.frame"
> names(arr_nam)
[1] "Arrondissement" "nb_comm"        "pop_tot"        "pop_dens_moy"   "geom"
> plot(arr_nam[1]) # les polygones ont été agrégé par arrondissement
```



- 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.
- La fonction **Sys.Time()** a été utilisée pour mesurer le temps d'exécution de l'opération d'agrégation. La durée de celle-ci est de 0,9 sec. La part la plus importante du temps de calcul est liée à la fusion des géométries (regroupement des communes en arrondissements).

- **Remarque importante** : si le résultat recherché porte uniquement sur l'agrégation des attributs, il est vivement conseillé d'éliminer le champ « geometry » en amont de l'étape d'agrégation. Cela permet de diminuer fortement le temps de calcul, ce dernier pouvant être très élevé lorsque la couche comporte un nombre de géométries plus important.

```
# convertir l'objet en dataframe et supprimer la géométrie des objets
# (abandon des classes "sf", "tbl_df" et "tbl")
df_comm_nam = as.data.frame(st_drop_geometry(comm_nam))
class(df_comm_nam)

t1=Sys.time()
df_arr_nam = df_comm_nam %>%
  group_by(Arrondissement) %>%
  summarize(nb_comm = n(),
            pop_tot = sum(Total),
            pop_dens_moy = mean(pop_dens))
t2=Sys.time()
(t2-t1)
df_arr
```

```
> t1=Sys.time()
> df_arr_nam = df_comm_nam %>%
+   group_by(Arrondissement) %>%
+   summarize(nb_comm = n(),
+             pop_tot = sum(Total),
+             pop_dens_moy = mean(pop_dens))
> t2=Sys.time()
> (t2-t1)
Time difference of 0.08975911 secs
> df_arr_nam
# A tibble: 3 x 4
  Arrondissement nb_comm pop_tot pop_dens_moy
  <chr>            <int>   <int>        <dbl>
1 Dinant              15  110610         75.1
2 Namur               16  316058        259.
3 Philippeville        7   66405         71.5
```

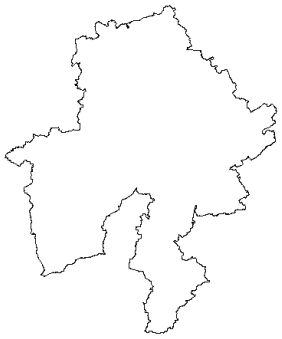
- On constate que le temps de calcul a été divisé par un facteur 10 !

☞ Produire un objet sf contenant 1 polygone 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'objet 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"
```



3.11 Fusion de plusieurs objets sf (fusion de couches)

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

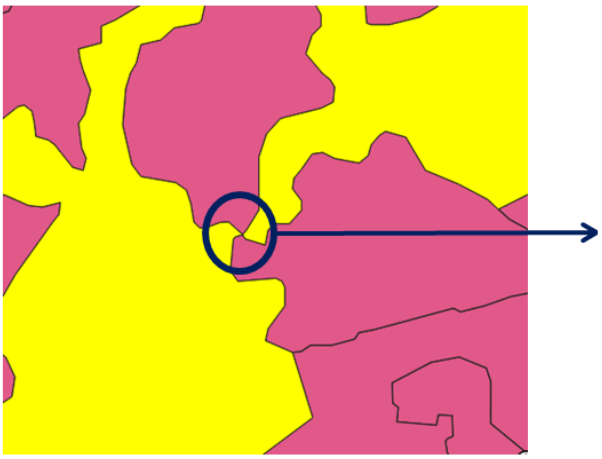
```
# Fusionner les shapefiles 100kmE39N29.shp et 100kmE39N30.shp
clc1=st_read("100kmE39N30.shp")
clc2=st_read("100kmE39N29.shp")
nrow(clc1)
nrow(clc2)

> 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
```

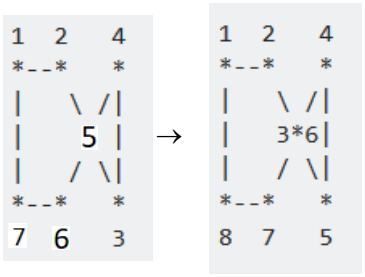
3.12 Vérifier la validité géométrique de couches vectorielles

- 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.




```
# 12. 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
```

3.13 Intersection

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

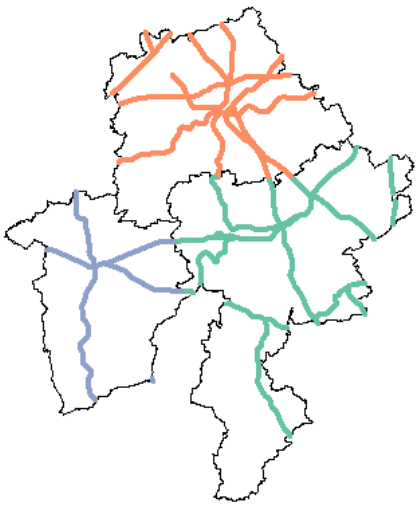
- **Remarque** : cette fonction ne doit pas être confondue avec la fonction ***st_intersects()*** utilisée pour réaliser des requêtes spatiales lors de sélection par localisation

