

Prise en main des packages sf, dplyr et ggplot

2020-2021

Contents

1	Prérequis	1
1.1	Le combo sf et dplyr	1
1.2	Les limites de l'exercice	2
2	Les données	2
2.1	Chargement des données	2
2.2	Filtrage des données : Valeurs NA	5
2.3	Selection d'un sous ensemble de données	6
2.3.1	Premier affichage	6
2.3.2	Sélection d'une région	7
2.3.3	Carte choroplète d'un facteur	9
2.4	Système de coordonnées de référence	11
3	Cartographie directe d'un attribut	12
3.1	Carte choroplète d'un facteur avec ggplot	12
3.2	Carte choroplète d'un facteur avec ggplot (bis)	14
3.3	Sous-ensembles des IRIS de Meuse	15
3.4	Carte à symbole proportionnels d'une quantité	16
3.4.1	centroïdes des IRIS	16
4	Manipulations des géométries	17
4.1	Jointure spatiale	17
4.2	Union de géométries	17
5	Manipulations basées sur les attributs	17

Objectif principal : Fournir une base de code qui utilise les packages **sf** (manipulation d'objets géographiques), **ggplot** (graphiques) et **dplyr** (manipulation de données au sens large) pour charger, afficher et requérir des objets spatiaux.

Objectif secondaire : Donner un aperçu des capacités SIG de R

1 Prérequis

- Connaissance minimale de R et des SIG
- installation des pacakges **sf** et **tidyverse** (contient **dplyr**, **ggplot2** et beaucoup d'autres choses)

```
install.packages("sf")
install.packages("tidyverse")
```

1.1 Le combo **sf** et **dplyr**

Très rapidement : le package **sf** manipule les objets géographiques comme des dataframes augmentés d'une colonne géométrie. Cette colonne géométrie "suit" les individus (les lignes) du dataframe lors des

manipulations. C'est très pratique pour les opérations (combinaisons, agrégations, filtrages, ...) opérés sur les **attributs** des objets , comme on le verra plus tard.

le package **tidyverse** qui contient **dplyr** défini de nombreux opérateurs récurrents dans la manipulation de données : filtrage , arrangement , agrégations, groupements. Il est “compatible” avec **sf** , et la combinasion des deux est d'une grande flexibilité.

1.2 Les limites de l'exercice

R ne permet pas de manipulation interactive à la souris. Tout se fait de façon programmatique en définissant les traitements dans un script.

2 Les données

Données ouvertes de consommation de gaz à l'IRIS disponibles sur [https://www.data.gouv.fr/fr/datasets/consommation-annuelle-de-gaz-par-iris-2010-a-2017/#_] Nous ne ferons pas beaucoup d'interprétation thématiques le but étant d'avoir un jeu de données d'une taille assez importante pour jouer avec.

2.1 Chargement des données

Le dataset est lourd : 725 Mo. Nous allons le charger une seule fois, puis nous ferons une selection des données de façon à ne travailler que sur un échantillon.

```
library(sf) #charge le package
library(tidyverse) #charge le package
df <- st_read("./consommation-annuelle-gaz-agregee-maille-iris.geojson")

## Reading layer `consommation-annuelle-gaz-agregee-maille-iris` from data source `/home/paulchapron/consommation-annuelle-gaz-agregee-maille-iris.geojson'
## replacing null geometries with empty geometries
## Simple feature collection with 203338 features and 32 fields (with 27503 geometries empty)
## geometry type:  GEOMETRY
## dimension:      XY
## bbox:            xmin: -4.689486 ymin: 42.40481 xmax: 8.232497 ymax: 51.08899
## geographic CRS: WGS 84
```

La fonction **st_read()** lit les données du fichier dont on donne le chemin d'accès, et s"lectionne le driver approprié en fonction du fichier (shp, geojson, etc.)

