Géomatique avec R. Manipuler, analyser et représenter des données géographiques

Olivier Delaigue

GÉOMATIQUE AVEC R

Manipuler, analyser et représenter des données géographiques

Olivier Delaigue

HYDRO

irstea

Manipuler, analyser et représenter des données géographiques avec R

Olivier Delaigue

9 décembre 2016

Sommaire

Introduction

Packages spatiaux

Dans R, de nombreux packages permettent de traiter, de près ou de loin, des questions relatives à la manipulation de données spatiales (tab. 1). Il peut s'agir de packages ayant trait à la manipulation d'objets spatiaux, à l'analyse spatiale, à la géostatistique, à la représentation cartographique, etc.

Tableau 1 – Liste de packages R relatifs aux traitements de données spatiales (168 recensés au 15/09/2016 par le package `ctv` ; core : maintenance effectuée par la R Core Team).

ade4	geonames	mapview	RgoogleMaps	spatial.tools
adehabitat	geoR (core)	marmap	rgrass7	spatstat (core)
adehabitatHR	geoRglm	MBA	RNetCDF	spatsurv
adehabitatHS	georob	McSpatial	rpostgis	spBayes
adehabitatLT	geospacom	micromap	RPyGeo	spBayesSurv
adehabitatMA	geosphere	ModelMap	RQGIS	spcosa
ads	geospt	ncdf4	RSAGA	spdep (core)
akima	geostatsp	ncf	RSurvey	spgrass6
AMOEBA	GeoXp	ngspatial	rtop	spgwr
ash	ggmap	nlme	rworldmap	sphet
aspace	glmmBUGS	OasisR	rworldxtra	splancs (core)
automap	gmt	OpenStreetMap	S2sls	splm
CARBayes	Grid2Polygons	osmar	seg	spselect
cartography	GriegSmith	pastecs	sgeostat	spsurvey
classInt (core)	gstat (core)	PBSmapping	shapefiles	spTimer
cleangeo	Guerry	PBSmodelling	shp2graph	SSN
CompRandFld	GWmodel	plotGoogleMaps	siplab	statebins
constrainedKriging	gwrr	plotKML	sp (core)	Stem
cshapes	hdeco	postGIStools	spacetime (core)	stplanr
dbmss	HSAR	PReMiuM	spacom	taRifx
DCluster (core)	igraph	ProbitSpatial	spaMM	tgp
deldir (core)	intamap	psgp	spanel	tmap
dggridR (core)	ipdw	quickmapr	sparr	trip
diseasemapping	landsat	ramps	spatcounts	tripack
DSpat	latticeDensity	RandomFields (core)	spatgraphs	tripEstimation
ecespa	lawn	rangeMapper	spatial	UScensus2000cdp
fields	lctools	RArcInfo	spatialCovariance	UScensus2000tract
FieldSim	leafletR	raster (core)	SpatialEpi	vardiag
gdalUtils	magclass	rasterVis	SpatialExtremes	vec2dtransf
gdistance	mapdata	RColorBrewer (core)	spatialkernel	vegan
Geneland	mapmisc	recmap	SpatialPosition	Watersheds
geoaxe	mapproj	regress	spatialprobit	wkb
GEOmap	maps	rgdal (core)	spatialsegregation	
geomapdata	maptools (core)	rgeos (core)	SpatialTools	

Étant donné le nombre de packages concernés, le but n'est nullement de les étudier tous. Dans cet ouvrage, seront donc passées en revue uniquement certaines fonctionnalités des packages les plus usités. Dans un premier temps, nous nous intéresserons aux questions relatives à la création et à la manipulation d'objets spatiaux, ainsi qu'à la lecture et l'écriture de fichiers spatiaux. Par la suite, passerons en revues les fonctionnalités des packages permettant de réaliser les principales analyses spatiales dans R, puis celles des packages wrappers, qui permettent de communiquer entre R et des bibliothèques de géomatique externes ou des logiciels de système d'information géographique. Enfin, nous aborderons les différentes possibilités proposées par R pour réaliser des représentations cartographiques ; il sera alors également question de l'utilisation de fonds de cartes statiques et dynamiques de types Google Maps ou OpenStreetMaps. En revanche, dans cet ouvrage, les questions relatives à la géostatistique ne seront pas abordées.

Système de coordonnées de référence

En cartographie, un système de coordonnées est un référentiel dans lequel on peut représenter des éléments dans l'espace. Ce système permet de se situer sur l'ensemble du globe terrestre grâce à un couple de coordonnées géographiques. En anglais, il est dénommé *coordinate reference system* (CRS) ou *spatial reference system* (SRS).

Dans R, c'est le package `rgdal` (Bivand *et al.*, 2014) qui permet de gérer les systèmes de coordonnées. Ce package utilise les fonctionnalités des bibliothèques de géomatique GDAL (GDAL Development Team, 2012) et PROJ.4 (Evenden *et al.*, 2015).

```r
library(rgdal)
```

La liste des différents systèmes de coordonnées disponibles dans R est accessible grâce la fonction **make_EPSG()**.

```r
EPSG <- make_EPSG()
names(EPSG)
## [1] "code" "note" "prj4"
head(EPSG[grep("RGF", EPSG$note), c("code", "note")])
##      code                    note
## 104  4171                 # RGF93
## 281  4624                # RGFG95
## 617  2154    # RGF93 / Lambert-93
## 1427 2972 # RGFG95 / UTM zone 22N
## 1767 3313 # RGFG95 / UTM zone 21N
## 2312 3942          # RGF93 / CC42
```

L'EPSG (*European Petroleum Survey Group*) [1] est un groupe ayant été créé en 1985 par Jean-Patrick Girbig, travaillant alors pour ELF Aquitaine. Ce groupe a défini une liste des systèmes de coordonnées géoréférencés et leur a associé des codes pour les identifier. En 2005, il est devenu le Comité de topographie et de positionnement (*Surveying and Positionning Comittee*) de l'Association internationale des producteurs de pétrole et de gaz (OGP). Ces codes, qui existent toujours sous l'appellation de "codes EPSG", sont notamment utilisés dans les standards de l'*Open Geospatial Consortium* (OGC) [2]. Jean-Patrick Girbig a également fondé l'*Americas Petroleum Survey Group* (APSG) [3], avec des objectifs semblables à ceux de l'EPSG créé 10 ans plus tôt.

Un système géodésique est un système de référence permettant d'exprimer les positions au voisinage de la Terre. La projection cartographique est un ensemble de techniques géodésiques permettant de représenter la surface de la Terre dans son ensemble ou en partie sur la surface plane d'une carte. Une projection s'appuie sur une sphère ou un ellipsoïde de révolution qui sont des modèles plus ou moins proches de la forme patatoïde réelle. On commence par choisir, à partir de son géoïde global, un ellipsoïde de révolution représentatif.

Un système géodésique peut recevoir plusieurs codes EPSG selon son utilisation. Ainsi, le système géodésique officiel "Réseau géodésique français" RGF93, valide en métropole, a pour code EPSG 6171. C'est un système de coordonnées géocentrique. L'ellipsoïde associé est nommée IAG GRS 1980. Lorsqu'un code EPSG est noté "géographique 2D", cela signifie que le système géodésique est réduit à la latitude et à la longitude. Lorsqu'il est noté "géographique 3D", cela signifie qu'il gère la latitude, la longitude et la hauteur sur l'ellipsoïde.

Le Registre des paramètres géodésiques EPSG (*Geodetic Parameter Registry* [4]) permet de trouver les systèmes de coordonnées en un lieu, ou une zone, qui correspond à un code EPSG.

Voici quelques-uns des codes EPSG des systèmes de coordonnées les plus utilisés en France :
- 2154 : *Lambert 93*, qui est la projection officielle pour les cartes de France métropolitaine depuis le décret du 26 décembre 2000 ;
- 27572 : *Lambert II Carto* (Centre), qui est l' ancienne projection officielle pour les cartes de France métropolitaine ;
- 3035 : *ETRS89 Lambert Azimuthal Equal-Area* (LAEA), qui est la projection officielle pour les cartes de l'Union européenne depuis le décret du 3 janvier 2000.

Voici quelques-uns des codes EPSG des systèmes de coordonnées mondiaux :
- 4326 : *World Geodetic System 1984* (WGS 84), qui est le système géodésique mondial révisé en 1984 ;
- 3857 : *Web Mercator* (ou *WGS 84/Pseudo-Mercator*) est un système de projection qui a été popularisé par Google Maps, puis par d'autres services Web tels qu'OpenStreetMap ou Bing Maps (il est utilisé par la plupart des principaux fournisseurs de carte en ligne) ;
- 54009 : *Mollweide* (EPSG non défini dans R) ;
- 54030 : *Robinson* (EPSG non défini dans R) ;
- 31259 : *Werner*.

1. Site web de l'EPSG : http://www.epsg.org/.
2. Site web de l'OGC : http://www.opengeospatial.org/.
3. Site web de l'APSG : http://www.apsg.info/.
4. Site web de l'EPSG Geodetic Parameter Dataset : http://www.epsg-registry.org/.

Sous R, la gestion des coordonnées spatiales est basée sur les produits de l'*Open Source Geospatial Foundation*[5]. Comme ces derniers, R utilise PROJ.4[6], une bibliothèque servant à effectuer des conversions d'une projection cartographique à une autre. Cette bibliothèque est basée sur les travaux menés par Evenden *et al.* à l'USGS (2015), mais il s'agit à présent d'un projet OSGeo maintenu par Frank Warmerdam.

Dans R, on utilise la fonction **CRS()**, pour accéder aux différents paramètres d'un système de coordonnées référencé à partir d'un code ESPG donné.

```
CRS("+init=epsg:2154")
## CRS arguments:
##  +init=epsg:2154 +proj=lcc +lat_1=49 +lat_2=44 +lat_0=46.5 +lon_0=3 +x_0=700000
## +y_0=6600000 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs
```

Si le code EPSG n'existe pas ou n'est pas référencé dans R, on peut simplement définir les différents paramètres en utilisant la syntaxe PRO4.J. Voici un exemple avec la projection *Mollweide*, qui n'est pas définie dans la liste renvoyée par la fonction **make_EPSG()**.

```
CRS("+proj=moll +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs")
## CRS arguments:
##  +proj=moll +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs +ellps=WGS84
## +towgs84=0,0,0
```

Attention, la chaîne de caractères définissant les paramètres doit être écrite en minuscules. En effet, sous Linux, si la casse de caractères n'est pas respectée, le système de coordonnées ne sera pas reconnu.

Les principaux paramètres PROJ.4 sont les suivants[7] :

- **epsg** : le code EPSG ;
- **proj** : l'acronyme du type de projection (*longlat* pour les systèmes géodésiques) ;
- **lat_0** : ϕ_0, l'origine géographique du parallèle central ;
- **lat_1** : l'origine géographique du premier parallèle de référence ;
- **lat_2** : l'origine géographique du second parallèle de référence ;
- **lat_ts** : la latitude de l'échelle véritable ;
- **lon_0** : λ_0, l'origine géographique du méridien central ;
- **k_0** : le facteur d'échelle (ancien nom) ;
- **k_1** : le facteur d'échelle (nouveau nom) ;
- **x_0** : x_0, l'abscisse fictive, ajoutée à la valeur x des coordonnées cartésiennes (utilisée dans les systèmes de grille pour éviter les coordonnées négatives de grille) ;
- **y_0** : y_0, l'ordonnée fictive, ajoutée à la valeur y des coordonnées cartésiennes (utilisée dans les systèmes de grille pour éviter les coordonnées négatives de grille) ;
- **ellps** : l'ellipsoïde utilisée ;
- **a** : l'axe majeur de l'ellipse ;
- **b** : l'axe mineur de l'ellipse ;
- **towgs84** : indique jusqu'à 7 paramètres de transformation Bursa-Wolf (ces paramètres peuvent être utilisés pour approximer une transformation de la donnée horizontale pour le système géodésique WGS 84) ;
- **pm** : le méridien alternatif (généralement un nom de ville) ;
- **units** : l'unité (angulaire ou linéaire) ;
- **zone** : la zone UTM (pour *Universal Transverse Mercator*).

La fonction **projInfo()** du package **rgdal** permet d'accéder à certains attributs des systèmes de coordonnées, tels que la projection, donnée et ellipsoïde :

```
head(projInfo(type = "proj"))
##     name                  description
## 1    aea              Albers Equal Area
## 2   aeqd          Azimuthal Equidistant
## 3   airy                          Airy
## 4 aitoff                        Aitoff
## 5   alsk Mod. Stererographics of Alaska
## 6  apian               Apian Globular I
head(projInfo(type = "ellps"))
##     name         major              ell               description
## 1  MERIT   a=6378137.0       rf=298.257                MERIT 1983
## 2  SGS85   a=6378136.0       rf=298.257 Soviet Geodetic System 85
## 3  GRS80   a=6378137.0 rf=298.257222101      GRS 1980(IUGG, 1980)
## 4  IAU76   a=6378140.0       rf=298.257                  IAU 1976
## 5   airy a=6377563.396    b=6356256.910                 Airy 1830
## 6 APL4.9  a=6378137.0.        rf=298.25       Appl. Physics. 1965
```

5. Site web de l'OSGeo : http://www.osgeo.org/.
6. Site web de PROJ.4 : http://trac.osgeo.org/proj/.
7. Pour plus de détails et pour consulter la liste complète des paramètres : https://trac.osgeo.org/proj/wiki/GenParms/.

```
head(projInfo(type = "datum"))
##       name ellipse                                        definition
## 1    WGS84   WGS84                                     towgs84=0,0,0
## 2   GGRS87   GRS80                        towgs84=-199.87,74.79,246.62
## 3    NAD83   GRS80                                     towgs84=0,0,0
## 4    NAD27  clrk66 nadgrids=@conus,@alaska,@ntv2_0.gsb,@ntv1_can.dat
## 5  potsdam  bessel   towgs84=598.1,73.7,418.2,0.202,0.045,-2.455,6.7
## 6 carthage clark80                          towgs84=-263.0,6.0,431.0
##                            description
## 1
## 2 Greek_Geodetic_Reference_System_1987
## 3            North_American_Datum_1983
## 4            North_American_Datum_1927
## 5          Potsdam Rauenberg 1950 DHDN
## 6                Carthage 1934 Tunisia
```

Pour connaître la version de PROJ.4 utilisée par le package **rgdal**, il faut utiliser la commande suivante :

```
getPROJ4VersionInfo()
## [1] "Rel. 4.8.0, 6 March 2012, [PJ_VERSION: 480]"
```

Dans R, il n'est pas obligatoire de définir le système de coordonnées des objets spatiaux. Dans ce cas, les objets présentent bien des coordonnées qui sont dans un système, mais ce dernier n'est pas identifié. Ceci ne pose pas de problème majeur pour réaliser des représentations cartographiques avec plusieurs objets, tant que ceux-ci ont bien été définis dans le même système. Sauf exception (§ 6.3.4), il n'existe pas de "projection à la volée" dans R. En revanche, pour pouvoir convertir un objet dans un système de coordonnées souhaité, il est indispensable que le système initial soit bien défini, sans quoi aucune transformation n'est évidemment possible. Par ailleurs, pour les traitements spatiaux, toutes les couches doivent être dans le même système de coordonnées, sans quoi toute analyse spatiale est impossible.

Deuxième partie

Manipulation des objets spatiaux

Chapitre 1

Structure des données spatiales

Dans R, il existe plusieurs formats de données spatiales. Les plus fréquemment utilisés sont ceux de la classe **sp** (objets vectoriels ou matriciels) et de la classe **raster** (objets matriciels), dont les méthodes sont définies par les packages **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a) et **raster** (Hijmans, 2015). Ces objets présentent la particularité d'être de mode **S4**. Le **S4**, contrairement au **S3** (la langage habituel de R), permet l'utilisation de fonctions qui permettent de considérer R comme un langage objet. Par extension, **S4** désigne aussi la programmation orientée objet sous R. L'accès aux attributs des objets **S4** se fait en utilisant les syntaxes **object@name** ou **slot(object, name)** ; c'est l'équivalent de **object$name** et **getElement(object, name)** en **S3**.