---

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

- La couche ***« comm_nam »*** a été créée au § 3.7.
- Charger le shapefile **routes.shp** dans un objet.
- Réaliser l'intersection des 2 objets sf.

```
# 13.1 Intersection de 2 couches (lignes et polygones) ---

f_routes=paste0(path_in,"/routes.shp")
routes=st_read(f_routes,stringsAsFactors = F)

int=st_intersection(routes,comm_nam)
names(int)
plot(arr_nam$geom) # arr_nam : créé au § 9.3
plot(int[14],lwd=4,add=T)
```



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

```
# Calculer la longueur des segments de route
int$long_km = as.numeric(st_length(int))/1000
int$long_km

f_out=paste0(path0,"/int_routes_comm.gpkg")
st_write(int,f_out,delete_layer=TRUE)

# Calculer la longueur de route par commune

df_int=as.data.frame(st_drop_geometry(int))
tableau1 = df_int %>%
  group_by(nom) %>%
  summarize(long_route = sum(long_km))
tableau1
```

```
> df_int=as.data.frame(st_drop_geometry(int))
> tableau1 = df_int %>%
+   group_by(nom) %>%
+   summarize(long_route = sum(long_km))
> tableau1
# A tibble: 34 x 2
   nom         long_route
   <chr>            <dbl>
 1 Andenne           17.1
 2 Anhée              8.13
 3 Assesse           18.3
 4 Beauraing         22.3
 5 Bièvre             9.32
 6 Cerfontaine        8.48
 7 Ciney             31.1
 8 Couvin            19.8
 9 Dinant            25.8
10 Doische            2.72
# ... with 24 more rows
```

☞ 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.

```
# 13.2 Intersection de 2 couches (polygones x polygones) -----------
# Intersection des communes de la province de Namur et de la couche CLC
# Utiliser le résultat de l'intersection pour 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
int=st_intersection(clc,comm_nam)
int$surf_ha=as.numeric(st_area(int))/10000

# Tableau de synthèse communes x code01
int=as.data.frame(st_drop_geometry(int))
head(int)

# l'abandon de la géométrie permet de réduire fortement le temps de calcul
# pour la création du tableau 2 ci-dessous
tableau2= int %>% group_by(nom,CODE_01) %>%
          summarize(surf_ha=sum(surf_ha))
tableau2
```

```
> tableau2
# A tibble: 124 x 3
# Groups:   nom [38]
   nom      CODE_01 surf_ha
   <chr>      <int>   <dbl>
 1 Andenne        1  2334.
 2 Andenne        2  4657.
 3 Andenne        3  1449.
 4 Andenne        5   173.
 5 Anhée          1   860.
 6 Anhée          2  3446.
 7 Anhée          3  2259.
 8 Anhée          5    32.8
 9 Assesse        1   725.
10 Assesse        2  5043.
# ... with 114 more rows
```

- La fonction ***spread()*** peut être utilisée pour présenter les résultats sous la forme d'un tableau à 2 entrées.

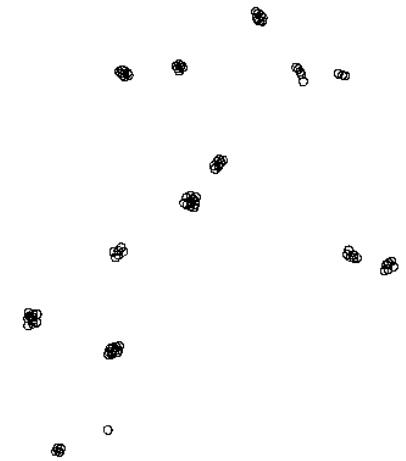
```
# Mise en forme du tableau 2
tableau3=spread(tableau2,CODE_01,surf_ha)
names(tableau3)=c("commune","Z.urbaines","Z.agricoles","Forêt","Eau")
tableau3

> tableau3
# A tibble: 38 x 5
# Groups:   commune [38]
   commune     Z.urbaines Z.agricoles  Forêt   Eau
   <chr>            <dbl>       <dbl>  <dbl> <dbl>
 1 Andenne          2334.       4657.  1449. 173.
 2 Anhée             860.       3446.  2259.  32.8
 3 Assesse           725.       5043.  2085.  NA
 4 Beauraing         812.       8439.  8214.  NA
 5 Bièvre            433.       3879.  6643.  NA
 6 Cerfontaine       424.       4317.  3432. 184.
 7 Ciney            1584.       9407.  3797.  NA
 8 Couvin           1516.       8084. 11061.  38.1
 9 Dinant           1068.       6518.  2321.  98.3
10 Doische           296.       4168.  3948.  NA
# ... with 28 more rows
```

3.14 Buffers

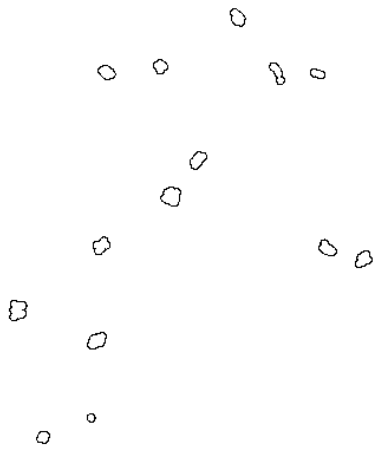
- La création de buffers autour des éléments d'un objet sf s'opère avec la fonction ***st_buffer()***.