Pour connaître le nom des attributs de la couche vecteur on utilise la fonction **names**. La fonction **head** affiche (par défaut) les 6 première lignes d'un dataframe.

```
names(df)

## [1] "tertiaire_nombre_de_pdl"
## [2] "agriculture_consommation_en_mwh"
## [3] "total_nombre_de_pdl"
## [4] "code_region"
## [5] "telereleve_residentiel_secretaire"
## [6] "tertiaire_consommation_en_mwh"
## [7] "residentiel_telereleve"
## [8] "annee_de_reference"
## [9] "code_epci"
## [10] "agriculture_telereleve"
## [11] "consommation_residentielle_secretaire"
## [12] "nom_epci"
## [13] "total_donnees_consommations"
## [14] "residentiel_nombre_de_pdl"
```

```

## [15] "agriculture_nombre_de_pdl"
## [16] "telereleve_non_affecte_secretes"
## [17] "annee"
## [18] "consommation_non_affecte_secretesee"
## [19] "iris"
## [20] "tertiaire_telereleve"
## [21] "residentiel_consommation_en_mwh"
## [22] "nom_region"
## [23] "code_iris"
## [24] "industrie_nombre_de_pdl"
## [25] "industrie_telereleve"
## [26] "industrie_consommation_en_mwh"
## [27] "non_affecte_telereleve"
## [28] "non_affecte_nombre_de_pdl"
## [29] "non_affecte_consommation_en_mwh"
## [30] "residentiel_part_thermosensible"
## [31] "residentiel_thermosensibilité_en_kwh_degc"
## [32] "geo_point_2d"
## [33] "geometry"

head(df)

## Simple feature collection with 6 features and 32 fields (with 2 geometries empty)
## geometry type: POLYGON
## dimension: XY
## bbox: xmin: -3.486859 ymin: 48.53348 xmax: -2.758353 ymax: 48.80272
## geographic CRS: WGS 84
##   tertiaire_nombre_de_pdl agriculture_consommation_en_mwh total_nombre_de_pdl
## 1                      8                         5021                  142
## 2                      4                           NA                  978
## 3                      1                           NA                  397
## 4                     NA                           NA                  21
## 5                     NA                           NA                  0
## 6                      3                           NA                 414
##   code_region telereleve_residentiel_secretese tertiaire_consommation_en_mwh
## 1          53                         False                   5767
## 2          53                         False                   1162
## 3          53                         False                   155
## 4          53                         False                      NA
## 5          53                         False                      NA
## 6          53                         False                  1908
##   residentiel_telereleve annee_de_reference code_epci agriculture_telereleve
## 1                      0%                    2015 200048775                100%
## 2                      4%                    2015 200048775                <NA>
## 3                      0%                    2015 200048775                <NA>
## 4                      0%                    2015 200048775                <NA>
## 5                     <NA>                  2015 200048775                <NA>
## 6                      4%                    2015 242200517                <NA>
##   consommation_residentielle_secretesee
## 1                               False
## 2                               False
## 3                               False
## 4                               False
## 5                               False
## 6                               False

```

```

## nom_epci
## 1 CA Lannion-Trégor Communauté
## 2 CA Lannion-Trégor Communauté
## 3 CA Lannion-Trégor Communauté
## 4 CA Lannion-Trégor Communauté
## 5 CA Lannion-Trégor Communauté
## 6 CA Saint-Brieuc Agglomération Baie d'Armor (Sbaba)

## total_donnees_consommations_residentiel_nombre_de_pdl
## 1 13433 133
## 2 14994 974
## 3 6632 396
## 4 275 21
## 5 0 NA
## 6 11582 411

## agriculture_nombre_de_pdl telereleve_non_affecte_secretises_annee
## 1 1 False 2013
## 2 NA False 2013
## 3 NA False 2013
## 4 NA False 2013
## 5 NA False 2013
## 6 NA False 2013

## consommation_non_affecte_secretisee
## 1 False
## 2 False
## 3 False
## 4 False
## 5 False
## 6 False

##
iris
## 1 Rte de Treguier-Le Rusquet
## 2 Ker Uhel
## 3 Louannec
## 4 IRIS indéterminé
## 5 Somme des agrégats secrétisés EPCI (CA Lannion-Trégor Communauté)
## 6 Centre Ville 1

## tertiaire_telereleve_residentiel_consommation_en_mwh nom_region code_iris
## 1 100% 2645 Bretagne 221130105
## 2 100% 13832 Bretagne 221130107
## 3 100% 6477 Bretagne 221340000
## 4 <NA> 275 Bretagne 221689999
## 5 <NA> NA Bretagne <NA>
## 6 100% 9674 Bretagne 221870101

## industrie_nombre_de_pdl industrie_telereleve industrie_consommation_en_mwh
## 1 NA <NA> NA
## 2 NA <NA> NA
## 3 NA <NA> NA
## 4 NA <NA> NA
## 5 NA <NA> NA
## 6 NA <NA> NA

## non_affecte_telereleve non_affecte_nombre_de_pdl
## 1 <NA> NA
## 2 <NA> NA
## 3 <NA> NA
## 4 <NA> NA

```

```

## 5 <NA> NA
## 6 <NA> NA
## non_affecte_consommation_en_mwh residentiel_part_thermosensible
## 1 NA <NA>
## 2 NA <NA>
## 3 NA <NA>
## 4 NA <NA>
## 5 NA <NA>
## 6 NA <NA>
## residentiel_thermosensibilite_en_kwh_degc geo_point_2d
## 1 <NA> 48.75115, -3.42848
## 2 <NA> 48.746081, -3.470839
## 3 <NA> 48.783678, -3.412389
## 4 <NA>
## 5 <NA>
## 6 <NA> 48.539932, -2.771961
## geometry
## 1 POLYGON ((-3.45827 48.75297...
## 2 POLYGON ((-3.486859 48.7487...
## 3 POLYGON ((-3.377819 48.7918...
## 4 POLYGON EMPTY
## 5 POLYGON EMPTY
## 6 POLYGON ((-2.758353 48.5339...