Il existe d'autres formats pour stocker les données vectorielles spatiales, mais ils sont souvent moins pratiques, et utilisés par un nombre restreint de packages. Parmi les packages le plus connus définissant un autre format de données spatiales, on peut compter **maps** (Becker R.A. (original S code) *et al.*, 2014b), **PBSmapping** (Schnute *et al.*, 2015, 2004) et **spatstat** (Baddeley & Turner, 2005 ; Baddeley *et al.*, 2013). Les classes d'objets gérés par ces packages, toutes de mode **S3**, seront décrites brièvement en fin de partie (§ 1.3).

1.1 Données vectorielles

Dans R, le format de données vectorielles le plus fréquemment utilisé est le format **sp**. C'est le package **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a) qui définit les méthodes permettant de créer et de manipuler ces objets spatiaux.

Selon le type d'entité géographique considéré, une classe secondaire est définie :
- **SpatialPoints** pour les points ;
- **SpatialMultiPoints** pour les groupes multiples de points ;
- **SpatialLines** pour les lignes ;
- **SpatialPolygons** pour les polygones.

Si des données attributaires sont liées aux entités géographiques, les objets sont d'une autre classe, dont le nom est constitué du même morphème, auquel s'ajoute le suffixe "**DataFrame**" :
- **SpatialPointsDataFrame** ;
- **SpatialMultiPointsDataFrame** ;
- **SpatialLinesDataFrame** ;
- **SpatialPolygonsDataFrame**.

Quel que soit le type d'entité considéré, les objets de la classe **sp** sont constitués de certains attributs communs :
- **bbox** : l'emprise géographique de la couche ;
- **proj4string** : le système de coordonnées de la couche ;
- **data** : la table attributaire (pour les classes **Spatial*DataFrame**).

D'autres attributs sont spécifiques à chaque type d'entité géographique.
Un objet **SpatialPoints(DataFrame)** possède les attributs spécifiques suivants :
- **coords** : la matrice contenant les coordonnées géographiques de chaque point ;
- **coords.nrs** : la position des colonnes de coordonnées dans le tableau initial (uniquement pour les objets **SpatialPointsDataFrame**).

Un objet **SpatialMultiPoints(DataFrame)** possède l'attribut spécifique suivant :
- **coords** : la liste des matrices contenant les coordonnées géographiques de chaque couche de points.

Un objet **SpatialLines(DataFrame)** possède l'attribut spécifique suivant :
- **lines** : la liste contenant les coordonnées géographiques des arcs.

Un objet **SpatialPolygons(DataFrame)** possède les attributs spécifiques suivants :
- **polygons** : la liste contenant les coordonnées géographiques des polygones ;
- **plotOrder** : l'ordre dans lequel devraient être tracés les polygones pour permettre une bonne lisibilité de la carte.

Comme point de départ des exemples qui vont suivre, on dispose d'un **data.frame**, contenant notamment des coordonnées géographiques, défini comme suit :

```
Pt_tab <- data.frame(
  X     = c( 281138,   78629,  559159,  971042,  957312,  665562,  281138, 1173551, 1122066, 1176984,
             1201010, 1173551,   54603,  239950,  123250,  198762,  524835,  54603),
  Y     = c(1823492, 2413857, 2657554, 2472207, 1809762, 1703359, 1823492, 1620983, 1699927, 1796033,
             1706791, 1620983, 2595772, 3007654, 3124354, 3536237, 2691878, 2595772),
  IDE   = rep(c("A", "B", "C")     , times = c(7, 5, 6)),
  CNTRY = rep(c("FR", "GB")        , times = c(12, 6)),
  CAP   = rep(c("Paris", "London"), times = c(12, 6))
  )
```

```
summary(Pt_tab)
##        X                 Y           IDE   CNTRY       CAP
##  Min.   :  54603   Min.   :1620983   A:7   FR:12   London: 6
##  1st Qu.: 209059   1st Qu.:1729102   B:5   GB: 6   Paris :12
##  Median : 541997   Median :2118674   C:6
##  Mean   : 602064   Mean   :2261117
##  3rd Qu.:1084310   3rd Qu.:2642108
##  Max.   :1201010   Max.   :3536237
```

- *X* : la longitude (exprimée en Lambert 2 étendu) ;
- *Y* : la latitude (exprimée en Lambert 2 étendu) ;
- *IDE* : un identifiant d'entité ;
- *CNTRY* : le code ISO 3166-1 alpha-2 du pays [1] ;
- *CAP* : le nom de la capitale du pays.

1.1.1 Classe SpatialPoints

Comme on vient de le voir, dans R, les données spatiales correspondant à des points sont de classe **SpatialPoints**. Pour créer un objet de classe **SpatialPointsDataFrame**, il existe plusieurs solutions.

Méthode 1. On transforme le **data.frame** des coordonnées en objet de classe **SpatialPoints** à l'aide de la fonction **SpatialPoints()**. Pour cela, on fournit la matrice (ou le **data.frame**) des longitudes (1re colonne) et latitudes (2e colonne) des points à l'argument **coords**. La définition du système de coordonnées, *via* l'argument **proj4string**, est facultative.

```
SPt <- SpatialPoints(coords = Pt_tab[, c("X", "Y")],
                     proj4string = CRS("+init=epsg:27572"))
class(SPt)
## [1] "SpatialPoints"
## attr(,"package")
## [1] "sp"
```

L'objet ainsi créé est déjà un objet spatial (**sp**) en tant que tel. On a la possibilité d'adjoindre des données attributaires à chaque entité de cet objet. La très grande majorité du temps, c'est ce format de données final que l'on souhaite obtenir. Afin d'attacher des données attributaires aux données spatiales, on combine cet objet **SpatialPoints** avec des données contenues dans un **data.frame**, pour former un objet de classe **SpatialPointsDataFrame**. Il faut fournir une table attributaire comportant autant de lignes que de couples de coordonnées de points existants.

```
SPtD <- SpatialPointsDataFrame(coords = SPt,
                               data = Pt_tab[, c("IDE", "CNTRY", "CAP")])
class(SPtD)
## [1] "SpatialPointsDataFrame"
## attr(,"package")
## [1] "sp"
```

Méthode 2. On peut atteindre le même résultat en une seule étape, en utilisant directement la fonction **SpatialPointsDataFrame()**, sans passer par l'intermédiaire de **SpatialPoints()**. Au lieu de passer un objet **SpatialPoints** à l'argument **coords** de la fonction, on lui passe le tableau des coordonnées. Dans l'argument **data**, on définit la table attributaire et, dans l'argument **proj4string**, on peut définir le système de coordonnées.

1. Site web de International Organization for Standardization : http://www.iso.org/iso/fr/country_codes.htm.

```
SPtD <- SpatialPointsDataFrame(coords = Pt_tab[, c("X", "Y")],
                               data = Pt_tab[, c("IDE", "CNTRY", "CAP")],
                               proj4string = CRS("+init=epsg:27572"))
class(SPtD)
## [1] "SpatialPointsDataFrame"
## attr(,"package")
## [1] "sp"
```

Méthode 3. On peut également transformer directement le **`data.frame`** en objet **`SpatialPointsDataFrame`** en explicitant quelles colonnes du tableau contiennent les coordonnées géographiques. Ces colonnes peuvent porter n'importe quels noms, et pas forcément X et Y, comme dans notre exemple.

```
SPtD <- Pt_tab
coordinates(SPtD) <- ~ X + Y
class(SPtD)
## [1] "SpatialPointsDataFrame"
## attr(,"package")
## [1] "sp"
```

Notez que si le **`data.frame`** contient uniquement les colonnes des coordonnées, sans donnée attributaire, il ne sera alors logiquement transformé qu'en objet de classe **`SpatialPoints`**.

```
SPtD <- Pt_tab[, c("X", "Y")]
coordinates(SPtD) <- ~ X + Y
class(SPtD)
## [1] "SpatialPoints"
## attr(,"package")
## [1] "sp"
```

Avec cette manière de faire, le système de coordonnées se définit *a posteriori*, au moyen de la fonction **`proj4string()`**.

```
proj4string(SPtD)
## [1] NA
proj4string(SPtD) <- CRS("+init=epsg:27572")
proj4string(SPtD)
## [1] "+init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_0=2200000 +a=637
```

Dans tous les cas, le résultat final est le même.

```
str(SPt)
## Formal class 'SpatialPoints' [package "sp"] with 3 slots
##   ..@ coords     : num [1:18, 1:2] 281138 78629 559159 971042 957312 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "X" "Y"
##   ..@ bbox       : num [1:2, 1:2] 54603 1620983 1201010 3536237
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "X" "Y"
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr "+init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=6000
```

Comme dit précédemment, les objets **`sp`** sont de mode **`S4`**. Pour lister les éléments de ce type d'objets, on utilise la fonction **`slotNames()`**. Pour un objet **`SpatialPoints`**, les composantes sont les suivantes :

```
mode(SPtD)
## [1] "S4"
slotNames(SPtD)
## [1] "data"        "coords.nrs"  "coords"      "bbox"        "proj4string"
```

La composante **`@data`** contient les données attributaires.

```
str(SPtD@data)
## 'data.frame': 18 obs. of  3 variables:
##  $ IDE  : Factor w/ 3 levels "A","B","C": 1 1 1 1 1 1 1 2 2 2 ...
##  $ CNTRY: Factor w/ 2 levels "FR","GB": 1 1 1 1 1 1 1 1 1 1 ...
##  $ CAP  : Factor w/ 2 levels "London","Paris": 2 2 2 2 2 2 2 2 2 2 ...
```

La composante **`@coords.nrs`** contient la position des colonnes de coordonnées dans le tableau initial ayant servi à la création de l'objet **`sp`**.

```
str(SPtD@coords.nrs)
##  int [1:2] 1 2
```

La composante **`@coords`** contient les coordonnées des points.

```
str(SPtD@coords)
##  num [1:18, 1:2] 281138 78629 559159 971042 957312 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:18] "1" "2" "3" "4" ...
##   ..$ : chr [1:2] "X" "Y"
```

La composante **@bbox** contient l'emprise géographique de l'objet.

```
str(SPtD@bbox)
##  num [1:2, 1:2] 54603 1620983 1201010 3536237
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:2] "X" "Y"
##   ..$ : chr [1:2] "min" "max"
```

La composante **@proj4string** contient les paramètres du système de coordonnées.

```
str(SPtD@proj4string)
## Formal class 'CRS' [package "sp"] with 1 slot
##   ..@ projargs: chr "+init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_
```

On peut tracer l'objet **SpatialPoints(DataFrame)** avec un simple appel à la fonction **plot()**. Par défaut, les points sont représentés sous forme de croix. Le ratio entre les axes des abscisses et des ordonnées est, quant à lui, respecté, ce qui n'est pas le cas dans l'utilisation conventionnelle de la fonction **plot()**.

```
plot(SPtD, col = "cyan4")
```

1.1.2 Classe SpatialMultiPoints

Il existe une autre classe d'objets vectoriels contenant des points qui est un peu plus complexe que la classe **SpatialPoints** ; il s'agit de la classe **SpatialMultiPoints**. Dans ce format, une ligne de la table attributaire peut être non pas raccordée à un seul et unique point (comme c'est le cas pour les objets **SpatialPoints**), mais à un groupe de plusieurs points.

La construction d'objets spatiaux **SpatialMultiPoints** est un petit peu plus complexe que ce que l'on vient de voir avec les objets de classe **SpatialPoints**.

Étape 1 – classes matrix ou data.frame. On commence par créer autant d'objets de classes **matrix** ou **data.frame** que l'on souhaite de couches de points. Chaque objet **matrix** ou **data.frame** correspond à un tableau contenant les coordonnées des points à grouper.

Le **data.frame** de départ contenait un identifiant (colonne *IDE*), ce qui permet de déterminer quelles sont les lignes à regrouper entre elles. Dans notre exemple, on souhaite créer 2 groupes de points, nommés *FR* et *GB*. Les différents groupes de points associés correspondront à une seule et même ligne dans la table attributaire.

```
pt1 <- Pt_tab[Pt_tab$CNTRY == "FR" , c("X", "Y")]
pt2 <- Pt_tab[Pt_tab$CNTRY == "GB" , c("X", "Y")]
class(pt1)
## [1] "data.frame"
str(pt1)
## 'data.frame': 12 obs. of  2 variables:
##  $ X: num  281138 78629 559159 971042 957312 ...
##  $ Y: num  1823492 2413857 2657554 2472207 1809762 ...
```

Étape 2 – classe SpatialMultiPoints. Ensuite, on liste l'ensemble des tableau de coordonnées de chaque groupe de points dans un objet **SpatialMultiPoints**. À cette étape, on peut définir le système de coordonnées.

```
SMp <- SpatialMultiPoints(coords = list(pt1, pt2), proj4string = CRS("+init=epsg:27572"))
class(SMp)
```

```
## [1] "SpatialMultiPoints"
## attr(,"package")
## [1] "sp"
```

```
str(SMp, max.level = 2)
```

```
## Formal class 'SpatialMultiPoints' [package "sp"] with 3 slots
##   ..@ coords     :List of 2
##   ..@ bbox       : num [1:2, 1:2] 54603 1620983 1201010 3536237
##   .. ..- attr(*, "dimnames")=List of 2
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
```

Étape 3 – classe `SpatialMultiPointsDataFrame`. Pour pouvoir créer un objet de classe **`SpatialMultiPointsDataFrame`**, il est nécessaire de disposer d'une table attributaire. Cette table doit comporter autant de lignes que de groupes de points ayant été définis. Afin de créer une jointure fiable, il est impératif que les noms de lignes de cette table correspondent à l'identifiant défini pour chaque objet **`Lines`**.

```
Mp_tab <- unique(Pt_tab[, c("CNTRY", "CAP"), drop = FALSE])
rownames(Mp_tab) <- Mp_tab$CNTRY
Mp_tab
```

```
##    CNTRY    CAP
## FR    FR  Paris
## GB    GB London
```

On associe l'objet **`SpatialMultiPoints`** avec la table attributaire pour former l'objet final **`SpatialMultiPointsDataFrame`**. On utilise l'argument **`match.ID = TRUE`** pour forcer la jointure entre les entités et les lignes de la table attributaire par l'identifiant qui a été défini.

```
SMpD <- SpatialMultiPointsDataFrame(coords = SMp, data = Mp_tab, match.ID = TRUE)
class(SMpD)
```

```
## [1] "SpatialMultiPointsDataFrame"
## attr(,"package")
## [1] "sp"
```

```
str(SMpD, max.level = 2)
```

```
## Formal class 'SpatialMultiPointsDataFrame' [package "sp"] with 4 slots
##   ..@ data       :'data.frame': 2 obs. of  2 variables:
##   ..@ coords     :List of 2
##   ..@ bbox       : num [1:2, 1:2] 54603 1620983 1201010 3536237
##   .. ..- attr(*, "dimnames")=List of 2
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
```

On liste les différentes composantes de l'objet **`SpatialMultiPointsDataFrame`** avec la fonction **`slotNames()`**.

```
mode(SMpD)
```

```
## [1] "S4"
```

```
slotNames(SMpD)
```

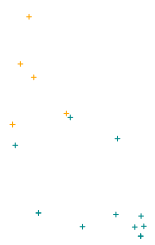
```
## [1] "data"        "coords"      "bbox"        "proj4string"
```

Un objet **`SpatialMultiPointsDataFrame`** est une liste contenant 4 éléments :

- **`data`** : les données attributaires ;
- **`coords`** : la liste des coordonnées des couches de points ;
- **`bbox`** : l'emprise géographique ;
- **`proj4string`** : les paramètres du système de coordonnées.