☞ Construire des buffers de 600 m autour des mâts éoliens décrits dans la couche **eoliennes.shp**. Regrouper ensuite ces buffers par parc éolien (champ [parc_id]).

```
# Exemple 1 : construire un buffer de 600 m autour
# des points de la couche eoliennes.shp
eol=st_read(paste0(path_in,"/eoliennes.shp"))
eol_buff=st_buffer(eol,dist=600)
plot(eol_buff$geometry)
```

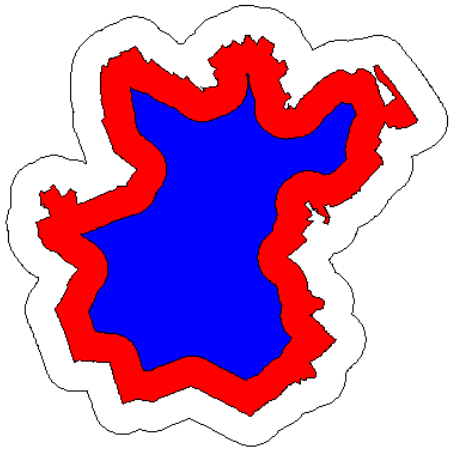


```
# Fusionner les buffers par parcs éoliens (champ [id_parc])
names(eol_buff)
parc_buff = eol_buff %>%
  group_by(id_parc) %>%
  summarize(nb_mat = n())
plot(parc_buff$geometry)
```



☞ Construire un buffer de 1 km autour de la ville de Gembloux.
Construire également un buffer avec une distance négative de -1km

```
# Buffer de 1 km autour de la commune de Gembloux
names(comm_nam) # créé au § 7
gembloux=comm_nam[comm_nam$nom=="Gembloux",]
buff_1km=st_buffer(gembloux,dist=1000)
plot(buff_1km$geom)
plot(gembloux$geom,add=T,col="red")

# buffer avec distance négative
buff_1km_inside=st_buffer(gembloux,dist=-1000)
plot(buff_1km_inside$geom,add=T,col="blue")
```

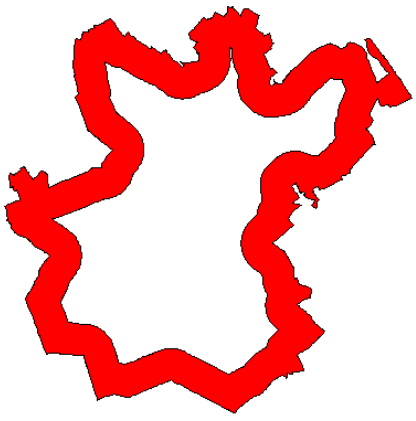


3.15 Difference

- 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 Gembloux située à moins de 1 km de la limite extérieure de la commune.

```
# 15. Difference  --------------------
anneau=st_difference(gembloux,buff_1km_inside)
plot(anneau$geom,col="red")
```



3.16 Calculs de distance

- Les calculs de distance peuvent prendre plusieurs formes. Nous présentons ici 2 exemples : la recherche du plus proche voisin et le calcul d'une matrice de distance .

☞ Les couches **eoleinnes.shp** et **power_lines.shp** contiennent respectivement la position d'éoliennes et le tracé de lignes électriques.
Rechercher pour chaque éolienne, l'identifiant de la ligne électrique la plus proche et la distance à celle-ci.

- La première étape consiste à associer à chaque éolienne, l'identifiant de la ligne la plus proche. Cette étape est réalisée avec une jointure spatiale (§ 8).
- Ensuite, la distance éolienne-ligne électrique est calculée. Pour faire en sorte que cette distance corresponde à la ligne électrique la plus proche, il suffit, au préalable, d'agréger les lignes électriques en 1 seul objet, avec la fonction ***dplyr::summarize()***.

```
# 16.1 Recherche du plus proche voisin et calcul de distance --------
# rechercher la ligne électrique la + proche de chaque éolienne
eol = st_read(paste0(path_in,"/eoliennes.shp"))
pow_l = st_read(paste0(path_in,"/power_lines.shp"))

# recherche de la ligne électrique la plus proche
eol$id_pow_l=st_join(eol, pow_l, join = st_nearest_feature)$full_id
eol$id_pow_l

# calcul de la distance à la ligne la plus proche
# au préalable, il faut agréger les lignes en 1 seul objet
pow_l=summarize(pow_l)
eol$d2pow_l=as.numeric(st_distance(eol,pow_l))
head(eol)
```

```
> head(eol)
Simple feature collection with 6 features and 4 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 158954.3 ymin: 109492.8 xmax: 160383.1 ymax: 110946.5
Projected CRS: Belge 1972 / Belgian Lambert 72
  ID_eol id_parc                  geometry   id_pow_l  d2pow_l
1      1      32 POINT (158954.3 110311.7) w111451151 4070.222
2      2      32 POINT (159649.2 110455.3) w111451151 3995.850
3      3      32 POINT (160045.6 110946.5) w111451151 4169.883
4      4      32 POINT (159236.6 109492.8) w111451151 3204.199
5      5      32 POINT (160383.1 110335.1) w111451151 3488.436
6      6      32   POINT (159628 109849.4) w111451151 3424.929
```

☞ Identifier, pour chaque localité contenue dans la couche localites.shp, les 2 localités les plus proches et calculer la distance à celles-ci.