```

On constate d'emblée que certains codes IRIS sont non attribués (NA). On va commencer par filtrer le datafram pour ne garder que les lignes dont le code IRIS est attribué .

2.2 Filtrage des données : Valeurs NA

La fonction `filter` du package `dplyr` permet de filtrer un datafram selon un critère défini par une `condition` (booléenne), comme dans un `if()` .

Nous allons utiliser la fonction `is.na` qui renvoie comme son nom l'indique TRUE si une valeur est NA , FALSE sinon.

```

x <- NA
is.na(x)

## [1] TRUE

y <- 12
is.na(y)

## [1] FALSE

df_filtered <- df %>% filter(!is.na(code_iris))
cat(nrow(df) - nrow(df_filtered), " lignes retirées")

## 12099 lignes retirées

```

On a filtré ainsi 12099 lignes dans ce dataset.

Au passage nous avons vu:

- la fonction `is.na()` qui teste si une valeur est NA.
- l'opératuer de négation ! boolén
- La fonction `nrow()`retourne le nombre de ligne d'un tableau.
- La fonction `cat()` affiche la version “imprimable” d'une séquence d'objets, séparés par des virgules.

- `dplyr` permet d'écrire les traitements en chaînes, au moyen de l'opérateur `%>%` , qui se lit **pipe** (en anglais hein, cf l'opérateur UNIX `|`), et qui lie les objets aux fonctions. Nous reviendrons sur cette syntaxe plus tard.
- remarquez la concision de l'opération de filtrage, qui tient en une seule ligne.

Autre fonction utile : `anyNA()` qui teste la présence de valeurs NA dans tout un ensemble : dataframe , colonne, ligne , collection, etc.. Nous nous en servons pour vérifier que la colonne `code_iris` ne contient plus de valeurs NA.

```
anyNA(df_filtered$code_iris)
```

```
## [1] FALSE
```

En revanche, le dataframe dans son ensemble contient encore des valeurs NA, dans d'autres colonnes :

```
anyNA(df_filtered)
```

```
## [1] TRUE
```

2.3 Selection d'un sous ensemble de données

2.3.1 Premier affichage

Nous pouvons afficher la géométrie de nos données, mais étant donné le grand nombre d'entités (~200k), et la finesse des mailles IRIS, ce sera **long** et guère lisible. ça risque aussi de faire planter la session R , quin n'aime pas qu'on lui demande d'afficher des milliers d'objets dans une sessions graphique.

Si vous avez pas mal de RAM (8Go) et que vous décidez d'afficher quand même l'objet, cela se fait en exécutant la ligne suivante :

```
plot(st_geometry(df_filtered))
```

Vous obtiendrez quelque chose comme ça :



2.3.2 Sélection d'une région

Nous allons sélectionner une région et constituer le sous ensemble des IRIS de cette région.

On commence par lister les valeurs des régions dans le dataframe. Le nom des régions est inscrit dans la colonne `nom_region` :

```
unique(df_filtered$nom_region)
```

```
## [1] Bretagne           Centre-Val de Loire
## [3] Grand Est          Hauts-de-France
## [5] Ile-de-France      Normandie
## [7] Bourgogne-Franche-Comté Nouvelle-Aquitaine
## [9] Occitanie          Pays de la Loire
## [11] Provence-Alpes-Côte d'Azur Auvergne-Rhône-Alpes
## [13] ALSACE             AQUITAIN
```

```

## [15] OCCITANIE          HAUTS-DE-FRANCE
## [17] PROVENCE-ALPES-COTE D'AZUR BRETAGNE
## [19] CENTRE-VAL DE LOIRE    ILE-DE-FRANCE
## [21] NORMANDIE          CENTRE
## [23] CHAMPAGNE-ARDENNE    FRANCHE-COMTE
## [25] HAUTE-NORMANDIE     NORD-PAS-DE-CALAIS
## [27] NOUVELLE-AQUITAINE  AUVERGNE-RHONE-ALPES
## [29] BOURGOGNE-FRANCHE-COMTE PAYS DE LA LOIRE
## [31] LANGUEDOC-ROUSSILLON LIMOUSIN
## [33] LORRAINE            PICARDIE
## [35] POITOU-CHARENTES    GRAND EST
## [37] RHONE-ALPES         MIDI-PYRENEES
## [39] AUVERGNE           BASSE-NORMANDIE
## [41] BOURGOGNE         

## 41 Levels: ALSACE AQUITAINNE AUVERGNE ... RHONE-ALPES

```

La fonction `unique()` retire les doublons d'une collection. C'est l'équivalent d'un `SELECT DISTINCT` en SQL.