On trace l'objet **`SpatialMultiPoints(DataFrame)`** avec la fonction **`plot()`**. Notez que l'on peut définir une couleur pour chacun des groupes de points.

```
plot(SMpD, col = c("cyan4", "orange"))
```



1.1.3 Classe SpatialLines

La construction d'objets spatiaux composés de polylignes est un peu plus complexe, et nécessite davantage d'étapes.

Étape 1 – classe `line`. On commence par créer autant d'objets de classe **`line`** que l'on souhaite de polylignes. Chaque objet **`line`** est composé par un tableau contenant les coordonnées des points à relier.

Le **`data.frame`** de départ contenait un identifiant (colonne *IDE*), ce qui permet de déterminer quelles sont les lignes à regrouper entre elles. Dans notre exemple, on souhaite créer 3 polylignes, nommées A, B et C.

```
liA <- Line(coords = Pt_tab[Pt_tab$IDE == "A" , c("X", "Y")])
liB <- Line(coords = Pt_tab[Pt_tab$IDE == "B" , c("X", "Y")])
liC <- Line(coords = Pt_tab[Pt_tab$IDE == "C" , c("X", "Y")])
class(liA)
## [1] "Line"
## attr(,"package")
## [1] "sp"
str(liA)
## Formal class 'Line' [package "sp"] with 1 slot
##   ..@ coords: num [1:7, 1:2] 281138 78629 559159 971042 957312 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:7] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "X" "Y"
```

Étape 2 – classe `Lines`. L'étape suivante consiste en un éventuel regroupement de certaines polylignes. Pour cela, on crée des objets **`Lines`** à partir d'une liste d'un (si pas de regroupement) ou plusieurs (si regroupement) objets **`line`**, et l'on fixe un identifiant pour chacun des objets **`Lines`**. Si regroupement il y a, les différentes polylignes associées correspondront à une seule et même ligne dans la table attributaire.

Dans notre exemple, on souhaite regrouper deux polylignes et en laisser une toute seule. On crée donc 2 objets **`Lines`** à partir des 3 objets **`line`**.

```
LiFR <- Lines(slinelist = list(liA, liB), ID = "FR")
LiGB <- Lines(slinelist = list(liC)     , ID = "GB")
class(LiFR)
## [1] "Lines"
## attr(,"package")
## [1] "sp"
str(LiFR)
## Formal class 'Lines' [package "sp"] with 2 slots
##   ..@ Lines:List of 2
##   .. ..$ :Formal class 'Line' [package "sp"] with 1 slot
##   .. .. .. ..@ coords: num [1:7, 1:2] 281138 78629 559159 971042 957312 ...
##   .. .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. .. ..$ : chr [1:7] "1" "2" "3" "4" ...
##   .. .. .. .. .. ..$ : chr [1:2] "X" "Y"
##   .. ..$ :Formal class 'Line' [package "sp"] with 1 slot
##   .. .. .. ..@ coords: num [1:5, 1:2] 1173551 1122066 1176984 1201010 1173551 ...
##   .. .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. .. ..$ : chr [1:5] "8" "9" "10" "11" ...
##   .. .. .. .. .. ..$ : chr [1:2] "X" "Y"
##   ..@ ID   : chr "FR"
```

Étape 3 – classe `SpatialLines`. Ensuite, on liste l'ensemble des objets **`Lines`** dans un objet **`SpatialLines`**. À cette étape, on peut définir le système de coordonnées.

```
SLi <- SpatialLines(LinesList = list(LiFR, LiGB), proj4string = CRS("+init=epsg:27572"))
class(SLi)
## [1] "SpatialLines"
## attr(,"package")
## [1] "sp"
str(SLi, max.level = 2)
## Formal class 'SpatialLines' [package "sp"] with 3 slots
##   ..@ lines      :List of 2
##   ..@ bbox       : num [1:2, 1:2] 54603 1620983 1201010 3536237
##   .. ..- attr(*, "dimnames")=List of 2
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
```

Étape 4 – classe `SpatialLinesDataFrame`. Pour pouvoir créer un objet de classe **`SpatialLinesDataFrame`**, il est nécessaire de disposer d'une table attributaire. Cette table doit comporter autant de lignes que d'objets **`Lines`** ayant été définis. Afin de créer une jointure fiable, il est impératif que les noms de lignes de cette table correspondent à l'identifiant défini pour chaque objet **`Lines`**.

```
Li_tab <- unique(Pt_tab[, c("CNTRY", "CAP"), drop = FALSE])
rownames(Li_tab) <- Li_tab$CNTRY
Li_tab
```

```
##    CNTRY    CAP
## FR    FR  Paris
## GB    GB London
```

On associe l'objet **SpatialLines** avec la table attributaire pour former l'objet final **SpatialLinesDataFrame**. On utilise l'argument **match.ID = TRUE** pour forcer la jointure entre les entités et les lignes de la table attributaire par l'identifiant qui a été défini.

```
SLiD <- SpatialLinesDataFrame(sl = SLi, data = Li_tab, match.ID = TRUE)
class(SLiD)
## [1] "SpatialLinesDataFrame"
## attr(,"package")
## [1] "sp"
str(SLiD, max.level = 2)
## Formal class 'SpatialLinesDataFrame' [package "sp"] with 4 slots
##   ..@ data       :'data.frame': 2 obs. of  2 variables:
##   ..@ lines      :List of 2
##   ..@ bbox       : num [1:2, 1:2] 54603 1620983 1201010 3536237
##   .. ..- attr(*, "dimnames")=List of 2
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
```

On liste les différentes composantes de l'objet **SpatialLinesDataFrame** avec la fonction **slotNames()**.

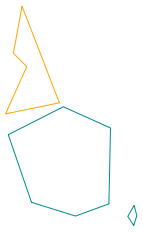
```
mode(SLiD)
## [1] "S4"
slotNames(SLiD)
## [1] "data"        "lines"       "bbox"        "proj4string"
```

Un objet **SpatialLinesDataFrame** est une liste contenant 4 éléments :

- **data** : les données attributaires ;
- **lines** : la liste des coordonnées des polylignes ;
- **bbox** : l'emprise géographique ;
- **proj4string** : les paramètres du système de coordonnées.

On trace l'objet **SpatialLines(DataFrame)** avec la fonction **plot()**. On peut attribuer une couleur à chaque polyligne.

```
plot(SLiD, col = c("cyan4", "orange"))
```



1.1.4 Classe SpatialPolygons

La construction d'objets spatiaux composés de polygones est très similaire à celle des polylignes.

Étape 1 – classe Polygon. On crée autant d'objets de classe **Polygon** que l'on souhaite de polygones. Chaque objet **Polygon** est constitué par une table contenant les coordonnées des points à relier. Attention, pour chacun des polygones, il est indispensable que les coordonnées des points de départ et d'arrivée soient identiques, afin que les contours soient bien fermés. Le **data.frame** de départ contenait une colonne (*IDE*), qui identifie les polygones que l'on souhaite regrouper.

Dans notre exemple, on souhaite créer 3 polygones, nommées : A, B et C. Par ailleurs, on souhaite que le polygone B comporte un trou en son milieu. Le polygone définissant le trou est reconnu automatiquement comme tel, étant donné le fait qu'il est complètement inclus dans un autre polygone. C'est aussi le cas si le sens de rotation des coordonnées de ponts formant le polygones suit le sens inverse de celui des aiguilles d'une montre.

```
poA  <- Polygon(coords = Pt_tab[Pt_tab$IDE == "A" , c("X", "Y")])
poB  <- Polygon(coords = Pt_tab[Pt_tab$IDE == "B" , c("X", "Y")])
poC  <- Polygon(coords = Pt_tab[Pt_tab$IDE == "C" , c("X", "Y")])
poBh <- Polygon(coords = data.frame(X = c( 429327,  733304,  733304,  429327,  429327),
```

```
                                    Y = c(2054697, 2054697, 2305031, 2305031, 2054697)))
class(poA)
## [1] "Polygon"
## attr(,"package")
## [1] "sp"
str(poA)
## Formal class 'Polygon' [package "sp"] with 5 slots
##   ..@ labpt  : num [1:2] 574031 2178758
##   ..@ area   : num 6.26e+11
##   ..@ hole   : logi FALSE
##   ..@ ringDir: int 1
##   ..@ coords : num [1:7, 1:2] 281138 78629 559159 971042 957312 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:7] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "X" "Y"
```

Un objet **Polygon** est une liste contenant 5 éléments :

- **labpt** : les coordonnées du centre de gravité ;
- **area** : l'aire du polygone ;
- **hole** : le polygone est ou non un trou ;
- **ringDir** : le sens de rotation des coordonnées ;
- **coords** : les coordonnées.

Étape 2 – classe Polygons. Puis, si on le souhaite, on peut regrouper certains polygones. Pour cela, on crée des objets **Polygons** à partir d'une liste d'un ou plusieurs objets **Polygon**, et l'on définit un identifiant pour chacun des objets **Polygons**. Chaque groupe correspond à une seule ligne dans la table attributaire.

Dans notre exemple, on souhaite regrouper deux polygones (dont un troué) et en laisser un tout seul. On créer donc 2 objets **Polygons** à partir des 3 **Polygon** initiaux et d'un **Polygon** définissant un trou.

```
PoFR <- Polygons(srl = list(poA, poB, poBh), ID = "FR")
PoGB <- Polygons(srl = list(poC)            , ID = "GB")
class(PoFR)
## [1] "Polygons"
## attr(,"package")
## [1] "sp"
str(PoFR)
## Formal class 'Polygons' [package "sp"] with 5 slots
##   ..@ Polygons :List of 3
##   .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
##   .. .. .. .. ..@ labpt  : num [1:2] 574031 2178758
##   .. .. .. .. ..@ area   : num 6.26e+11
##   .. .. .. .. ..@ hole   : logi FALSE
##   .. .. .. .. ..@ ringDir: int 1
##   .. .. .. .. ..@ coords : num [1:7, 1:2] 281138 78629 559159 971042 957312 ...
##   .. .. .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. .. .. ..$ : chr [1:7] "1" "2" "3" "4" ...
##   .. .. .. .. .. .. ..$ : chr [1:2] "X" "Y"
##   .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
##   .. .. .. .. ..@ labpt  : num [1:2] 1166140 1706396
##   .. .. .. .. ..@ area   : num 6.9e+09
##   .. .. .. .. ..@ hole   : logi FALSE
##   .. .. .. .. ..@ ringDir: int 1
##   .. .. .. .. ..@ coords : num [1:5, 1:2] 1173551 1122066 1176984 1201010 1173551 ...
##   .. .. .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. .. .. ..$ : chr [1:5] "8" "9" "10" "11" ...
##   .. .. .. .. .. .. ..$ : chr [1:2] "X" "Y"
##   .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
##   .. .. .. .. ..@ labpt  : num [1:2] 581316 2179864
##   .. .. .. .. ..@ area   : num 7.61e+10
##   .. .. .. .. ..@ hole   : logi TRUE
##   .. .. .. .. ..@ ringDir: int -1
##   .. .. .. .. ..@ coords : num [1:5, 1:2] 429327 733304 733304 429327 429327 ...
##   .. .. .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. .. .. ..$ : NULL
##   .. .. .. .. .. .. ..$ : chr [1:2] "X" "Y"
##   ..@ plotOrder: int [1:3] 1 3 2
##   ..@ labpt    : num [1:2] 574031 2178758
##   ..@ ID       : chr "FR"
##   ..@ area     : num 6.33e+11
```

Un objet **Polygons** est une liste contenant 5 éléments :

- **Polygons** : un objet **Polygon** ;
- **plotOrder** : l'ordre dans lequel devraient être tracés les polygones ;
- **labpt** : les coordonnées du centre de gravité du groupe de polygones ;
- **ID** : l'identifiant du groupe de polygones ;

– **area** : la somme des aires du groupe de polygones (l'aire du trou n'est pas décomptée).

Attention, la surface calculée est erronée pour les groupes de polygones contenant un ou plusieurs trous. Ces derniers ne sont pas pris en compte, alors que leurs superficies devraient être soustraites. Néanmoins, il existe des fonctions qui permettent de remédier à ce problème et de calculer correctement les aires (§ 4.2.7).

Étape 3 – classe SpatialPolygons. Ensuite, on liste l'ensemble des objets **Polygons** dans un objet **SpatialPolygons**. À cette étape, on a la possibilité de définir le système de coordonnées.

```
SPo <- SpatialPolygons(Srl = list(PoFR, PoGB), proj4string = CRS("+init=epsg:27572"))
class(SPo)
## [1] "SpatialPolygons"
## attr(,"package")
## [1] "sp"
str(SPo, max.level = 2)
## Formal class 'SpatialPolygons' [package "sp"] with 4 slots
##   ..@ polygons   :List of 2
##   ..@ plotOrder  : int [1:2] 1 2
##   ..@ bbox       : num [1:2, 1:2] 54603 1620983 1201010 3536237
##   .. ..- attr(*, "dimnames")=List of 2
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
```

Un objet **SpatialPolygons** est une liste contenant 4 éléments :

– **polygons** : un objet **Polygons** ;
– **plotOrder** : l'ordre dans lequel devraient être tracés les groupes de polygones ;
– **bbox** : l'emprise géographique ;
– **proj4string** : le système de coordonnées.

Étape 4 – classe SpatialPolygonsDataFrame. À présent, on construit la table attributaire. Comme c'était le cas pour les lignes, la table doit contenir autant de lignes que d'objets **Polygons** définis. Une nouvelle fois, on s'assure que les noms de lignes de cette table correspondent à l'identifiant défini pour chaque objet **Polygons**.

```
Po_tab <- unique(Pt_tab[, c("CNTRY", "CAP"), drop = FALSE])
rownames(Po_tab) <- Po_tab$CNTRY
Po_tab
##    CNTRY    CAP
## FR    FR  Paris
## GB    GB London
```

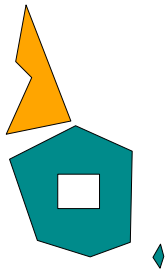
On associe l'objet **SpatialPolygons** avec la table attributaire pour former l'objet final **SpatialPolygonsDataFrame**. On utilise l'argument **match.ID = TRUE** pour forcer la jointure entre les entités et les lignes de la table attributaire par l'identifiant que l'on a défini.

```
SPoD <- SpatialPolygonsDataFrame(Sr = SPo, data = Po_tab, match.ID = TRUE)
class(SPoD)
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
str(SPoD, max.level = 2)
## Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
##   ..@ data       :'data.frame': 2 obs. of  2 variables:
##   ..@ polygons   :List of 2
##   ..@ plotOrder  : int [1:2] 1 2
##   ..@ bbox       : num [1:2, 1:2] 54603 1620983 1201010 3536237
##   .. ..- attr(*, "dimnames")=List of 2
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
```

Un objet **SpatialPolygonsDataFrame** est une liste contenant 5 éléments, les 4 éléments de l'objet **SpatialPolygons** et le nouvel élément adjoint **@data** qui contient la table attributaire.