- Cette question s'apparente au calcul d'une matrice de distance dans laquelle on ne considère que les 3 éléments les plus proches de chaque localités. En effet la localité la plus proche de chaque localité est la localité elle-même.
- La matrice de distance est construite à l'aide de la fonction ***nngeo::st_nn()***. Le paramètre k définit le nombre de plus proches voisins à considérer. L'option « returnDist=T » permet de conserver l'information relative à la distance dans le résultat.

```
# 16.2 matrice de distance ------------
# chercher les 2 localités les plus proches de chaque localité
f_loc=paste0(path_in,"/localites.shp")
loc=read_sf(f_loc,stringsAsFactors = F)
st_crs(loc)=31370
list1=nngeo::st_nn(loc,loc,k=3,returnDist = T)
names(list1)
head(list1$nn)
head(list1$dist)
```

```
> names(list1)
[1] "nn"   "dist"
> head(list1$nn)
[[1]]
[1]  1 18  7

[[2]]
[1]  2 38 49

[[3]]
[1]  3 12  7
```

```
> head(list1$dist)
[[1]]
[1]    0.000 3032.446 4315.549

[[2]]
[1]    0.000 3135.988 4144.936

[[3]]
[1]    0.000 3917.639 4385.976
```

- Les résultats se présentent sous la forme d'une liste constituée de 2 listes : « nn » et « dist »
- La liste « nn » reprend, pour chaque localité, la liste des k voisins les plus proches. La liste « dist » contient les distances correspondant à ces k voisins.
- Ainsi, la première localité contenue dans ***« loc »*** a comme voisins les plus proches, les localités situées en position 18 et 7 dans ***« loc »***. Ces 2 localités sont situées respectivement à 3032 m et 4316 m.
- Pour rendre ces résultats exploitables, il convient de les mettre en forme en fonction des besoins de l'utilisateur. Ici, nous avons choisi de faire apparaître ceux-ci dans le dataframe de la couche « loc », sous la forme de 4 colonnes contenant les noms des 2 localités voisines et les distances auxquelles elles se trouvent. Cette mise en forme s'effectue au sein d'une boucle qui passe en revue chaque localité.
- Le résultat est ensuite sauvegardé dans 1 geopackage pour être visualisé dans QGIS.

```
# Utiliser une boucle pour mettre le résultat en forme

loc$vois1=NA
loc$vois2=NA
loc$dvois1=NA
loc$dvois2=NA
# boucle pour extraire l'info
for (i in 1:nrow(loc)){
  nn=list1$nn[[i]]
  dist=list1$dist[[i]]
  loc$vois1[i]=loc$NOM[nn[2]]
  loc$vois2[i]=loc$NOM[nn[3]]
  loc$dvois1[i]=dist[2]
  loc$dvois2[i]=dist[3]
}
head(loc)
f_out=paste0(path_out,"/loc_voisins.gpkg")
st_write(loc,f_out,delete_layer = T)
```

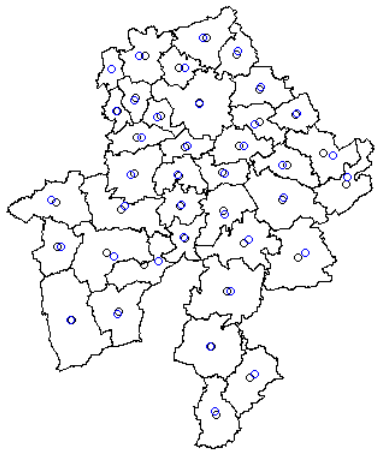
```
> head(loc)
Simple feature collection with 6 features and 7 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 210756.8 ymin: 74509.14 xmax: 218055 ymax: 97809.87
Projected CRS: Belge 1972 / Belgian Lambert 72
# A tibble: 6 x 8
     ID NOM          one            geometry vois1        vois2     dvois1 dvois2
  <int> <chr>      <dbl>         <POINT [m]> <chr>        <chr>      <dbl>  <dbl>
1     1 Rochefort      1 (211025.3 94425.68) Jemelle      Han/Lesse  3032.  4316.
2     2 Hargimont      1 (217108.3 97809.87) Harsin       Humain     3136.  4145.
3     3 Wavreille      1 (212918.7 90334.98) Grupont      Han/Lesse  3918.  4386.
4     4 Arville        1   (218055 80514.48) Saint-Hubert Smuid       3978.  4057.
5     5 Libin          1 (213752.4 74509.14) Anloy        Smuid      4118.  4502.
6     6 Tellin         1 (210756.8 85898.09) Grupont      Wavreille  4625.  4936.
```