On pourrait écrire la même chose avec la syntaxe des pipes de `dplyr` :

```

df_filtered$nom_region %>% unique()

## [1] Bretagne          Centre-Val de Loire
## [3] Grand Est        Hauts-de-France
## [5] Ile-de-France    Normandie
## [7] Bourgogne-Franche-Comté Nouvelle-Aquitaine
## [9] Occitanie        Pays de la Loire
## [11] Provence-Alpes-Côte d'Azur Auvergne-Rhône-Alpes
## [13] ALSACE           AQUITAINNE
## [15] OCCITANIE        HAUTS-DE-FRANCE
## [17] PROVENCE-ALPES-COTE D'AZUR BRETAGNE
## [19] CENTRE-VAL DE LOIRE    ILE-DE-FRANCE
## [21] NORMANDIE          CENTRE
## [23] CHAMPAGNE-ARDENNE    FRANCHE-COMTE
## [25] HAUTE-NORMANDIE     NORD-PAS-DE-CALAIS
## [27] NOUVELLE-AQUITAINE  AUVERGNE-RHONE-ALPES
## [29] BOURGOGNE-FRANCHE-COMTE PAYS DE LA LOIRE
## [31] LANGUEDOC-ROUSSILLON LIMOUSIN
## [33] LORRAINE           PICARDIE
## [35] POITOU-CHARENTES   GRAND EST
## [37] RHONE-ALPES         MIDI-PYRENEES
## [39] AUVERGNE           BASSE-NORMANDIE
## [41] BOURGOGNE         

## 41 Levels: ALSACE AQUITAINNE AUVERGNE ... RHONE-ALPES

```

Qui peut se lire ainsi : «la colonne `nom_region` du dataframme `df_filtered` est envoyée à la fonction `unique()`»

On voit aussi que certaines régions sont en double et que les niveaux régionaux varient: les noms sont à la fois inscrits en majuscules et en minuscules, et certaines sous-régions sont présentes.

On va choisir une région (celle que vous voulez), moi je prends la région **Grand Est** (Lorraine, Alsace, Champagne-Ardennes). Les objets qui m'intéressent sont donc ceux dont l'attribut `nom_region` vaut :

- GRAND EST, ou
- Grand Est, ou
- LORRAINE, ou
- ALSACE, ou

- CHAMPAGNE-ARDENNE

Pour sélectionner les données du Grand Est , il nous faut une condition de filtrage. On va utiliser l'opérateur ensembliste `%in%` qui teste si un élément appartient à un ensemble de valeurs

Voici un exemple d'utilisation

```
"donald" %in% c("riri", "fifi", "loulou")  
## [1] FALSE  
8 %in% c(2,4,6,8,10,12)  
  
## [1] TRUE  
3.14 %in% c("PI", 3.14, "pi")  
  
## [1] TRUE
```

Au passage on note la fonction `c()` qui permet de créer des vecteurs par **extension** (i.e. en donnant les valeurs)

Dans notre cas pour ne conserver que les IRIS dont l'attribut `nom_region` appartient aux valeurs qui nous intéressent, le traitement s'écrit :

```
df_GE <- df_filtered %>% filter(nom_region %in% c("GRAND EST", "Grand Est", "LORRAINE",  
"ALSACE", "CHAMPAGNE-ARDENNE" ))  
nrow(df_GE)  
  
## [1] 16257
```

le dataframe `df_GE` contient ~16k entités spatiales, ce qui est plus raisonnable pour commencer à afficher les objets.

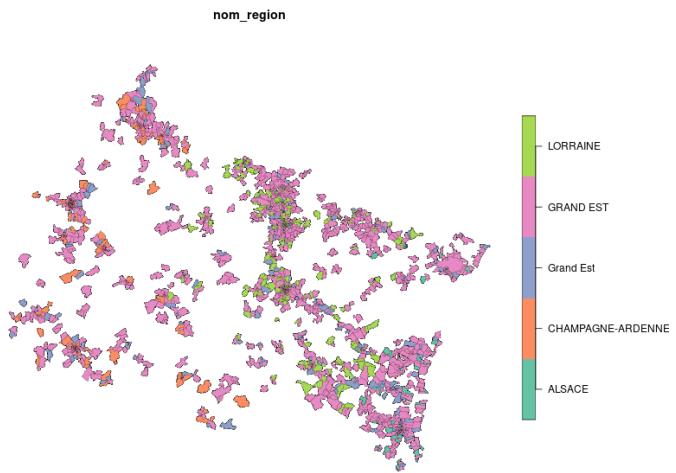
2.3.3 Carte choroplète d'un facteur

Voyons rapidement comment afficher les géométries colorées par un attribut avec la fonction `plot` , redéfinie par le package `sf`. Cette fonction prend en argument soit une géométrie, soit un attribut d'un objet `sf`, mais **sans** l'opérateur `$` , uniquement avec l'opérateur `[]` et le nom de l'attribut entre quotes.

```
plot(st_geometry(df_GE))
```



```
df_GE$nom_region <- droplevels(df_GE$nom_region)  
plot(df_GE["nom_region"], lwd = 0.2, key.width = lcm(6))
```



Pour comprendre les arguments supplémentaires de la fonction `plot()` pour l'épaisseur des arêtes des polygones et la taille de la légende , reportez-vous à la documentation.