On trace l'objet **SpatialPolygons(DataFrame)** avec la fonction **plot()**. On peut attribuer une couleur à chacune des entités de l'objet.

```
plot(SPoD, col = c("cyan4", "orange"))
```



1.1.5 Indexer des objets vectoriels de classe sp

On peut connaître le nombre d'entités contenues dans un objet **sp**, grâce à la fonction **length()**.

```
length(SPoD)
## [1] 2
```

La fonction **coordinates()** s'applique aux différentes classes d'objets **sp** et permet de récupérer différentes sorties :

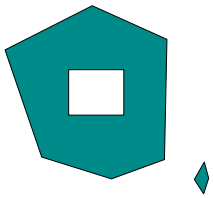
- **SpatialPoints(DataFrame)** : la matrice des coordonnées des points ;
- **SpatialMultiPoints(DataFrame)** : *idem* ;
- **SpatialLines(DataFrame)** : la liste des coordonnées des arcs ;
- **SpatialPolygons(DataFrame)** : les centroïdes des polygones.

```
head(coordinates(SPtD))
##        X       Y
## 1 281138 1823492
## 2  78629 2413857
## 3 559159 2657554
## 4 971042 2472207
## 5 957312 1809762
## 6 665562 1703359
head(coordinates(SMpD))
##     [,1]    [,2]
## 1 281138 1823492
## 1  78629 2413857
## 1 559159 2657554
## 1 971042 2472207
## 1 957312 1809762
## 1 665562 1703359
coordinates(SLiD)
## [[1]]
## [[1]][[1]]
##        X       Y
## 1 281138 1823492
## 2  78629 2413857
## 3 559159 2657554
## 4 971042 2472207
## 5 957312 1809762
## 6 665562 1703359
## 7 281138 1823492
##
## [[1]][[2]]
##          X       Y
## 8  1173551 1620983
## 9  1122066 1699927
## 10 1176984 1796033
## 11 1201010 1706791
## 12 1173551 1620983
##
##
## [[2]]
## [[2]][[1]]
##        X       Y
## 13  54603 2595772
## 14 239950 3007654
## 15 123250 3124354
## 16 198762 3536237
## 17 524835 2691878
## 18  54603 2595772
coordinates(SPoD)
##        [,1]    [,2]
## FR 574031.3 2178758
## GB 277801.9 2951856
```

On peut indexer numériquement un objet **sp**, en utilisant la syntaxe "**[,]**", comme pour les objets de classes **matrix** ou **data.frame**.

Dans l'exemple suivant, on ne conserve que le premier groupe de polygones de la liste, c'est-à-dire le premier **Polygons** du **SpatialPolygons(DataFrame)**.

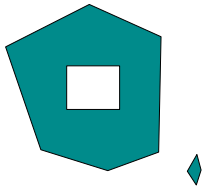
```
plot(SPoD[1, ], col = "cyan4")
```



On peut indexer compte tenu des informations contenues dans la table attributaire. On se sert donc du résultat de l'indexation de la table attributaire pour indexer indirectement l'objet **sp**.

On ne considère ici que les polygones de la France en sélectionnant les lignes de la table attributaire pour lesquelles les valeurs contenues dans la colonne *CNTRY* valent *FR*.

```
plot(SPoD[SPoD@data$CNTRY == "FR", ], col = "cyan4")
```

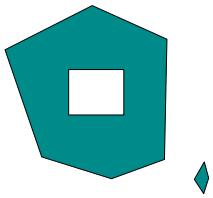


Notez que, pour réaliser le test logique, nous récupérons la colonne d'intérêt de la table attributaire avec une syntaxe faisant apparaître l'élément **@data**. Ce n'est pas obligatoire ; on peut, en effet, se passer de son écriture, comme le montre les lignes de code ci-dessous. Cependant, même si elle est plus verbeuse, cette syntaxe sera préférée dans l'ensemble de l'ouvrage, car plus respectueuse de la structure même des données de classe **sp**.

```
SPoD@data$CNTRY
## [1] FR GB
## Levels: FR GB
SPoD$CNTRY
## [1] FR GB
## Levels: FR GB
```

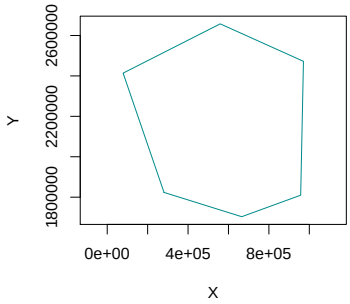
On peut aussi indexer l'objet **sp** sur n'importe quel élément qui informe sur les caractéristiques d'une entité géographique, par exemple l'aire du polygone. Ici, on ne retient que le groupe de polygones qui présente la superficie la plus grande.

```
(SPoD_area <- sapply(SPoD@polygons, function(x) slot(x, "area")))
## [1] 632985048648 185162660204
plot(SPoD[which.max(SPoD_area), ], col = "cyan4")
```



Pour avoir accès à une sous-entité, il faut descendre d'un niveau dans l'élément **@polygons**, au niveau des éléments **@Polygons**. Ici, on ne retient que le polygone qui présente la superficie la plus grande au sein de son groupe. Comme l'objet représenté n'est pas de classe **sp**, pour que le ratio soit respecté entre les deux axes, il faut utiliser l'argument **asp = 1**, sans quoi la représentation sera déformée.

```
(SPoD_area2 <- sapply(SPoD@polygons[[1]]@Polygons, function(x) slot(x, "area")))
## [1] 626087257104   6897791544  76095778318
str(SPoD@polygons[[1]]@Polygons[which.max(SPoD_area2)])
## List of 1
##  $ :Formal class 'Polygon' [package "sp"] with 5 slots
##   .. ..@ labpt  : num [1:2] 574031 2178758
##   .. ..@ area   : num 6.26e+11
##   .. ..@ hole   : logi FALSE
##   .. ..@ ringDir: int 1
##   .. ..@ coords : num [1:7, 1:2] 281138 78629 559159 971042 957312 ...
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:7] "1" "2" "3" "4" ...
##   .. .. .. ..$ : chr [1:2] "X" "Y"
plot(SPoD@polygons[[1]]@Polygons[[which.max(SPoD_area2)]]@coords, type = "l", col = "cyan4", asp = 1)
```



1.2 Données matricielles

Dans R, il existe deux formats principaux de données matricielles : le format **sp** et le format **raster**. Ces deux formats permettent aussi bien de gérer des rasters simples que des rasters à plusieurs bandes.

1.2.1 Classe sp

Comme on l'a vu, c'est le package **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a) qui permet de créer et manipuler les objets spatiaux de classe **sp**. Les objets matriciels, gérés par ce package, présentent le même type de structure que les objets vectoriels que l'on vient de présenter (§ 1.1).

```
library(sp)
```

Selon le type d'entité géographique considéré, une classe secondaire est définie :

- **SpatialPixels(DataFrame)** pour des pixels, formant une grille rectangulaire possiblement incomplète ;
- **GridTopology(DataFrame)** pour une grille rectangulaire vide ;
- **SpatialGrid(DataFrame)** pour une grille rectangulaire pleine.

Il existe plusieurs manières d'aboutir à un objet matriciel.

On peut partir de coordonnées de points espacés régulièrement, et qui présentent au moins une colonne d'attributs. Dans notre cas, on dispose d'un **data.frame** contenant des coordonnées X et Y, ainsi que deux colonnes d'attributs.

```
head(Pt_reg)
##           X       Y IDE CNTRY
## 1 1181755.9 1698671   3     2
## 2  329147.1 1820472   2     2
## 3  450948.4 1820472   2     2
## 4  572749.6 1820472   2     2
## 5  694550.9 1820472   2     2
## 6  816352.1 1820472   2     2
```

Pour le transformer en objet matriciel, on le convertit tout d'abord en objet vectoriel de classe **SpatialPointsDataFrame**. Pour cela, on applique une des méthodes décrites précédemment.

```
SPtD_reg <- Pt_reg
coordinates(SPtD_reg) <- ~ X + Y
proj4string(SPtD_reg) <- CRS("+init=epsg:27572")
```

1.2.1.1 Classe SpatialPixelsDataFrame

On crée un objet **SpatialPixelsDataFrame**, c'est-à-dire un objet présentant une structure maillée potentiellement incomplète. En cela, ces objets sont un peu intermédiaires entre des objets vectoriels et des objets matriciels.

Méthode 1. On crée un objet **SpatialPixelsDataFrame** en passant par l'intermédiaire d'un **SpatialPixels**. L'objet passé en argument **points** de la fonction **SpatialPixels()** doit être de classe **SpatialPointsDataFrame**.

```
SPi  <- SpatialPixels(points = SPtD_reg)
SPiD <- SpatialPixelsDataFrame(points = SPi, data = Pt_reg[, c("IDE", "CNTRY")])
```

Méthode 2. On crée un objet **SpatialPixelsDataFrame** sans passer par l'intermédiaire d'un **SpatialPixels**. L'objet passé en argument **points** de la fonction **SpatialPixelsDataframe()** peut être de classe **matrix** ou **SpatialPointsDataFrame**.

```
SPiD <- SpatialPixelsDataFrame(points = Pt_reg[, c("X", "Y")], data = Pt_reg[, c("IDE", "CNTRY")])
SPiD <- SpatialPixelsDataFrame(points = SPtD_reg              , data = Pt_reg[, c("IDE", "CNTRY")])
```

Dans les deux cas, le résultat obtenu est le suivant :

```
str(SPiD)
## Formal class 'SpatialPixelsDataFrame' [package "sp"] with 7 slots
##   ..@ data       :'data.frame': 50 obs. of  2 variables:
##   .. ..$ IDE  : num [1:50] 3 2 2 2 2 2 2 2 2 2 ...
##   .. ..$ CNTRY: num [1:50] 2 2 2 2 2 2 2 2 2 2 ...
##   ..@ coords.nrs : num(0)
##   ..@ grid       :Formal class 'GridTopology' [package "sp"] with 3 slots
##   .. .. ..@ cellcentre.offset: Named num [1:2] 207346 1698671
##   .. .. .. ..- attr(*, "names")= chr [1:2] "X" "Y"
##   .. .. ..@ cellsize         : Named num [1:2] 121801 121801
##   .. .. .. ..- attr(*, "names")= chr [1:2] "X" "Y"
##   .. .. ..@ cells.dim        : Named int [1:2] 9 15
##   .. .. .. ..- attr(*, "names")= chr [1:2] "X" "Y"
##   ..@ grid.index : int [1:50] 135 119 120 121 122 123 124 110 111 112 ...
##   ..@ coords     : num [1:50, 1:2] 1181756 329147 450948 572750 694551 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:50] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "X" "Y"
##   ..@ bbox       : num [1:2, 1:2] 146445 1637771 1242657 3464789
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "X" "Y"
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr "+init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=6000
```

Comme dit plus haut, la structure de l'objet est très semblable à celles des autres objets de classe **sp**.

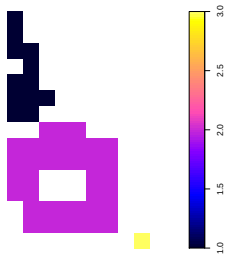
```
slotNames(SPiD)
## [1] "data"        "coords.nrs"  "grid"        "grid.index"  "coords"      "bbox"
## [7] "proj4string"
```

Un objet **SpatialPixelsDataFrame** est constitué des 7 éléments suivants :

- **data** : la table attributaire ;
- **coords.nrs** : la position des colonnes de coordonnées dans le tableau initial ;
- **grid** : la topologie de la grille ;
- **grid.index** : la position des pixels sur la grille (1 : coin supérieur gauche) ;
- **coords** : les coordonnées de pixels ;
- **bbox** : l'emprise géographique ;
- **proj4string** : le système de coordonnées.

On trace l'objet **SpatialPixelsDataFrame** à l'aide de la fonction **plot()**. On voit bien qu'il s'agit de points disposés sur une grille régulière.

```
plot(SPiD)
```



1.2.1.2 Classe SpatialGridDataFrame

Méthode 1. On transforme tout d'abord l'objet **SpatialPointsDataFrame** en **SpatialPixelsDataFrame**. On peut réaliser cette opération comme précédemment (§ 1.2.1.1), ou bien en forçant l'objet à être défini comme tel, en utilisant la fonction **gridded()**. Puis, on peut le convertir en un objet au format matriciel de la classe **SpatialGridDataFrame**. L'objet de classe **SpatialGridDataFrame** étant défini sur une grille complète, il est davantage conforme à l'idée que l'on se fait *a priori* de données matricielles que l'objet de classe **SpatialPixelsDataFrame**.

```
SGrD <- SPtD_reg
gridded(SGrD) <- TRUE
class(SGrD)
## [1] "SpatialPixelsDataFrame"
## attr(,"package")
## [1] "sp"
SGrD <- as(SGrD, "SpatialGridDataFrame")
class(SGrD)
## [1] "SpatialGridDataFrame"
## attr(,"package")
## [1] "sp"
```

Un objet **SpatialGridDataFrame** est constitué des 4 éléments suivants :

- **data** : la table attributaire ;
- **grid** : la topologie de la grille ;
- **bbox** : l'emprise géographique ;
- **proj4string** : le système de coordonnées.

```
slotNames(SGrD)
## [1] "data"        "grid"        "bbox"        "proj4string"
```

La topologie de la grille se résume ainsi :

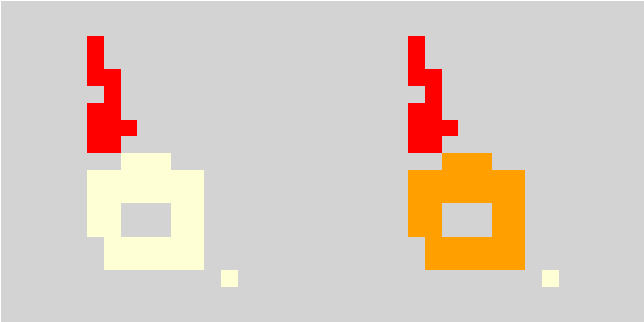
- les coordonnées de la maille inférieure gauche ;
- la taille des cellules ;
- les dimensions de la grille.

```
SGrD@grid
##                        X         Y
## cellcentre.offset 207345.9 1698671.2
## cellsize          121801.3  121801.3
## cells.dim              9.0      15.0
```

Avec le format **SpatialGridDataFrame**, pour une grille donnée, on peut associer très facilement plusieurs données attributaires contenues dans un objet de classe **data.frame**. Il existe donc un seul et unique format **sp** qui gère indifféremment les rasters simples et les rasters à plusieurs bandes.

Pour dessiner un objet **SpatialGridDataFrame**, on utilise la fonction **image()**, qui, par défaut, ne trace pas les axes et la légende. La fonction **plot()**, ne renvoie, quant à elle, que la position des centres des mailles de la grille.

```
par(mfrow = c(1, 2))
par(bg = "lightgrey")
image(SGrD["CNTRY"], asp = 1)
image(SGrD["IDE"]  , asp = 1)
```

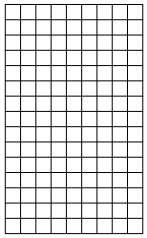


Méthode 2. On commence par définir la topologie d'une grille de classe **GridTopology**. Pour cela, il faut préciser, pour chaque dimension, les plus petites coordonnées (**cellcentre.offset**), la taille des cellules (**cellsize**), le nombre de cellules (**cells.dim**).

```
Gr <- GridTopology(cellcentre.offset = c(146445.3, 1637771.0),
                   cellsize          = c(121801.3,  121801.3),
                   cells.dim         = c(9, 15))
class(Gr)
## [1] "GridTopology"
## attr(,"package")
## [1] "sp"
```

Puis, on transforme cette grille en objet **SpatialGrid**. À cette étape, si on le souhaite, il est possible de préciser le système de coordonnées.

```
SGr <- SpatialGrid(grid = Gr, proj4string = CRS("+init=epsg:27572"))
class(SGr)
## [1] "SpatialGrid"
## attr(,"package")
## [1] "sp"
plot(SGr)
```



On constitue une table attributaire dans un **data.frame**, qui sera adjointe à l'objet **SpatialGrid** déjà existant. Les valeurs du **data.frame** doivent impérativement être définies au format numérique.

```
SGr_data <- data.frame(IDE = c(
                       01,NA,NA,NA,NA,NA,NA,NA,NA,
                       01,NA,NA,NA,NA,NA,NA,NA,NA,
                       01,01,NA,NA,NA,NA,NA,NA,NA,
                       NA,01,NA,NA,NA,NA,NA,NA,NA,
                       01,01,NA,NA,NA,NA,NA,NA,NA,
                       01,01,01,NA,NA,NA,NA,NA,NA,
                       01,01,NA,NA,NA,NA,NA,NA,NA,
                       NA,NA,02,02,02,NA,NA,NA,NA,
                       02,02,02,02,02,02,02,NA,NA,
                       02,02,02,02,02,02,02,NA,NA,
                       02,02,NA,NA,NA,02,02,NA,NA,
                       02,02,NA,NA,NA,02,02,NA,NA,
                       NA,02,02,02,02,02,02,NA,NA,
                       NA,02,02,02,02,02,02,NA,NA,
                       NA,NA,NA,NA,NA,NA,NA,NA,03),
                        CNTRY = c(
                       01,NA,NA,NA,NA,NA,NA,NA,NA,
                       01,NA,NA,NA,NA,NA,NA,NA,NA,
                       01,01,NA,NA,NA,NA,NA,NA,NA,
                       NA,01,NA,NA,NA,NA,NA,NA,NA,
                       01,01,NA,NA,NA,NA,NA,NA,NA,
                       01,01,01,NA,NA,NA,NA,NA,NA,
                       01,01,NA,NA,NA,NA,NA,NA,NA,
                       NA,NA,02,02,02,NA,NA,NA,NA,
                       02,02,02,02,02,02,02,NA,NA,
                       02,02,02,02,02,02,02,NA,NA,
                       02,02,NA,NA,NA,02,02,NA,NA,
                       02,02,NA,NA,NA,02,02,NA,NA,
                       NA,02,02,02,02,02,02,NA,NA,
```

```
                    NA,02,02,02,02,02,02,NA,NA,
                    NA,NA,NA,NA,NA,NA,NA,NA,02)
          )
```

On convertit l'objet **SpatialGrid** en objet **SpatialGridDataFrame** en lui adjoignant la table d'attributs.

```
SGrD <- SpatialGridDataFrame(grid = SGr, data = SGr_data)
class(SGrD)
## [1] "SpatialGridDataFrame"
## attr(,"package")
## [1] "sp"
```

On aboutit alors au même résultat qu'avec la méthode précédente.