3.17 Conversion entre types géométriques

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 garantit en outre que chaque centroïde est situé à l'intérieur de l'élément qu'il représente.

```
# 17. Conversion entre types de géométries -----------------

# 17.1 Création de centroïdes ------------------
centro=st_centroid(comm_nam)
plot(comm_nam$geom)
plot(centro$geom,add=TRUE)

# pour imposer la présence du centroïdes à l'intérieur de l'objet
centro2=st_point_on_surface(comm_nam)
plot(centro2$geom,add=T,col="blue")
```



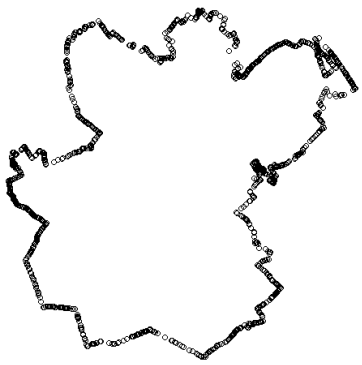
Convertir des polygones en lignes ou en points

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

```
# 17.2 Convertir des polygones en lignes ou en points -----

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

# vérifier dans QGIS
# la frontière séparant 2 communes est "dédoublée"
f_out=paste0(path_out,"/comm_line.gpkg")
st_write(comm3,f_out, delete_layer=TRUE)
```

- Convertir ensuite les lignes en points correspondant aux vertex constitutifs de ces lignes.

```
# Convertir des lignes/polygones -> points
gembloux=comm3[comm3$ADMUNAFR=="Gembloux",]
pnt=st_cast(gembloux,"POINT")
nrow(pnt)
plot(pnt$geometry)
```



- Dans certains cas, on souhaite transformer les polylignes en segments correspondant aux lignes séparant 2 vertex successifs.

```
# éclater des lignes en segments
segm=nngeo::st_segments(gembloux)
nrow(segm)
names(segm)
class(segm$result)
plot(segm$result)
```

```
> nrow(segm)
[1] 1646
> names(segm)
 [1] "INS"           "numprov"       "nom"           "Lieu"
 [5] "Hommes"        "Femmes"        "Total"         "surf"
 [9] "surf_ha"       "surf_km2"      "id_arr"        "Arrondissement"
[13] "pop_dens"      "result"
> class(segm$result)
[1] "sfc_LINESTRING" "sfc"
>
```

- La fonction ***nngeo::st_segments()*** prend en charge cette transformation. La couche produite contient n+1 objet, n étant le nombre de vertex contenu dans la ligne qui constitue le périmètre de la commue de Gembloux.
- A noter que dans le résultat produit par cette fonction, les données de géométrie sont stockées dans un champ baptisé [result].

3.17.1 Convertir des polygones en lignes ou en points

- L'exemple ci-dessous montre comment procéder pour renommer 1 champ contenant les géométries d'un objet sf.

```
# 17.3 Renommer le champ contenant les géométries d'1 objet sf  -·
names(segm)
names(segm)[14]="geometry"
st_geometry(segm)="geometry"
plot(segm$geometry)
```

3.18 Conversions dataframe → sf et sf → dataframe

Convertir un dataframe contenant 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.

- **Remarque importante** : lors de cette opération, les champs contenant les coordonnées sont supprimés du dataframe et un champ [geometry] est ajouté.

```
# 18.1 Conversion df -> sf

# Convertir le fichier localite_wallonie.csv en 1 couche de points
f_in=paste0(path_in,"/localites_wallonie.csv")
loc_wall=read.table(f_in,header=T,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)
```

```
> loc_wall=read.table(f_in,header=T,sep=";",stringsAsFactors = F)
> names(loc_wall)
[1] "ID"  "NOM" "X"   "Y"
> loc_wall=st_as_sf(loc_wall,coords=c("X","Y"),crs=31370)
> plot(loc_wall$geometry)
> names(loc_wall)
[1] "ID"       "NOM"      "geometry"
```

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()***. Celle-ci a déjà été utilisée précédemment, lors de la création de tableaux de synthèse (§ 3.10)
- Si l'on veut conserver les coordonnées « x » et « y » après cette suppression des géométries, il convient de les intégrer dans le dataframe avant l'abandon de la colonne « geometry ».

```
# 18.2 Conversion sf -> df

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

# les coordonnées ont disparu !! -> créer des champs XY au préalable
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)
```

3.19 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 construire ces objets en utilisant 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.

st_point(), st_linestring()
↓
sfg (objet géométrique simple)
↓
st_sfc()
↓
sfc (objet géométrique simple + CRS) + **dataframe**
↓
st_sf()
↓
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()***.

Exemple 1 – générer des quadrats

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

- Avant de créer ces quadrats, il convient de générer 1 couche contenant les limites de la Wallonie.

```
# 19.1. générer 50 quadrats de 1 km²  répartis aléatoirement en Wallonie -

# créer 1 couche "Wallonie"
f_in=paste0(path_in,"/communes.shp")
comm=st_read(f_in,stringsAsFactors = F)
wall=summarize(comm) # créer 1 couche "Wallonie"
plot(wall$geometry)
```

- Ensuite, 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()***.