À noter : l'attribut `nom_region` est de type `factor`. (Pour connaître le type d'un attribut , utilisez la fonction `class()`). R conserve les ** valeurs de tous les facteurs possibles** (les modalités en français, en R ce sont des `levels`) des attributs de type `factor`. Ici nous avons un sous-ensemble du dataframe complet, dont l'attribut `nom_region` appartient à 5 modalités : “GRAND EST”, “Grand Est”, “LORRAINE”, “ALSACE”, “CHAMPAGNE-ARDENNE” : nous l'avons filtré dans ce but.

Mais lorsqu'on observe les modalités de l'attribut `nom_region`, on constate qu'il a 41 modalités :

```
levels(df_GE$nom_region)
```

```
## [1] "ALSACE"                      "AQUITAINE"
## [3] "AUVERGNE"                     "AUVERgne-RHôNE-ALPES"
## [5] "Auvergne-Rhône-Alpes"          "BASSE-NORMANDIE"
## [7] "BOURGOGNE"                    "BOURGOGNE-FRANCHE-COMTE"
## [9] "Bourgogne-Franche-Comté"       "Bretagne"
## [11] "BRETAGNE"                     "CENTRE"
## [13] "Centre-Val de Loire"          "CENTRE-VAL DE LOIRE"
## [15] "CHAMPAGNE-ARDENNE"             "FRANCHE-COMTE"
## [17] "Grand Est"                   "GRAND EST"
## [19] "HAUTE-NORMANDIE"              "Hauts-de-France"
## [21] "HAUTS-DE-FRANCE"               "Ile-de-France"
## [23] "ILE-DE-FRANCE"                 "LANGUEDOC-ROUSSILLON"
## [25] "LIMOUSIN"                     "LORRAINE"
## [27] "MIDI-PYRENEES"                "NORD-PAS-DE-CALAIS"
## [29] "Normandie"                   "NORMANDIE"
## [31] "Nouvelle-Aquitaine"            "NOUVELLE-AQUITAINE"
## [33] "Occitanie"                    "OCCITANIE"
## [35] "Pays de la Loire"              "PAYS DE LA LOIRE"
## [37] "PICARDIE"                     "POITOU-CHARENTES"
## [39] "PROVENCE-ALPES-COTE D'AZUR"   "Provence-Alpes-Côte d'Azur"
## [41] "RHONE-ALPES"
```

Or, lorsqu'on cherche à afficher l'attribut de type `factor` d'un objet spatial, R et `sf` utilisent toutes les modalités pour tracer le cartouche de légende pour les couleurs. Pour éviter d'avoir un cartouche avec 41 valeurs de couleurs, (ils y a 41 modalités pour le facteur `nom_region` dans le dataframe complet) nous

redéfinissons les modalités de l'attributs `nom_region` en ne conservant que les modalités présentes dans notre sous-ensemble, par la fonction `droplevels()`, dont on se sert pour reaffecter l'attribut `nom_region` du dataframe `df_GE`.

2.4 Système de coordonnées de référence

Pour connaître le CRS d'un objet spatial, s'il est défini, on utilise la fonction `st_crs()` :

```
st_crs(df_GE)
```

```
## Coordinate Reference System:  
##   User input: WGS 84  
##   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]],  
##                     ID["EPSG",4326]]
```