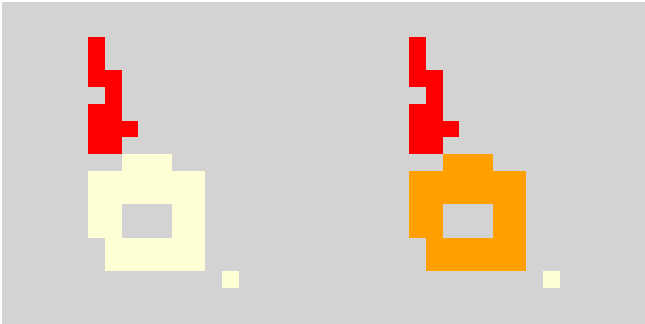
```
str(SGrD)
## Formal class 'SpatialGridDataFrame' [package "sp"] with 4 slots
##   ..@ data       :'data.frame': 135 obs. of  2 variables:
##   .. ..$ IDE  : num [1:135] 1 NA NA NA NA NA NA NA NA 1 ...
##   .. ..$ CNTRY: num [1:135] 1 NA NA NA NA NA NA NA NA 1 ...
##   ..@ grid       :Formal class 'GridTopology' [package "sp"] with 3 slots
##   .. .. ..@ cellcentre.offset: num [1:2] 146445 1637771
##   .. .. ..@ cellsize         : num [1:2] 121801 121801
##   .. .. ..@ cells.dim        : int [1:2] 9 15
##   ..@ bbox       : num [1:2, 1:2] 85545 1576870 1181756 3403890
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr "+init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=6000
```

1.2.1.3 Indexer des objets matriciels de classe sp

Les objets matriciels de classe **sp** s'indexent en utilisant la syntaxe "**[ ]**", comme pour les vecteurs.

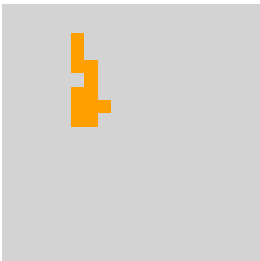
Dans l'exemple suivant (déjà eu plus haut), on réalise l'extraction d'un objet **SpatialGridDataFrame** par l'une ou l'autre des colonnes de la table attributaire. Autrement dit, on ne conserve qu'une colonne de la table.

```
par(mfrow = c(1, 2))
par(bg = "lightgrey")
image(SGrD["CNTRY"])
image(SGrD["IDE"])
```



Il est également possible de réaliser l'indexation sur l'élément *@data*.

```
par(bg = "lightgrey")
image((SGrD[SGrD@data[, "CNTRY"] == 1, ]))
```



1.2.2 Classe raster

Un autre format matriciel est souvent préféré, car très souple d'utilisation et très puissant, c'est la classe **raster** du package **raster** (Hijmans, 2015). Sa structure est très différente de celles des objets de classe **sp** (§ 1.2).

```
library(raster)
```

Il existe 3 sous-classes d'objet **raster** :
- **RasterLayer** : un raster à une seule bande ;
- **RasterStack** : un raster à plusieurs bandes (multiples fichiers), de mêmes emprises géographiques et de mêmes résolutions (équivaut à un groupe de **RasterLayer**) ;
- **RasterBrick** : un raster à plusieurs bandes (un seul fichier), de mêmes emprises géographiques et de mêmes résolutions (équivaut à un **RasterLayer** à plusieurs couches), plus rapide a manipuler en termes de temps de calcul, mais moins souple d'utilisation que les objets **RasterStack**.

Par ailleurs, il existe un autre objet de classe **raster**, très différent, c'est l'objet de sous-classe **extent**.

1.2.2.1 Classe Extent

L'objet **extent** n'est pas objet matriciel en tant que tel, mais il correspond à la définition de l'emprise géographique servant aux autres objets de classe **raster**.

Cet objet se définit *via* la fonction **extent()**, qui peut prendre en argument : un objet **raster**, les bornes de l'emprise géographique sous forme de vecteur (*xmin*, *xmax*, *ymin*, *ymax*) ou de matrice (1re ligne : *xmin*, *xmax* ; 2de ligne : *ymin*, *ymax*) ou encore l'élément **@bbox** d'un objet **sp**.

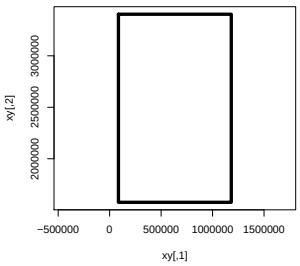
```
Rex <- extent(85544.65, 1181756, 1576870, 3403890)
Rex
## class       : Extent
## xmin        : 85544.65
## xmax        : 1181756
## ymin        : 1576870
## ymax        : 3403890
Rex <- extent(SGrD@bbox)
Rex
## class       : Extent
## xmin        : 85544.65
## xmax        : 1181756
## ymin        : 1576870
## ymax        : 3403890
```

Sa structure est très simple et se décline comme suit :

```
str(Rex)
## Formal class 'Extent' [package "raster"] with 4 slots
##   ..@ xmin: num 85545
##   ..@ xmax: num 1181756
##   ..@ ymin: num 1576870
##   ..@ ymax: num 3403890
```

On peut dessiner un objet **extent** par un simple appel à la fonction **plot()**.

```
plot(Rex, lwd = 5, asp = 1)
```

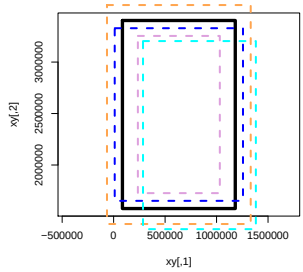


On peut modifier les limites d'un objet **extent** en réalisant une simple opération numérique sur celui-ci. Selon le résultat recherché, il n'est pas forcément utile de travailler avec un vecteur de 4 valeurs étant donné le fait que la règle de recyclage de R s'applique ici à ce type d'objet.

```
Rex1 <- Rex - 3e5
Rex2 <- Rex + 3e5
Rex3 <- Rex + c(+1.5e5, -1.5e5)
Rex4 <- Rex + c(+2e5, +2e5, - 2e5, -2e5)
```

Graphiquement, on obtient :

```
par(xpd = TRUE)
plot(Rex , lwd = 5, asp = 1)
plot(Rex1, col = "plum", lwd = 3, lty = 2, add = TRUE)
plot(Rex2, col = "tan1", lwd = 3, lty = 2, add = TRUE)
plot(Rex3, col = "blue", lwd = 3, lty = 2, add = TRUE)
plot(Rex4, col = "cyan", lwd = 3, lty = 2, add = TRUE)
```



Il est possible de convertir un objet **extent** en un objet **SpatialPolygons** de la manière suivante :

```
as(Rex, "SpatialPolygons")
## class       : SpatialPolygons
## features    : 1
## extent      : 85544.65, 1181756, 1576870, 3403890  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
```

1.2.2.2 Classe RasterLayer

Contrairement aux objets de classe **SpatialGridDataFrame** (§ 1.2.1.2), pour une grille donnée, avec les objets de classe **RasterLayer**, on ne dispose que d'une seule donnée attributaire.

Méthode 1. On peut obtenir un objet **raster** directement à partir d'une matrice. La matrice de départ est la suivante :

```
MaIDE <- matrix(SGr_data$IDE, ncol = 9, byrow = TRUE)
MaIDE
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,]    1   NA   NA   NA   NA   NA   NA   NA   NA
##  [2,]    1   NA   NA   NA   NA   NA   NA   NA   NA
##  [3,]    1    1   NA   NA   NA   NA   NA   NA   NA
##  [4,]   NA    1   NA   NA   NA   NA   NA   NA   NA
##  [5,]    1    1   NA   NA   NA   NA   NA   NA   NA
##  [6,]    1    1    1   NA   NA   NA   NA   NA   NA
##  [7,]    1    1   NA   NA   NA   NA   NA   NA   NA
##  [8,]   NA   NA    2    2    2   NA   NA   NA   NA
##  [9,]    2    2    2    2    2    2    2   NA   NA
## [10,]    2    2    2    2    2    2    2   NA   NA
## [11,]    2    2   NA   NA   NA    2    2   NA   NA
## [12,]    2    2   NA   NA   NA    2    2   NA   NA
## [13,]   NA    2    2    2    2    2    2   NA   NA
## [14,]   NA    2    2    2    2    2    2   NA   NA
## [15,]   NA   NA   NA   NA   NA   NA   NA   NA    3
```

On construit alors un objet de classe **RasterLayer** en fournissant la matrice précédemment créée à la fonction **raster()**, et dans laquelle il faudra définir l'emprise de la grille. À cette étape, on peut préciser le système de coordonnées.

```
RLa <- raster(MaIDE, xmn = 85544.65, xmx = 1181756, ymn = 1576870, ymx = 3403890,
              crs = CRS("+init=epsg:27572"))
RLa
## class       : RasterLayer
## dimensions  : 15, 9, 135  (nrow, ncol, ncell)
## resolution  : 121801.3, 121801.3  (x, y)
## extent      : 85544.65, 1181756, 1576870, 3403890  (xmin, xmax, ymin, ymax)
## coord. ref. : +init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_0=2200
## data source : in memory
## names       : layer
## values      : 1, 3  (min, max)
```

Méthode 2. Il est également possible de créer l'objet **RasterLayer** en fournissant un objet **SpatialGridDataFrame** à la fonction **raster()**. Dans ce cas là, il faut bien définir la colonne d'intérêt de la table attributaire (par défaut, si on ne le précise pas, ce sera la première). Si ce n'était pas déjà fait dans le **SpatialGridDataFrame**, on peut préciser ici le système de coordonnées (déjà fait dans le présent exemple).

```
RLa <- raster(SGrD["IDE"])
RLa
```

```
## class       : RasterLayer
## dimensions  : 15, 9, 135  (nrow, ncol, ncell)
## resolution  : 121801.3, 121801.3  (x, y)
## extent      : 85544.65, 1181756, 1576870, 3403890  (xmin, xmax, ymin, ymax)
## coord. ref. : +init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_0=2200
## data source : in memory
## names       : IDE
## values      : 1, 3  (min, max)
```

Méthode 3. On peut aussi créer un **RasterLayer** pour lequel on précise l'emprise géographique, ainsi que les dimensions de la grille. Les valeurs des mailles, quant à elles, sont entrées en tant que vecteur dans l'argument **vals**. On peut préciser ou non le système de coordonnées.

```
RLa<- raster(xmn = 85544.65, xmx = 1181756, ymn = 1576870, ymx = 3403890,
             nrows = 15, ncols = 9,
             vals = SGr_data$IDE,
             crs = CRS("+init=epsg:27572"))
```

Méthode 4. Il est également possible de créer un **RasterLayer** vide (ou non) en fournissant un objet **extent** à la fonction **raster()**. On crée un objet **extent** grâce à la fonction **extent()**, qui peut prendre 4 valeurs sous la forme d'un vecteur, d'une liste ou d'une matrice.

```
Ext <- extent(85544.65, 1181756, 1576870, 3403890)
Ext <- extent(list(x = c(85544.65, 1181756), y = c(1576870, 3403890)))
Ext <- extent(SGrD["IDE"]@bbox)
```

On fournit donc l'**extent** à la fonction **raster()**.

```
RLaNA <- raster(Ext, crs = CRS("+init=epsg:27572"), nrows = 15, ncols = 9)
RLaNA
## class       : RasterLayer
## dimensions  : 15, 9, 135  (nrow, ncol, ncell)
## resolution  : 121801.3, 121801.3  (x, y)
## extent      : 85544.65, 1181756, 1576870, 3403890  (xmin, xmax, ymin, ymax)
## coord. ref. : +init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_0=2200
```

Si le raster est vide, comme dans l'exemple, on assigne *a posteriori* des valeurs contenues dans un vecteur.

```
RLa <- RLaNA
values(RLa) <- SGr_data$IDE
RLa
## class       : RasterLayer
## dimensions  : 15, 9, 135  (nrow, ncol, ncell)
## resolution  : 121801.3, 121801.3  (x, y)
## extent      : 85544.65, 1181756, 1576870, 3403890  (xmin, xmax, ymin, ymax)
## coord. ref. : +init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_0=2200
## data source : in memory
## names       : layer
## values      : 1, 3  (min, max)
```

Méthode 5. On peut également créer un **RasterLayer** en utilisant la fonction **rasterFromXYZ()**, à partir d'un tableau (**matrix** ou **data.frame**) contenant 3 colonnes : la première pour les longitudes, la deuxième pour les latitudes, et la troisième pour les valeurs à assigner à chaque pixel. Bien entendu, les coordonnées devront êtres régulières, sans quoi la création du raster sera impossible.