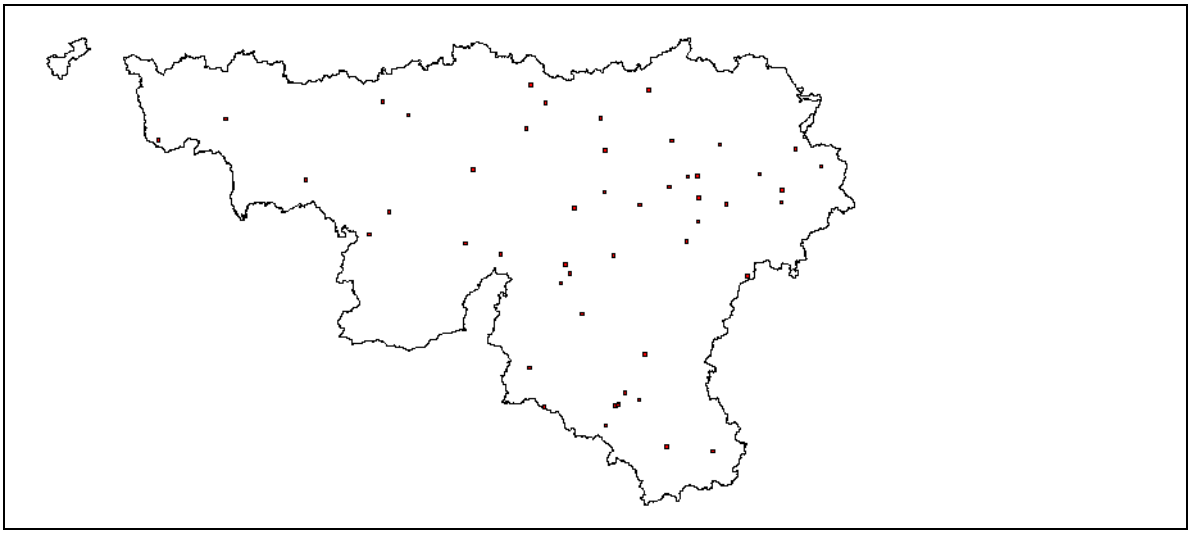
```
# 19.1. générer 50 quadrats de 1 km²  répartis aléatoirement en Wallonie -
comm=st_read("communes.shp",stringsAsFactors = F)

# 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))
f_out=paste0(path_out,"/centers.shp")
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 créé. 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()***.

```
# step 2 : générer les quadrats et les stocker dans 1 objet sf
quadrat=NULL # objet sf dans lequel vont être stockés les quadrats
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()***.

```
# step 3 : convertir la géométrie des objets (lignes -> polygones)
class(quadrat$geometry)
quadrat=st_cast(quadrat,"POLYGON")
plot(wall$geometry)
plot(quadrat$geometry,add=T,col="red")
class(quadrat$geometry,col="red")
```



Exemple 2 – générer des ellipses et des polygones convexes

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

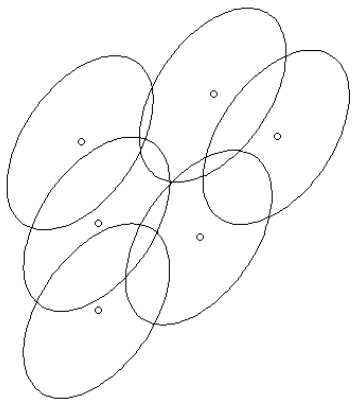
- Le modèle théorique pour définir l'emprise d'une éolienne est celui d'une ellipse dont les demi-axes correspondent respectivement à 7 fois et 4 fois le diamètre du rotor de l'éolienne. Le grand axe de l'ellipse est orienté dans la direction des vents dominants. Ce type de représentation est utilisé 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.

```
# 19.2. générer des ellipses représentant la zone d'influence des éoliennes --
#       du parc d'Ernage (Gembloux) et délimiter l'emprise théorique du parc

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

# Lecture des localisations des éoliennes
eol=st_read(paste0(path_in,"/eoliennes_ernage.shp"))

# paramètres de base
d_rotor = 80 # diamètre du rotor en m
a=7*d_rotor
b=4*d_rotor
azimut=235 # direction des 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.

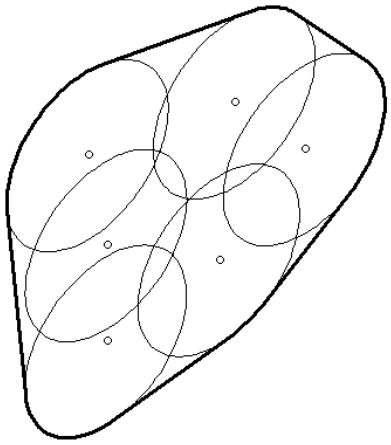
- Transformation de la liste en dataframe puis en objet de type **sf**. Contenant les points situés sur le périmètre de l'ellipse
- Transformation des points en polygone avec les fonctions ***summarize()*** et ***st_cast()***.
- Empilement des polygones dans la variable finale avec la fonction ***rbind()***.

```
emprise_eol=NULL
# 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
plot(emprise_eol$geometry)
plot(eol$geometry,add=T)
```



- La dernière étape consiste à tracer une enveloppe convexe autour des ellipses, après avoir agrégé celles-ci en 1 seul objet.