Pour reprojecter l'objet, on utilise la fonction `st_transform()` avec en argument le code EPSG du CRS désiré. Ici , on définit un Lambert-93 (code EPSG 2154) et on reaffecte le produit de la transformation au dataframe lui-même :

```
df_GE <- st_transform(df_GE, 2154)  
st_crs(df_GE)
```

```
## Coordinate Reference System:  
##   User input: EPSG:2154  
##   wkt:  
## PROJCRS["RGF93 / Lambert-93",  
##          BASEGEOGCRS["RGF93",  
##                         DATUM["Reseau Geodesique Francais 1993",  
##                                       ELLIPSOID["GRS 1980",6378137,298.257222101,  
##                                                 LENGTHUNIT["metre",1]],  
##                         PRIMEM["Greenwich",0,  
##                                   ANGLEUNIT["degree",0.0174532925199433]],  
##                         ID["EPSG",4171]],  
##          CONVERSION["Lambert-93",  
##                      METHOD["Lambert Conic Conformal (2SP)",  
##                                ID["EPSG",9802]],  
##                      PARAMETER["Latitude of false origin",46.5,  
##                                 ANGLEUNIT["degree",0.0174532925199433],  
##                                 ID["EPSG",8821]],  
##                      PARAMETER["Longitude of false origin",3,  
##                                 ANGLEUNIT["degree",0.0174532925199433],  
##                                 ID["EPSG",8822]],  
##                      PARAMETER["Latitude of 1st standard parallel",49,
```

```

##           ANGLEUNIT["degree",0.0174532925199433],
##           ID["EPSG",8823]],
##           PARAMETER["Latitude of 2nd standard parallel",44,
##           ANGLEUNIT["degree",0.0174532925199433],
##           ID["EPSG",8824]],
##           PARAMETER["Easting at false origin",700000,
##           LENGTHUNIT["metre",1],
##           ID["EPSG",8826]],
##           PARAMETER["Northing at false origin",6600000,
##           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["France"],
##               BBOX[41.15,-9.86,51.56,10.38]],
##               ID["EPSG",2154]]

```

On constate que l'attribut CRS de l'objet spatial a bien été changé.

3 Cartographie directe d'un attribut

3.1 Carte choroplète d'un facteur avec ggplot

Voici comment obtenir une carte choroplète de l'attribut nom_region avec ggplot2

```
table(df_GE$nom_region)
```

```

##          ALSACE          AQUITAIN
##          395              0
##          AUVERGNE        AUVERGNE-RHONE-ALPES
##          0                  0
##          Auvergne-Rhône-Alpes      BASSE-NORMANDIE
##          0                  0
##          BOURGOGNE      BOURGOGNE-FRANCHE-COMTE
##          0                  0
##          Bourgogne-Franche-Comté      Bretagne
##          0                  0
##          BRETAGNE          CENTRE
##          0                  0
##          Centre-Val de Loire      CENTRE-VAL DE LOIRE
##          0                  0
##          CHAMPAGNE-ARDENNE      FRANCHE-COMTE
##          608                  0
##          Grand Est          GRAND EST
##          10171                4052
##          HAUTE-NORMANDIE      Hauts-de-France
##          0                  0

```

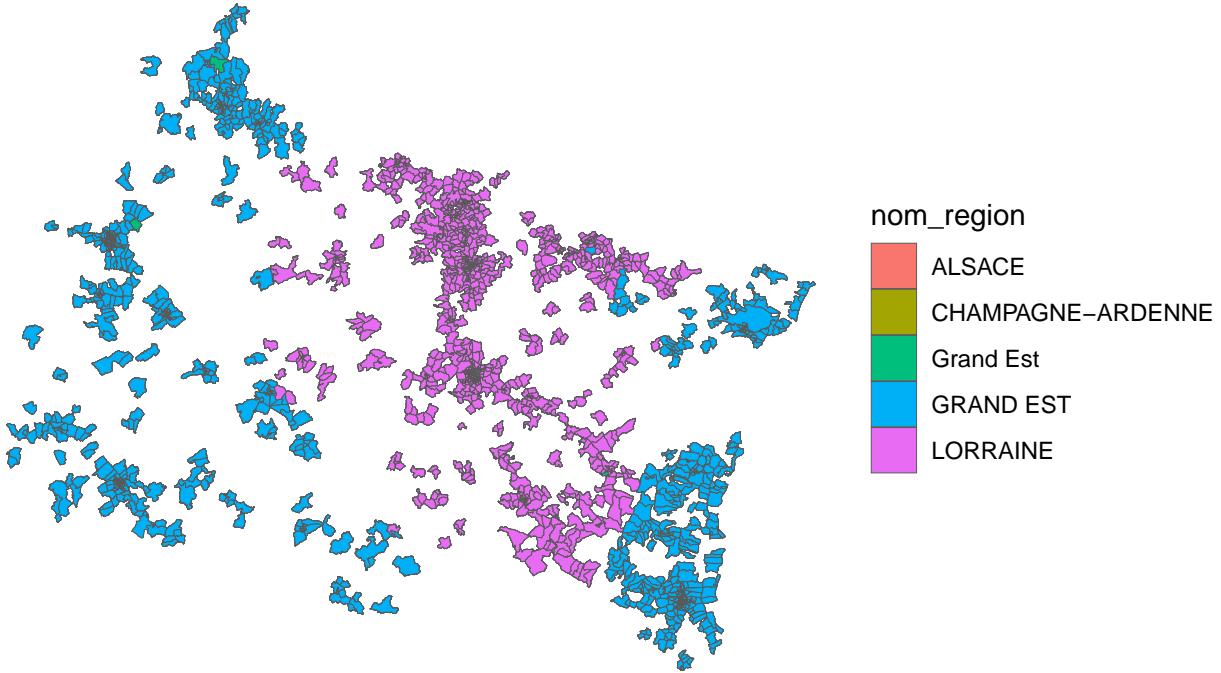
```

##          HAUTS-DE-FRANCE           Ile-de-France
##          0                           0
##          ILE-DE-FRANCE           LANGUEDOC-ROUSSILLON
##          0                           0
##          LIMOUSIN                 LORRAINE
##          0                           1031
##          MIDI-PYRENEES           NORD-PAS-DE-CALAIS
##          0                           0
##          Normandie                NORMANDIE
##          0                           0
##          Nouvelle-Aquitaine      NOUVELLE-AQUITAINE
##          0                           0
##          Occitanie                OCCITANIE
##          0                           0
##          Pays de la Loire         PAYS DE LA LOIRE
##          0                           0
##          PICARDIE                 POITOU-CHARENTES
##          0                           0
## PROVENCE-ALPES-COTE D'AZUR Provence-Alpes-Côte d'Azur
##          0                           0
##          RHONE-ALPES
##          0

p1 <- ggplot(df_GE) +
  geom_sf(data=df_GE ,aes(geometry = geometry , fill=nom_region), size=0.1) +
  #scale_fill_discrete(name = "Valeur brute de \n l'attribut nom_region") +
  theme_void()

p1