Le **data.frame** de départ est le suivant :

```
head(RLa_tab)
##             X       Y IDE
## [1,] 146445.3 3342989   1
## [2,] 268246.6 3342989  NA
## [3,] 390047.9 3342989  NA
## [4,] 511849.2 3342989  NA
## [5,] 633650.5 3342989  NA
## [6,] 755451.8 3342989  NA
```

On le donne à la fonction **rasterFromXYZ()** et l'on définit le système de coordonnées.

```
RLa <- rasterFromXYZ(RLa_tab, crs = CRS("+init=epsg:27572"))
RLa
## class       : RasterLayer
## dimensions  : 15, 9, 135  (nrow, ncol, ncell)
## resolution  : 121801.3, 121801.3  (x, y)
## extent      : 85544.65, 1181756, 1576870, 3403890  (xmin, xmax, ymin, ymax)
## coord. ref. : +init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_0=2200
## data source : in memory
## names       : IDE
## values      : 1, 3  (min, max)
```

Comme déjà dit, la structure des objets de classe **RasterLayer** est plus complexe que celle des objets **SpatialGridDataFrame**. On dispose d'informations sur :

- **file** : le fichier matriciel ;

- **data** : les valeurs de la matrice ;
- **legend** : la légende (indisponible) ;
- **title** : le titre ;
- **extent** : l'emprise géographique ;
- **rotated** : le raster a subi une rotation ;
- **rotation** : les paramètres de la rotation (indisponible) ;
- **ncols** : le nombre de colonnes ;
- **nrows** : le nombre de lignes ;
- **crs** : le système de coordonnées ;
- **history** : l'enregistrement de l'historique de traitement (indisponible) ;
- **z** : (non défini et indisponible)

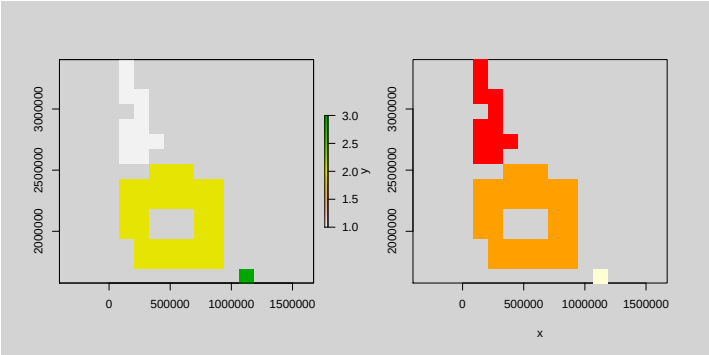
```
slotNames(RLa)
##  [1] "file"     "data"     "legend"   "title"    "extent"   "rotated"  "rotation" "ncols"
##  [9] "nrows"    "crs"      "history"  "z"
str(RLa)
## Formal class 'RasterLayer' [package "raster"] with 12 slots
##   ..@ file    :Formal class '.RasterFile' [package "raster"] with 13 slots
##   .. .. ..@ name        : chr ""
##   .. .. ..@ datanotation: chr "FLT4S"
##   .. .. ..@ byteorder   : chr "little"
##   .. .. ..@ nodatavalue : num -Inf
##   .. .. ..@ NAchanged   : logi FALSE
##   .. .. ..@ nbands      : int 1
##   .. .. ..@ bandorder   : chr "BIL"
##   .. .. ..@ offset      : int 0
##   .. .. ..@ toptobottom : logi TRUE
##   .. .. ..@ blockrows   : int 0
##   .. .. ..@ blockcols   : int 0
##   .. .. ..@ driver      : chr ""
##   .. .. ..@ open        : logi FALSE
##   ..@ data    :Formal class '.SingleLayerData' [package "raster"] with 13 slots
##   .. .. ..@ values    : num [1:135] 1 NA NA NA NA NA NA NA NA 1 ...
##   .. .. ..@ offset    : num 0
##   .. .. ..@ gain      : num 1
##   .. .. ..@ inmemory  : logi TRUE
##   .. .. ..@ fromdisk  : logi FALSE
##   .. .. ..@ isfactor  : logi FALSE
##   .. .. ..@ attributes: list()
##   .. .. ..@ haveminmax: logi TRUE
##   .. .. ..@ min       : num 1
##   .. .. ..@ max       : num 3
##   .. .. ..@ band      : int 1
##   .. .. ..@ unit      : chr ""
##   .. .. ..@ names     : chr "IDE"
##   ..@ legend  :Formal class '.RasterLegend' [package "raster"] with 5 slots
##   .. .. ..@ type      : chr(0)
##   .. .. ..@ values    : logi(0)
##   .. .. ..@ color     : logi(0)
##   .. .. ..@ names     : logi(0)
##   .. .. ..@ colortable: logi(0)
##   ..@ title   : chr(0)
##   ..@ extent  :Formal class 'Extent' [package "raster"] with 4 slots
##   .. .. ..@ xmin: num 85545
##   .. .. ..@ xmax: num 1181756
##   .. .. ..@ ymin: num 1576870
##   .. .. ..@ ymax: num 3403890
##   ..@ rotated : logi FALSE
##   ..@ rotation:Formal class '.Rotation' [package "raster"] with 2 slots
##   .. .. ..@ geotrans: num(0)
##   .. .. ..@ transfun:function ()
##   ..@ ncols   : int 9
##   ..@ nrows   : int 15
##   ..@ crs     :Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr "+init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=6000
##   ..@ history : list()
##   ..@ z       : list()
```

Notez que l'on peut convertir un objet de classe **RasterLayer** en objets **SpatialPixelsDataFrame** ou **SpatialGridDataFra** à l'aide de la fonction **as()**.

```
class(as(RLa, "SpatialPixelsDataFrame"))
## [1] "SpatialPixelsDataFrame"
## attr(,"package")
## [1] "sp"
class(as(RLa, "SpatialGridDataFrame"))
## [1] "SpatialGridDataFrame"
## attr(,"package")
## [1] "sp"
```

Ici, on a le choix de dessiner la carte à l'aide des fonctions **image()** ou **plot()**. Par défaut, la fonction **plot()** ajoute la légende, ce qui n'est pas le cas de la fonction **image()**. Les deux fonctions tracent les axes. Quand on utilise la fonction **image()**, il faut spécifier l'argument **asp = 1** pour conserver la même unité sur les axes des abscisses et des ordonnées (ce que fait par défaut la fonction **plot()** sur ce type d'objets).

```
par(mfrow = c(1, 2), bg = "lightgrey")
plot(RLa)
image(RLa, asp = 1)
```



1.2.2.3 Classes RasterBrick et RasterStack

On peut réaliser des rasters à plusieurs bandes de classes **RasterBrick** ou **RasterStack**.

Méthode 1. Pour créer un raster à plusieurs bandes de classe **RasterBrick**, on utilise la fonction **brick()**. On peut partir d'objets **SpatialPixelsDataFrame** ou **SpatialGridDataFrame** présentant plusieurs données attributaires.

```
RBr <- brick(SGrD)
class(RBr)
## [1] "RasterBrick"
## attr(,"package")
## [1] "raster"
```

Méthode 2. Il est également possible de fournir, à la fonction **brick()**, une liste de plusieurs rasters construits indépendamment les uns des autres, mais de mêmes dimensions et de mêmes résolutions.

```
RBr <- brick(list(raster(SGrD["IDE"]), raster(SGrD["CNTRY"])))
class(RBr)
## [1] "RasterBrick"
## attr(,"package")
## [1] "raster"
```

Méthode 3. On peut également créer un **RasterLayer** en utilisant la fonction **rasterFromXYZ()**, à partir d'un tableau (**matrix** ou **data.frame**), contenant plus de 3 colonnes : la première pour les longitudes, la deuxième pour les latitudes, et les suivantes pour les valeurs à assigner à chaque pixel. Bien entendu, les coordonnées devront êtres régulières, sans quoi la création du raster sera impossible.

Le **data.frame** de départ est le suivant :

```
head(RLa_tab)
##             X       Y IDE CNTRY
## [1,] 146445.3 3342989   1     1
## [2,] 268246.6 3342989  NA    NA
## [3,] 390047.9 3342989  NA    NA
## [4,] 511849.2 3342989  NA    NA
## [5,] 633650.5 3342989  NA    NA
## [6,] 755451.8 3342989  NA    NA
```

On le fournit à la fonction **rasterFromXYZ()** et l'on définit le système de coordonnées.

```
RBr <- rasterFromXYZ(RLa_tab, crs = CRS("+init=epsg:27572"))
class(RBr)
## [1] "RasterBrick"
## attr(,"package")
## [1] "raster"
```

Avec les 3 méthodes, le résultat obtenu est le suivant :

```
RBr
## class       : RasterBrick
## dimensions  : 15, 9, 135, 2  (nrow, ncol, ncell, nlayers)
## resolution  : 121801.3, 121801.3  (x, y)
## extent      : 85544.65, 1181756, 1576870, 3403890  (xmin, xmax, ymin, ymax)
## coord. ref. : +init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_0=2200
```

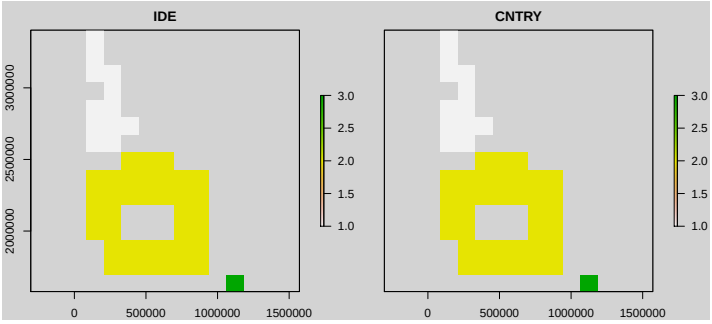
```
## data source : in memory
## names       : IDE, CNTRY
## min values  :   1,     1
## max values  :   3,     3
```

Notez que l'on peut convertir des objets de classes **RasterBrick** ou **RasterStack** en objets **SpatialPixelsDataFrame** ou **SpatialGridDataFrame** à l'aide de la fonction **as()**.

```
class(as(RBr, "SpatialPixelsDataFrame"))
## [1] "SpatialPixelsDataFrame"
## attr(,"package")
## [1] "sp"
class(as(RBr, "SpatialGridDataFrame"))
## [1] "SpatialGridDataFrame"
## attr(,"package")
## [1] "sp"
```

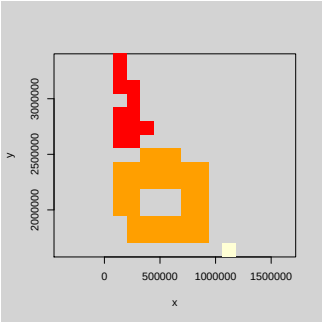
On affiche simultanément les cartes de toutes les bandes avec la fonction **plot()**.

```
par(bg = "lightgrey")
plot(RBr)
```



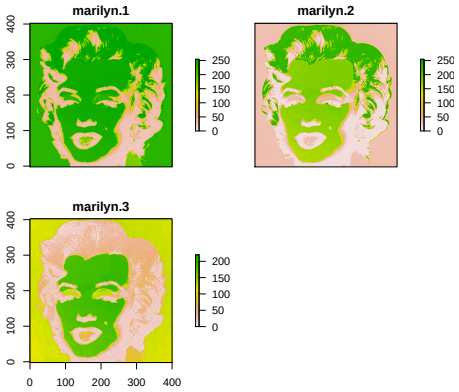
On peut afficher une seule bande avec les fonctions **plot()** ou **image()** en l'indexant avec des doubles crochets ("**[[ ]]**") :

```
par(bg = "lightgrey")
image(RBr[["CNTRY"]], asp = 1)
```



Dans le cas des rasters à 3 bandes de couleurs en RGB, un simple appel à la fonction **plot()** dessine un graphique pour chaque bande.

```
marylin
## class       : RasterBrick
## dimensions  : 400, 400, 160000, 3  (nrow, ncol, ncell, nlayers)
## resolution  : 1, 1  (x, y)
## extent      : 0, 400, 0, 400  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## data source : /home/irstea/Samba/smbe/Data/boulot/misc/scientific_doc/R/formation/irstea/033-geomatique_LIV/01_r
## names       : marilyn.1, marilyn.2, marilyn.3
## min values  :         0,         0,         0
## max values  :       255,       255,       255
plot(marylin)
```



Si l'on souhaite tracer le raster de mélange des 3 bandes (rouge verte et bleue), il faut utiliser la fonction **plotRGB()**.

```
plotRGB(marylin)
```

C'est exactement la même chose avec les rasters à plusieurs bandes de classe **RasterStack**.

1.2.2.4 Extraire des caractéristiques des objets matriciels de classe raster

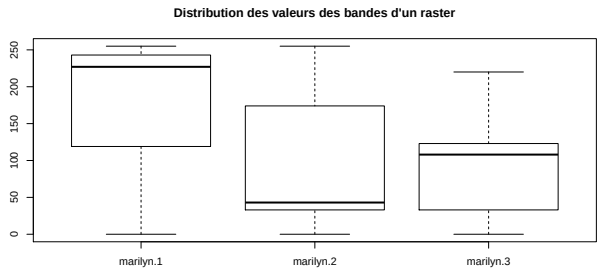
Plusieurs fonctions permettent ainsi d'extraire les caractéristiques d'un raster :

- **nrow()** : le nombre de lignes ;
- **ncol()** : le nombre de colonnes ;
- **dim()** : le nombre de lignes et nombre de colonnes ;
- **ncell()** : le nombre de cellules ;
- **coordinates()** : les coordonnées des centres des pixels ;
- **origin()** : les coordonnées du pixel le plus proche du point d'origine de coordonnées (0,0) ;
- **extent()** : l'emprise géographique ;
- **res()** : la résolution en X et en Y ;
- **nlayers()** : le nombre de couches (typiquement pour les rasters à plusieurs bandes).

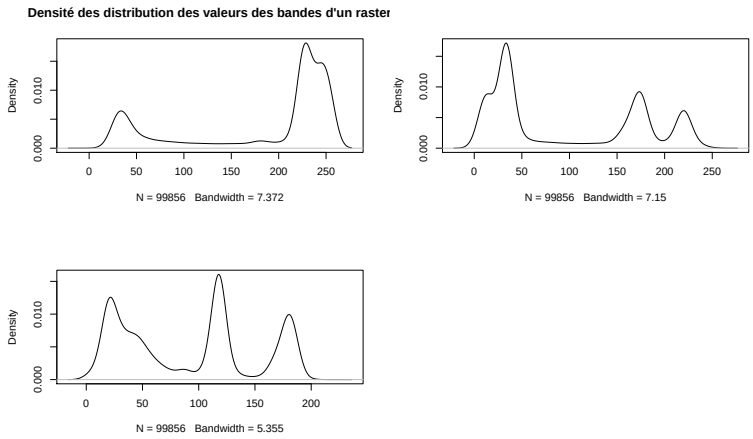
```
nrow(RLa)
## [1] 15
ncol(RLa)
## [1] 9
dim(RLa)
## [1] 15  9  1
ncell(RLa)
## [1] 135
coordinates(RLa)[1:4, ]
##             x       y
## [1,] 146445.3 3342989
## [2,] 268246.6 3342989
## [3,] 390047.9 3342989
## [4,] 511849.2 3342989
origin(RLa)
## [1] -36256.65  -6546.55
extent(RLa)
## class       : Extent
## xmin        : 85544.65
## xmax        : 1181756
## ymin        : 1576870
## ymax        : 3403890
res(RLa)
## [1] 121801.3 121801.3
nlayers(RLa)
## [1] 1
```

Par ailleurs, afin de mieux apprécier la distribution des valeurs d'un raster, le package **raster** permet de tracer des boîtes à moustaches et des graphes de densité de rasters simples ou à plusieurs bandes, et ce grâce aux fonctions **boxplot()** et **density()**.

```
boxplot(marylin, main = "Distribution des valeurs des bandes d'un raster")
```

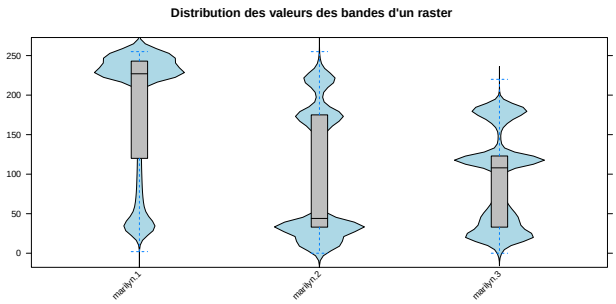



```
density(marylin,  main = "Densité des distribution des valeurs des bandes d'un raster")
```

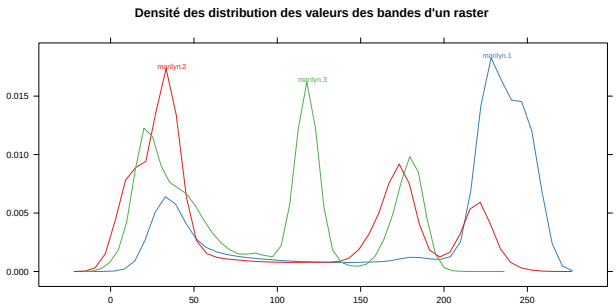


Le package **rasterVis** (Perpiñán & Hijmans, 2014), propose les fonctions **bwplot()** et **densityplot()**, qui permettent de réaliser les mêmes types de représentations, mais un peu plus travaillées.

```
rasterVis::bwplot(marylin, main = "Distribution des valeurs des bandes d'un raster")
```




```
rasterVis::densityplot(marylin, main = "Densité des distribution des valeurs des bandes d'un raster")
```



1.2.2.5 Indexer objets matriciels de classe `raster`

Les objets matriciels de classe **raster** s'indexent en utilisant des crochets.