```
# tracer 1 enveloppe convexe autour des ellipses
limite = summarize(emprise_eol)
limite= st_convex_hull(limite)
plot(limite$geometry,add=T,lwd=3)
```

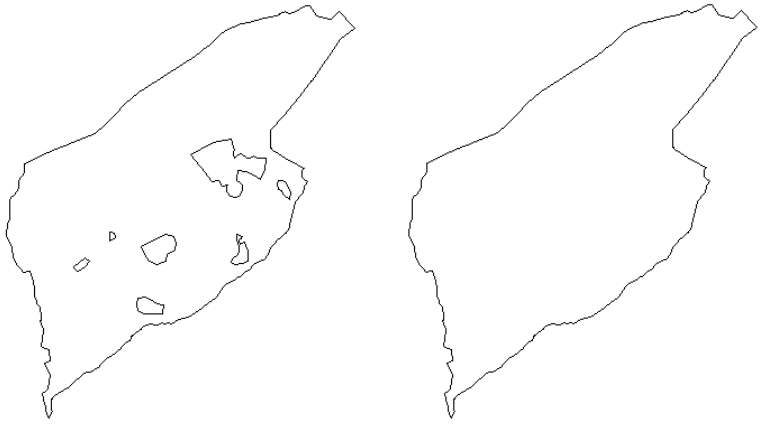


3.20 Divers

Comblement de trous

- Les comblement de trous peut s'avérer nécessaire dans certains cas particuliers. On l'effectue avec la fonction ***nngeo::st_remove_holes()***.

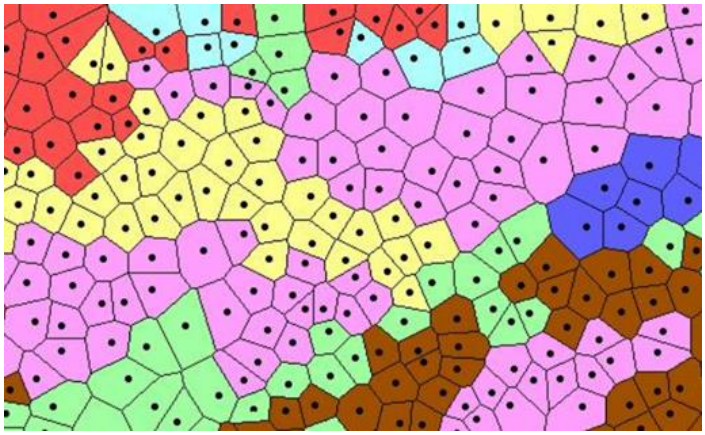
```
# 20.1 Comblement de trous ----------------
f_in=paste0(path_in,"/compartiment.gpkg")
comp=st_read(f_in,stringsAsFactors = F)
plot(comp$geom)

comp=nngeo::st_remove_holes(comp)
plot(comp$geom)
```



Avant *Après*

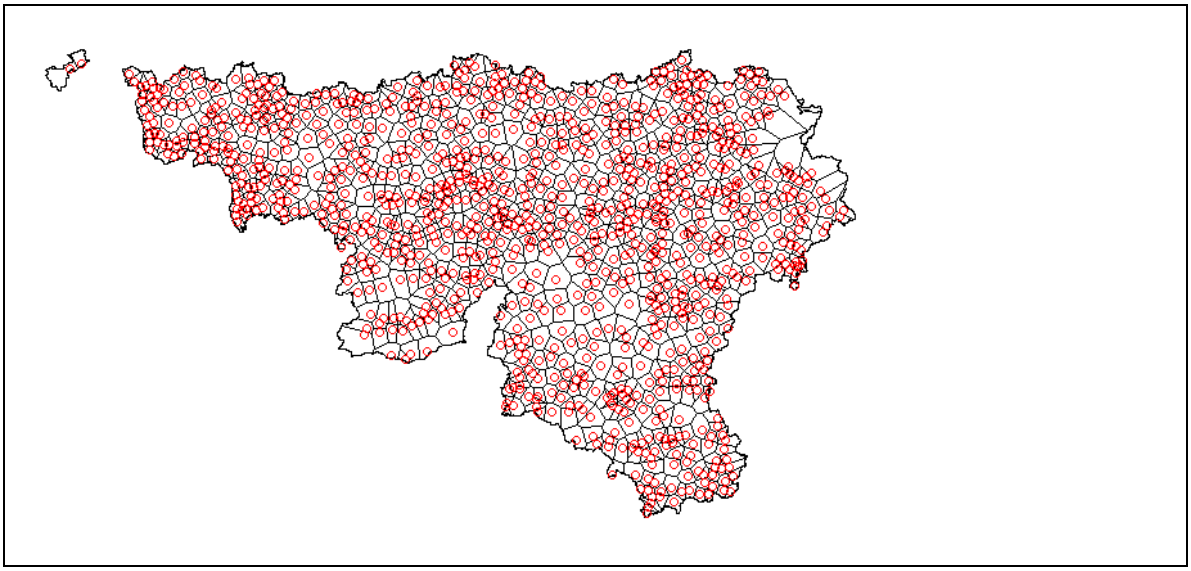
Polygones de Voronoï

- Les polygones de Voronoï constituent un découpage de l'espace déterminé par les distances à un ensemble de points. Cette partition est telle que chaque point de l'espace se retrouve dans 1 polygone associé au point le plus proche.



Le script présenté ci-dessous construit une couche de polygones de Voronoï en s'appuyant sur les points de la couche **localites.shp**. Le résultat final est ensuite découpé aux limites de la Wallonie.