```



La syntaxe de ggplot est assez particulière , reportez-vous à l'excellent site du MTES et sa formation à R , notamment le chapitre 5 qui parle de ggplot2 pour un panorama plus complet: [<https://mtes-mct.github.io/parcours-r/m5/index.html>]

3.2 Carte choroplète d'un facteur avec `ggplot` (bis)

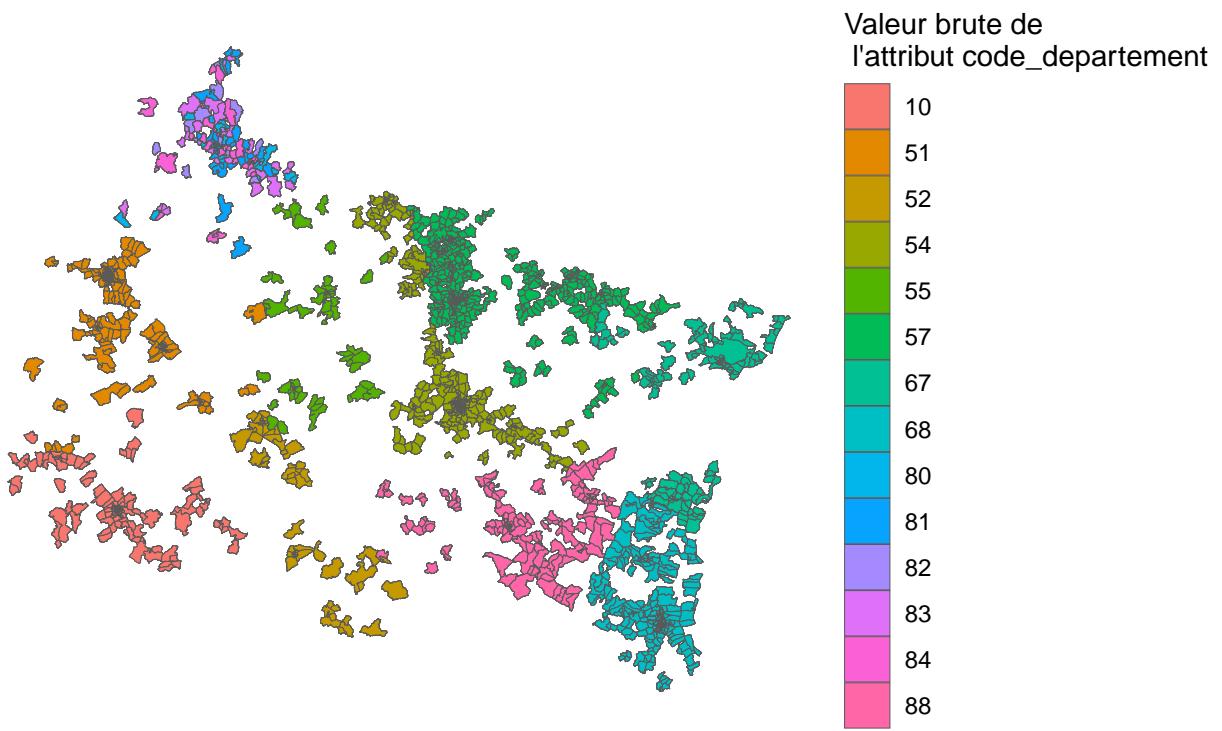
Visiblement l'attribut `nom_region` est assez peu informatif et mal renseigné : tout est dans la région Grand Est, il y a des doublons et certaines valeurs sont le nom de régions datant d'avant la réforme territoriale. On va cartographier les IRIS en les colorant par département