Il est possible d'extraire l'ensemble des valeurs de mailles avec la fonction **values()** ou de simples crochets vides "**[ ]**".

```
values(RLa)
##   [1]  1 NA NA NA NA NA NA NA NA  1 NA NA NA NA NA NA NA NA  1  1 NA NA NA NA NA NA NA NA  1 NA
##  [31] NA NA NA NA NA NA  1  1 NA NA NA NA NA NA NA  1  1  1 NA NA NA NA NA NA  1  1 NA NA NA NA
##  [61] NA NA NA NA NA  2  2  2 NA NA NA NA  2  2  2  2  2  2  2 NA NA  2  2  2  2  2  2  2 NA NA
##  [91]  2  2 NA NA NA  2  2 NA NA  2  2 NA NA NA  2  2 NA NA NA  2  2  2  2  2  2 NA NA NA  2  2
## [121]  2  2  2  2 NA NA NA NA NA NA NA NA NA NA  3
RLa[]
##   [1]  1 NA NA NA NA NA NA NA NA  1 NA NA NA NA NA NA NA NA  1  1 NA NA NA NA NA NA NA NA  1 NA
##  [31] NA NA NA NA NA NA  1  1 NA NA NA NA NA NA NA  1  1  1 NA NA NA NA NA NA  1  1 NA NA NA NA
##  [61] NA NA NA NA NA NA  2  2  2 NA NA NA NA  2  2  2  2  2  2  2 NA NA  2  2  2  2  2  2  2 NA NA
##  [91]  2  2 NA NA NA  2  2 NA NA  2  2 NA NA NA  2  2 NA NA NA  2  2  2  2  2  2 NA NA NA  2  2
## [121]  2  2  2  2 NA NA NA NA NA NA NA NA NA NA  3
```

Pour extraire des lignes, des colonnes ou des valeurs d'un **raster**, on utilise la syntaxe "**[,]**", comme pour l'indexation d'une matrice.

```
RLa[2, ] # ligne 2
## [1]  1 NA NA NA NA NA NA NA NA
RLa[ ,9] # colonne 9
##  [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA  3
RLa[2,9] # ligne 2 et colonne 9
##
## NA
```

Comme c'est également le cas pour les matrices, si l'on souhaite uniquement extraire des valeurs d'un **raster** (et non des lignes ou des colonnes), on peut aussi utiliser la syntaxe "**[ ]**" ou la fonction **extract()**.

```
RLa[19] # valeur 19 (ligne 2 et colonne 9)
##
## 1
extract(RLa, 19)
##
## 1
```

Il est également possible d'extraire des valeurs connaissant les indices des lignes et/ou des colonnes à partir des fonctions **cellFromRow()**, **cellFromCol()** ou **cellFromRowCol()** qui renvoient des numéros des cellules.

```
cellFromRow(RLa, rownr = 2)
## [1] 10 11 12 13 14 15 16 17 18
RLa[cellFromRow(RLa, rownr = 2)]
## [1]  1 NA NA NA NA NA NA NA NA
cellFromCol(RLa, colnr = 9)
##  [1]   9  18  27  36  45  54  63  72  81  90  99 108 117 126 135
RLa[cellFromCol(RLa, colnr = 9)]
##  [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA  3
cellFromRowCol(RLa, rownr = 2, colnr = 9)
## [1] 18
RLa[cellFromRowCol(RLa, rownr = 2, colnr = 9)]
##
## NA
```

Pour récupérer une ou plusieurs des couches d'un objet de classe **RasterStack** ou **RasterBrick**, il faut utiliser la fonction **subset()**. L'argument **subset**, peut prendre le numéro de la bande (**integer**) ou son nom **character**. Si l'on passe un vecteur de longueur 1 à l'argument **subset**, on récupère alors un objet de classe **RasterLayer**, sinon on conserve la classe d'origine. Notes que l'on peut également utiliser les doubles crochets ("**[[ ]]**") pour effectuer l'extraction des couches d'intérêt

```
subset(marylin, subset = 1)
## class       : RasterLayer
## band        : 1  (of  3  bands)
## dimensions  : 400, 400, 160000  (nrow, ncol, ncell)
## resolution  : 1, 1  (x, y)
## extent      : 0, 400, 0, 400  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## data source : /home/irstea/Samba/smbe/Data/boulot/misc/scientific_doc/R/formation/irstea/033-geomatique_LIV/01_r
## names       : marilyn.1
## values      : 0, 255  (min, max)

marylin[[1:2]]
```

1.3 Objets de mode S3

Outre les objets de mode **S4** qui viennent d'être présentés, il existe, dans R, d'autres formats d'objets spatiaux, tous de mode **S3**. Ces derniers étant moins usités, ils ne seront présentés que brièvement.

Les principaux packages permettant de manipuler les divers formats spatiaux de mode **S3** sont les suivants : **maps** (Becker R.A. (original S code) *et al.*, 2014b), **PBSmapping** (Schnute *et al.*, 2015, 2004), **spatstat** (Baddeley & Turner, 2005 ; Baddeley *et al.*, 2013). Leur utilisation peut présenter un intérêt pour leurs bases de données géographiques ou leurs fonctionnalités complémentaires qui sont parfois plus efficientes que celles qui manipulent les objets **S4**. Ils demeurent tout de même souvent moins pratiques à manipuler en raison de leur faible compatibilité avec la majorité des autres packages de géomatique.

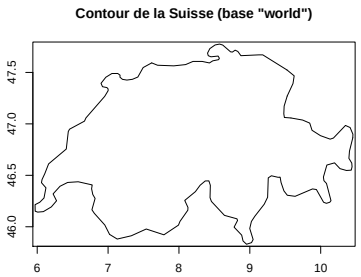
1.3.1 Classe map

Le package **maps** (Becker R.A. (original S code) *et al.*, 2014b) permet de dessiner très facilement des cartes. À l'aide de la fonction **map()**, on charge des données retournées sous forme d'une liste, et l'on dessine la carte en même temps. Les données sont directement issues du package ; on n'a donc pas besoin de faire appel à un fichier extérieur. Par défaut, les coordonnées sont exprimées en degrés décimaux en WGS 84. On peut dessiner une carte mondiale ou celle d'un pays en particulier.

```
library(maps)
```

Dans l'exemple suivant, on décide de tracer la carte de la Suisse. Le contour du polygone est stocké dans la base de données *world*.

```
map_SW1 <- map(database = "world", regions = "switzerland")
map.axes()
title('Contour de la Suisse (base "world")')
```




```
class(map_SW1)
## [1] "map"
```

Un objet **map** présente une structure bien plus simple que celle des objets de classe **sp** ; il se présente sous forme d'une liste de 4 éléments.

- **x** : la longitude des points de contour ;
- **y** : la latitude des points de contour ;
- **range** : l'emprise géographique ;
- **names** : les noms des entités.

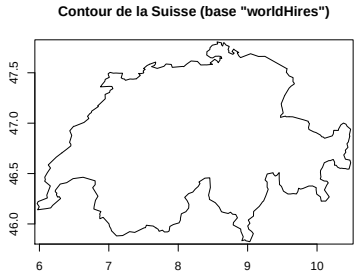
```
str(map_SW1)
## List of 4
##  $ x    : num [1:156] 9.53 9.61 9.63 9.52 NA ...
##  $ y    : num [1:156] 47.3 47.4 47.5 47.5 NA ...
##  $ range: num [1:4] 5.97 10.45 45.83 47.78
##  $ names: chr "Switzerland"
##  - attr(*, "class")= chr "map"
```

Ces cartes sont légères, mais peu précises. Le package **mapdata** (Becker R.A. (original S code) *et al.*, 2014a) possède des données avec une résolution un peu plus élevée.

```
library(mapdata)
```

Si l'on reprend l'exemple du contour de la Suisse, on peut voir que le nombre de points composant le polygone de contour est ici plus important. Le contour du polygone est stocké dans la base de données *worldHires*.

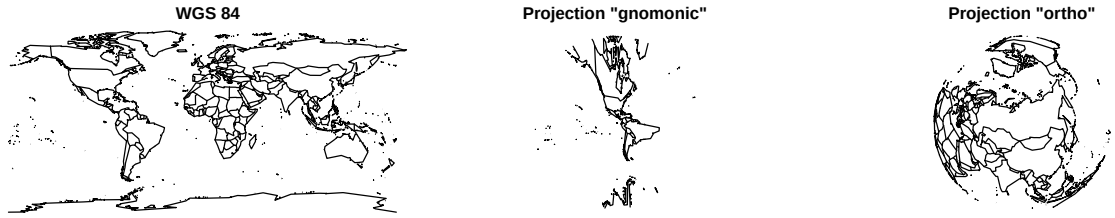
```
data(worldHiresMapEnv)
map_SW2 <- map(database = "worldHires", region = "switzerland")
map.axes()
title('Contour de la Suisse (base "worldHires")')
```




```
class(map_SW2)
## [1] "map"
str(map_SW2)
## List of 4
##  $ x    : num [1:399] 7.68 7.71 7.72 7.77 7.8 ...
##  $ y    : num [1:399] 46 45.9 45.9 45.9 45.9 ...
##  $ range: num [1:4] 5.97 10.49 45.82 47.81
##  $ names: chr "Switzerland"
##  - attr(*, "class")= chr "map"
```

Par défaut, la fonction **map()** utilise le WGS 84, mais il est aisé d'utiliser d'autres systèmes de coordonnées et de gérer l'orientation de la projection.

```
par(mfrow = c(1, 3))
map("world", mar = c(0, 0, 2, 0)) ; title('WGS 84')
map("world", proj = "gnomonic", orientation = c( 0,-100, 0), mar = c(0, 0, 2, 0))
  title('Projection "gnomonic"')
map("world", proj = "ortho"   , orientation = c(60,  90, 0), mar = c(0, 0, 2, 0))
  title('Projection "ortho"')
```



La compatibilité entre les objets **map** et les objets **sp** est assurée par le package **maptools** (Bivand & Lewin-Koh, 2015). Ce dernier propose les fonctions suivantes :

- **map2SpatialLines()**,
- **map2SpatialPolygons()**.

Attention, pour autoriser la conversion en objet **sp**, l'argument **fill** de la fonction **map()** doit être défini à la valeur **TRUE**, sans quoi la cohérence du contour n'est pas assurée, ce qui empêche l'obtention du bon nombre d'entités (lignes ou polygones).

```
map_libya <- map(database = "world", region = "libya", fill = TRUE, plot = FALSE)
IDs <- rep("Libye", length(map_libya$names))

sp_libya_Li <- map2SpatialLines(map_libya, IDs = IDs, proj4string = CRS("+proj=longlat +datum=WGS84"))
sp_libya_Li
## class       : SpatialLines
## features    : 1
## extent      : 9.310254, 25.15049, 19.49663, 33.18193  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
sp_libya_Po <- map2SpatialPolygons(map_libya, IDs = IDs, proj4string = CRS("+proj=longlat +datum=WGS84"))
sp_libya_Po
## class       : SpatialPolygons
## features    : 1
## extent      : 9.310254, 25.15049, 19.49663, 33.18193  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

Bien qu'aisées à réaliser, ces cartes restent relativement basiques. Par ailleurs, on est ici limité aux données stockées dans les packages. La plupart du temps, on souhaite travailler avec des fichiers géographiques externes,

tels que des fichiers au format Shapefile.

1.3.2 Classes EventData et PolySet

Le package **PBSmapping** (Schnute *et al.*, 2015, 2004) présente d'autres formats de données et en particulier les classes **EventData** et **PolySet**. L'intérêt de ce package est qu'il propose plusieurs fonctionnalités d'analyses spatiales qui sont très efficientes.

Ce package ne travaille pas dans tous les systèmes de coordonnées, mais seulement en coordonnées géographiques WGS 84 (code EPSG 4326) ou avec les projections UTM et Mercator (code EPSG 3857).

```
library(PBSmapping)
```

Les objets vectoriels de points sont de classe **EventData**, et correspondent à des **data.frame** comportant impérativement les colonnes suivantes :

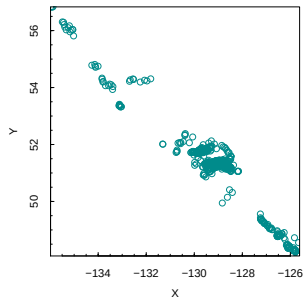
- *EID* : l'identifiant ;
- *X* : la longitude ;
- *Y* : la latitude.

```
data(surveyData)
class(surveyData)
## [1] "EventData"  "data.frame"
str(surveyData)
## Classes 'EventData' and 'data.frame': 674 obs. of  9 variables:
##  $ EID   : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ X     : num  -129 -129 -129 -129 -129 ...
##  $ Y     : num  51.4 51.4 51.4 51.3 51.4 ...
##  $ trip  : num  41333 41333 41333 41333 41333 ...
##  $ tow   : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ catch : num  828 773 295 596 699 ...
##  $ effort: num  30 32 31 30 30 30 30 30 31 30 ...
##  $ depth : num  220 212 176 252 219 ...
##  $ year  : num  1966 1966 1966 1966 1966 ...
##  - attr(*, "projection")= chr "LL"
##  - attr(*, "zone")= num 9
```

On peut convertir un **data.frame** en **EventData** avec la fonction **as.EventData()**. La définition du système de coordonnées est facultative.

La carte est tracée *via* un appel à la fonction **plotPoints()**.

```
plotPoints(surveyData, col = "cyan4")
```



Il n'existe pas de distinction entre les objets vectoriels contenant des lignes ou des points. Ils sont tous deux de classe **PolySet**. Ils comportent les colonnes suivantes :

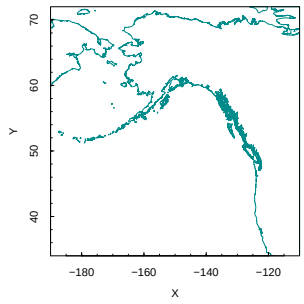
- *PID* : l'identifiant primaire de la sous-entité ;
- *SID* : l'identifiant secondaire de la sous-entité (facultatif) ;
- *POS* : la position du point au sein de la sous-entité (décroissant s'il s'agit d'un trou) ;
- *X* : la longitude ;
- *Y* : la latitudes.

```
data(nepacLL)
class(nepacLL)
## [1] "PolySet"    "data.frame"
str(nepacLL)
## Classes 'PolySet' and 'data.frame': 75305 obs. of  4 variables:
##  $ PID: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ POS: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ X  : num  -180 -180 -180 -180 -180 ...
##  $ Y  : num  69 69 69 69 69 ...
##  - attr(*, "PolyData")='data.frame': 0 obs. of  0 variables
```

```
##  - attr(*, "projection")= chr "LL"
```

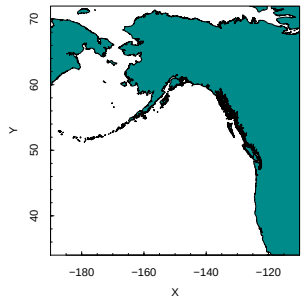
On trace des lignes avec la fonction **plotLines()**.

```
plotLines(nepacLL, col = "cyan4")
```



On trace des polygones avec la fonction **plotPolys()**.

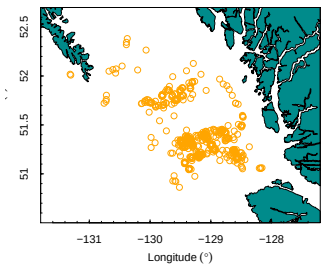
```
plotPolys(nepacLL, col = "cyan4")
```



Les fonctions **plotLines()** et **plotPolys()** ne respectent pas le ratio entre les axes. Pour palier à ce comportement, on peut utiliser la fonction **plotMap()**.

Pour superposer des couches, il faut faire appel aux fonction **addPoints()**, **addLines()** ou **addPolys()**.