```
# 20.2 polygones de voronoï-------------
# découper la Wallonie sur base de la proximité aux différentes localités
f_loc=paste0(path_in,"/localites.shp")
loc=read_sf(f_loc,stringsAsFactors = F)
st_crs(loc)=31370
# créer 1 couche avec les limites de la Wallonie
f_comm=paste0(path_in,"/communes.shp")
comm=st_read(f_comm,stringsAsFactors = F)
wall=summarize(comm) # utilisé pour découper le résulat
# créer la couche de polygones de Voronoï
loc=st_intersection(loc,wall) # élimine les localités hors Wallonie
all_loc=dplyr::summarize(loc) # regrouper les points avant de générer les pol. de
v = st_voronoi(all_loc) # création des polygones de voronoi
v = st_collection_extract(v,type="POLYGON") # convertir les géométries -> POLYGON
v$id_vor=seq.int(nrow(v)) # attribuer 1 identifiant
# attribuer les attributs des localités aux polygones de voronoï
int=st_intersection(loc,v) # croiser les polygones avec les localités
df=as.data.frame(st_drop_geometry(int))
v=left_join(v,df, by=c("id_vor"="id_vor")) # ajouter les attributs des localités
# découper les polygones aux limites de la Wallonie
v=st_intersection(v,wall)
plot(v$geometry)
plot(loc$geometry,add=T,col="red")
# sauvegarder les résultat
f_out=paste0(path_out,"/vor.gpkg")
st_write(v,f_out,delete_layer = T)
```



4. Résumé des fonctions présentées dans le tutoriel

Fonctions	Descriptif	Référence
st_read() st_write() st_bbox() st_layers() st_as_sf() vect()	Charge une couche vectorielle dans un objet **sf** Sauvegarde d'un objet **sf** dans une couche vectorielle Fournit les coordonnées de l'emprise spatiale d'une couche sf Affiche la liste des couches contenues dans 1 base de données geopackage Convertit un objet spatVector en objet sf Convertit 1 objet sf en objet spatVector	3.2
st_crs() st_crs() = st_transform()	Affiche le CRS d'un objet **sf** Modifie le CRS d'un objet **sf** Reprojette un objet **sf** (changement de CRS)	3.3
read.table() write.table()	Lit un fichier csv (création d'un dataframe) Sauvegarde un dataframe dans 1 fichier .csv ou .txt	3.4
select()	Sélectionner et/ou renommer des champs dans un dataframe (création d'un nouveau dataframe)	3.5
left_join	Jointure de tables	3.6
= sf[]	Sélection par attributs	3.7
 st_intersects() st_disjoint() st_is_within_distance() st_crosses()	Sélection par localisation Relation d'intersection Relation d'absence d'intersection (disjointure) Relation de proximité Relation de croisement	3.8
st_area() st_coordinates()	Calcule la surface des éléments d'un objet **sf** Extrait les coordonnées d'un objet **sf**	3.9
summarize()	Crée des tableaux de synthèse Fusionne des éléments dans un objet **sf**	3.10
rbind()	Fusionne plusieurs couches vectorielles	3.11
st_is_valid() st_make_valid()	Vérifie la validité topologique d'un objet **sf** Corrige des problèmes topologiques dans un objet **sf**	3.12
st_intersection() spread()	Intersection de 2 objets **sf** Transforme 1 dataframe en tableau à 2 entrées	3.13
st_buffer()	Génère des buffers autour des éléments d'un objet **sf**	3.14
st_difference()	Produit une couche de différence géométrique entre 2 objets **sf**.	3.15
st_join() st_nearest feature()	Jointure spatiale Opérateur « + proche voisin » d'une jointure spatiale	3.16

st_distance() st_nn	Calcule la distance entre les éléments de 2 objets **sf** Construit une matrice de distance	
st_centroid() st_point_on_surface() st_cast() st_segments()	Génère des centroïdes pour les éléments d'un objet **sf** Force les centroïdes à se trouver à l'intérieur des géométries Convertit des polygones en polylignes ou des polygones/ polylignes en points Converti des polylignes en segments de lignes	3.17
st_as_sf() st_coordinates() st_drop_geometry()	Convertit un dataframe en objet sf en considérant les colonnes contenant des coordonnées x et y Extrait les coordonnées d'un objet **sf** Supprimant la géométrie d'un objet **sf**	3.18
st_sample() st_linestring() sf_sfc() st_sf st_cast() st_ellipse() Rotate()	Génère des points répartis aléatoirement au sein d'une zone de référence Création d'objets géométriques de base (sfg) de type linéaire Transforme des objets sfg en objets sfc Transforme des objets sfc en objets sf Transforme le type de géométrie pour des objets sf Génère des ellipses Applique une rotation à un objet géométrique	3.19
 st_convex_hull() st_remove_holes() st_voronoi()	Divers Génère des enveloppes convexes Supprime les trous dans des polygones Génère des polygones de Voronoï	3.20