Pour obtenir les départements , on examine les attributs du dataframe. Les deux premiers chiffres du code IRIS correspondent au numéro de département. Pour le récupérer, on convertit l'attribut `code_iris` en chaîne de caractère (avec la fonction `as.character()`), puis on coupe cette chaîne de caractères pour ne garder que les 2 premiers caractères, avec la fonction `substr()`. On stocke le résultat de cette chaîne de traitement (notez à nouveau l'utilisation du pipe `%>%` pour enchaîner les traitements) dans un nouvel attribut : `code_departement`

```
df_GE$code_departement <- df_GE$code_iris %>% as.character() %>% substr(1,2)

p3 <- ggplot()+
  geom_sf(data=df_GE ,aes(fill=code_departement), size=0.1)+
  scale_fill_discrete(name = "Valeur brute de \n l'attribut code_departement ") +
  theme_void()

p3
```



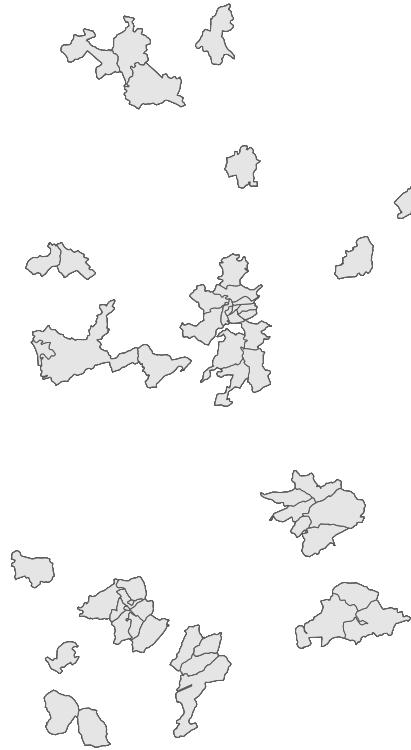
Cela produit une carte un peu plus utile, même si ce n'est pas le but de ce tutoriel.

3.3 Sous-ensembles des IRIS de Meuse

Nous allons encore réduire la taille de notre échantillon pour ne garder que les IRIS du département de la Meuse. Le code de département de la Meuse est le 55. Nous filtre le dataframme du Grand-Est pour ne conserver que les IRIS dont l'attribut `code_departement` vaut 55. (Notez que l'attribut `code_departement` est de type `character`, d'où le test d'égalité avec la valeur "55".)

```
df_55 <- df_GE %>% filter(code_departement=="55")
p4 <- ggplot()+
  geom_sf(data=df_55 , size=0.1)+
  labs(title = "Les IRIS de Meuse")+
  theme_void()
p4
```

Les IRIS de Meuse



3.4 Carte à symbole proportionnels d'une quantité

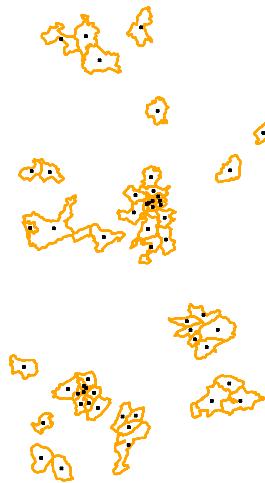
Comme vous le savez, on ne représente pas une variable de stock par un aplat de couleur, la carte choroplèthe est donc exclue. Nous allons créer des cercles proportionnels à une quantité (l'attribut `total_données_consommations`), que nous allons superposer aux polygônes des IRIS.

3.4.1 centroïdes des IRIS

Nous allons localiser les cercles proportionnels sur les centroïdes des polygônes des IRIS. Pour créer les centroïdes , on utilise la fonction `st_centroid()` Cette fonctionne pas si des polygons sont vides, ce qui est le cas de quelques polygon du datafram `df_55`. On les enlève par filtrage en utilisant le prédicat `st_is_empty()`

```
df_55 <- df_55 %>% filter(!st_is_empty(geometry))
centroïdes <- st_centroid(df_55)

## Warning in st_centroid.sf(df_55): st_centroid assumes attributes are constant
## over geometries of x
plot(st_geometry(df_55), border="orange")
plot(st_geometry(centroïdes), add=T, pch=4, cex= 0.1)
```



4 Manipulations des géométries

la liste complètes des opérateurs “purement” géométriques de `sf` est disponibles ici : [<https://r-spatial.github.io/sf/articles/sf3.html#geometrical-operations-1>]

4.1 Jointure spatiale

4.2 Union de géométries

5 Manipulations basées sur les attributs