```
plotMap(nepacLL, col = "cyan4", xlim = c(-131.8, -127.2), ylim = c(50.5, 52.7))
addPoints(surveyData, col = "orange")
```



Pour d'autres types de représentations graphiques, on peut appeler les fonctions **addBubbles()**, **addStipples()**, **addLabels()**.

La compatibilité entres les objets **PolySet** et les objets **sp** est assurée par le package **maptools** (Bivand & Lewin-Koh, 2015). Il propose les fonctions suivantes :

- **PolySet2SpatialLines()** ;
- **PolySet2SpatialPolygons()** ;
- **SpatialLines2PolySet()** ;
- **SpatialPolygons2PolySet()**.

1.3.3 Classes ppp, psp et owin

Le package **spatstat** (Baddeley & Turner, 2005 ; Baddeley *et al.*, 2013) présente d'autres formats de données et en particulier les classes **owin**, **ppp psp** et **im**. Ce package propose de très nombreuses fonctionnalités d'analyses spatiales, en particulier pour le traitement des semis de points.

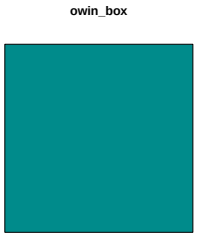
```
library(spatstat)
```

La première étape consiste à définir une emprise géographique sur laquelle travailler. Pour cela, on utilise la fonction **owin()**. La forme de l'emprise n'est pas obligatoirement rectangulaire ; elle peut être polygonale et même comporter des trous.

```
owin_box <- owin(c(0, 1), c(0, 1))
class(owin_box)
## [1] "owin"
str(owin_box)
## List of 4
##  $ type  : chr "rectangle"
##  $ xrange: num [1:2] 0 1
##  $ yrange: num [1:2] 0 1
##  $ units :List of 3
##   ..$ singular  : chr "unit"
##   ..$ plural    : chr "units"
##   ..$ multiplier: num 1
##   ..- attr(*, "class")= chr "units"
##  - attr(*, "class")= chr "owin"
```

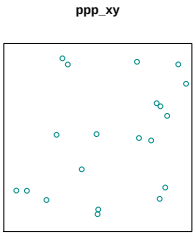
On peut dessiner l'objet **owin** avec la fonction **plot**.

```
plot(owin_box, col = "cyan4")
```



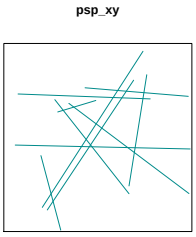
Pour créer un pattern de points, il faut utiliser la fonction **ppp()**. L'objet ainsi créé est de classe **ppp**. Il peut être dessiné *via* un appel à la fonction **plot()**.

```
ppp_xy <- ppp(runif(20), runif(20), window = owin_box)
class(ppp_xy)
## [1] "ppp"
str(ppp_xy)
## List of 5
##  $ window    :List of 4
##   ..$ type  : chr "rectangle"
##   ..$ xrange: num [1:2] 0 1
##   ..$ yrange: num [1:2] 0 1
##   ..$ units :List of 3
##   .. ..$ singular  : chr "unit"
##   .. ..$ plural    : chr "units"
##   .. ..$ multiplier: num 1
##   .. ..- attr(*, "class")= chr "units"
##   ..- attr(*, "class")= chr "owin"
##  $ n         : int 20
##  $ x         : num [1:20] 0.498 0.414 0.28 0.708 0.928 ...
##  $ y         : num [1:20] 0.0922 0.3308 0.5144 0.9026 0.8889 ...
##  $ markformat: chr "none"
##  - attr(*, "class")= chr "ppp"
plot(ppp_xy, cols = "cyan4")
```



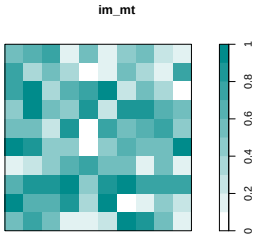
Pour créer des lignes, il faut utiliser la fonction **psp()**. L'objet ainsi créé est de classe **ppp**. Il peut être dessiné *via* un appel à la fonction **plot()**.

```
psp_xy <- psp(runif(10), runif(10), runif(10), runif(10), window = owin_box)
class(psp_xy)
## [1] "psp"  "list"
str(psp_xy)
## List of 4
##  $ ends      :'data.frame': 10 obs. of  4 variables:
##   ..$ x0: num [1:10] 0.778 0.205 0.303 0.231 0.76 ...
##   ..$ y0: num [1:10] 0.70524 0.12851 0.00761 0.11471 0.83355 ...
##   ..$ x1: num [1:10] 0.0756 0.7399 0.1974 0.6903 0.6657 ...
##   ..$ y1: num [1:10] 0.731 0.958 0.404 0.805 0.243 ...
##  $ window    :List of 4
##   ..$ type  : chr "rectangle"
##   ..$ xrange: num [1:2] 0 1
##   ..$ yrange: num [1:2] 0 1
##   ..$ units :List of 3
##   .. ..$ singular  : chr "unit"
##   .. ..$ plural    : chr "units"
##   .. ..$ multiplier: num 1
##   .. ..- attr(*, "class")= chr "units"
##   ..- attr(*, "class")= chr "owin"
##  $ n         : int 10
##  $ markformat: chr "none"
##  - attr(*, "class")= chr [1:2] "psp" "list"
plot(psp_xy, col = "cyan4")
```



Pour créer un raster, on utilise la fonction **im()**.

```
im_mt <- im(matrix(runif(100), ncol = 10))
class(im_mt)
## [1] "im"
str(im_mt)
## List of 10
##  $ v     : num [1:10, 1:10] 0.562 0.929 0.639 0.154 0.967 ...
##  $ dim   : int [1:2] 10 10
##  $ xrange: num [1:2] 0.5 10.5
##  $ yrange: num [1:2] 0.5 10.5
##  $ xstep : num 1
##  $ ystep : num 1
##  $ xcol  : num [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ yrow  : num [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ type  : chr "real"
##  $ units :List of 3
##   ..$ singular  : chr "unit"
##   ..$ plural    : chr "units"
##   ..$ multiplier: num 1
##   ..- attr(*, "class")= chr "units"
##  - attr(*, "class")= chr "im"
plot(im_mt, col = colourmap(colorRampPalette(c("white", "cyan4"))(10), breaks = seq(0, 1, length = 11)))
```



La compatibilité entres les objets **ppp psp**, **owin** et les objets **sp** est assurée par le package **maptools** (Bivand & Lewin-Koh, 2015) . Il propose les fonctions suivantes :

- `as.SpatialPoints.ppp()` ;
- `as.SpatialPointsDataFrame.ppp()` ;
- `as.SpatialGridDataFrame.ppp)` ;
- `as.SpatialGridDataFrame.im()` ;
- `as.psp.Line()` ;
- `as.psp.Lines()` ;
- `as.psp.SpatialLines()` ;
- `as.psp.SpatialLinesDataFrame()` ;
- `as.SpatialLines.psp()` ;
- `as.SpatialPolygons.tess()` ;
- `as.SpatialPolygons.owin()`.

Chapitre 2

Import et export de données

2.1 Données attributaires

Le package **foreign** (R Core Team, 2015) permet d'importer les données attributaires contenues dans un fichier DBF.

```
library(foreign)
```

Pour cela, il suffit d'utiliser la fonction **read.dbf()**.

```
WD_tb_file <- paste("../00_data", dir_VEC, "world_cntry_WGS84.dbf", sep = "/")
WD_ctr_W84_tb <- read.dbf(file = WD_tb_file)
```

Le résultat est renvoyé sous forme de **data.frame**.

```
str(WD_ctr_W84_tb)
## 'data.frame': 1627 obs. of  2 variables:
##  $ ID   : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ names: Factor w/ 1625 levels "Afghanistan",..: 125 1 6 7 8 2 563 561 562 5 ...
##  - attr(*, "data_types")= chr  "N" "C"
```

2.2 Données vectorielles

R permet d'importer des fichiers géographiques externes et ce, dans de nombreux formats. Le package **rgdal** (Bivand *et al.*, 2014) fournit les principaux outils permettant l'importation de fichiers géographiques externes. La fonction **ogrDrivers()** permet de lister les formats que le package **rgdal** permet d'importer et ceux qu'il permet d'exporter (tab. 2.1). Il est donc possible de convertir des fichiers d'un format à l'autre en passant par R.

```
head(rgdal::ogrDrivers())
##         name write
## 1 AeronavFAA FALSE
## 2     ARCGEN FALSE
## 3     AVCBin FALSE
## 4     AVCE00 FALSE
## 5        BNA  TRUE
## 6    CartoDB FALSE
```

2.2.1 Package maptools

Le package **maptools** (Bivand & Lewin-Koh, 2015) permet d'importer ou d'exporter des données vectorielles au format Shapefile. Par ailleurs, comme nous venons de le voir, il permet la conversion entre des objets spatiaux de classe **sp** et d'autres classes moins conventionnelles (§ 1.3.1, 1.3.2 et 1.3.3). Il propose également de nombreuses fonctionnalités d'analyse spatiale.

```
library(maptools)
```

2.2.1.1 Import

Le package **maptools** permet de lire des cartes à partir de fichiers ESRI Shapefile. On peut importer les données en utilisant 4 fonctions différentes :

- **readShapePoints()** : pour les fichiers contenant des points ;
- **readShapeLines()** : pour les fichiers contenant des lignes ;
- **readShapePoly()** : pour les fichiers contenant des polygones ;
- **readShapeSpatial()** : pour les fichiers contenant n'importe quel type d'entités.

TABLEAU 2.1 – Liste des formats de fichiers vectoriels importables et exportables avec les pilotes OGR du package **rgdal**.

Format	Écriture	Format	Écriture	Format	Écriture
AeronavFAA	non	GPSBabel	oui	PGDump	oui
ARCGEN	non	GPSTrackMaker	oui	PGeo	non
AVCBin	non	GPX	oui	PostgreSQL	oui
AVCE00	non	HTF	non	REC	non
BNA	oui	Idrisi	non	S57	oui
CartoDB	non	Interlis 1	oui	SDTS	non
CouchDB	oui	Interlis 2	oui	SEGUKOOA	non
CSV	oui	KML	oui	SEGY	non
DGN	oui	LIBKML	oui	SQLite	oui
DODS	non	MapInfo File	oui	SUA	non
DXF	oui	Memory	oui	SVG	non
EDIGEO	non	MSSQLSpatial	oui	SXF	non
ElasticSearch	oui	MySQL	oui	TIGER	oui
ESRI Shapefile	oui	NAS	non	UK .NTF	non
Geoconcept	oui	ODBC	oui	VFK	non
GeoJSON	oui	ODS	oui	VRT	non
Geomedia	non	OGDI	non	Walk	non
GeoRSS	oui	OpenAir	non	WAsP	oui
GFT	oui	OpenFileGDB	non	WFS	non
GME	oui	OSM	non	XLS	non
GML	oui	PCIDSK	oui	XLSX	oui
GMT	oui	PDF	oui	XPlane	non
GPKG	oui	PDS	non		

Lorsqu'on importe un Shapefile, on peut spécifier ou non son système de coordonnées. Le système n'est pas importé depuis le fichier de projection (".prj"), même s'il existe.

Dans l'exemple suivant, on importe les données d'un Shapefile contenant les polygones des pays du monde, définis dans le système de coordonnées WGS 84 (code ESPG 4326).

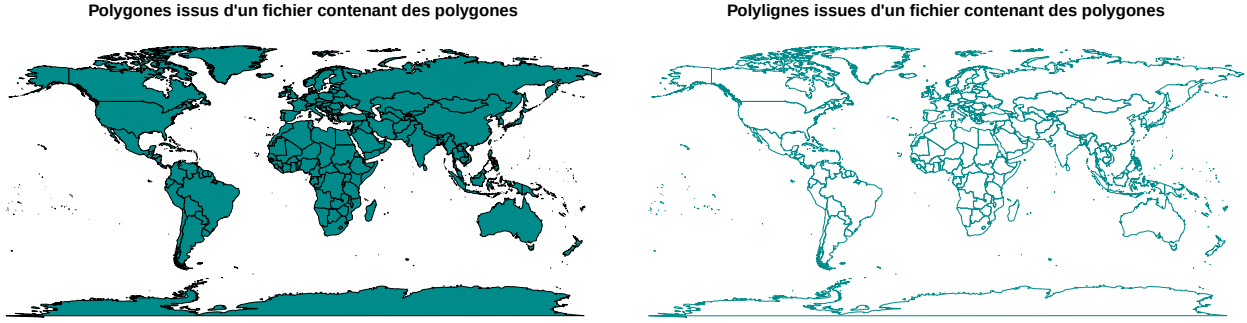
```
WD_file <- paste(dir_VEC, "world_cntry_WGS84.shp", sep = "/")
WD_ctr_W84_po <- readShapePoly(WD_file, proj4string = CRS("+init=epsg:4326"))
```

Il est possible de lire un fichier contenant des polygones en ne conservant que les contours (pour des questions de poids d'objet, par exemple) en réalisant l'importation comme s'il s'agissait de lignes. Dans l'exemple suivant, on charge les polygones en tant que lignes de contour.

```
WD_ctr_W84_li <- readShapeLines(WD_file, proj4string = CRS("+init=epsg:4326"))
```

Ici, on voit que les deux objets créés à partir d'un même fichier sont bien de classes différentes.

```
class(WD_ctr_W84_po)
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
class(WD_ctr_W84_li)
## [1] "SpatialLinesDataFrame"
## attr(,"package")
## [1] "sp"
par(mfrow = c(1, 2))
plot(WD_ctr_W84_po, col = "cyan4", main = "Polygones issus d'un fichier contenant des polygones")
plot(WD_ctr_W84_li, col = "cyan4", main = "Polylignes issues d'un fichier contenant des polygones")
```



On peut constater que si l'on force les données à être considérées en polylignes plutôt qu'en polygones, l'objet créé est plus léger.

```
object.size(WD_ctr_W84_po)
## 6344232 bytes
object.size(WD_ctr_W84_li)
## 4476072 bytes
```

2.2.1.2 Export

De la même manière que pour l'import, le package **maptools** permet d'écrire des fichiers ESRI Shapefile. On peut exporter les données en utilisant 4 fonctions différentes :

- **writePointsShape()** : pour les fichiers contenant des points ;
- **writeLinesShape()** : pour les fichiers contenant des lignes ;
- **writePolyShape()** : pour les fichiers contenant des polygones ;
- **writeSpatialShape()** : pour les fichiers contenant n'importe quel type d'entités.

Ces fonctions n'écrivent pas le fichier de projection (".prj").

```
writeLinesShape(WD_ctr_W84_li, "WD_ctr_W84_li.shp")
```

2.2.2 Package rgdal

Le package **rgdal** (Bivand *et al.*, 2014) implémente les outils OGR dans R. Il permet donc de lire et d'écrire de nombreux formats de fichiers spatiaux de données vectorielles.

```
library(rgdal)
```

2.2.2.1 Import

Avec le package **rgdal**, on importe les fichiers grâce à la fonction **readOGR()**. Ici, si le fichier contenant le système de coordonnées (".prj") accompagnant le Shapefile existe, il sera lu. Par défaut, le système sera alors reconnu automatiquement dans R. Toutefois, il est parfois recommandé de forcer la définition du système dans l'argument **p4s**.

Attention, sous Windows, l'argument **dsn** (chemin du répertoire contenant le fichier) ne doit pas comporter de *slash* terminal, et le nom de la couche, défini dans l'argument **layer**, ne doit pas comporter l'extension ".shp". C'est vrai pour les fichiers Shapefile, mais ce n'est pas le cas pour tous les formats de fichiers ; cela dépend des pilotes informatiques utilisés (**driver**). D'ailleurs la définition même des arguments **dsn** et **layer** varient selon le pilote utilisé. L'argument **p4s** permet de définir la système de coordonnées, même si ce dernier est déjà contenu dans la couche importée.

```
WD_ctr_W84_po <- readOGR(dsn = dir_VEC, layer = "world_cntry_WGS84", p4s = "+init=epsg:4326")
```

2.2.2.2 Export

On exporte des données à l'aide de la fonction **writeOGR()**. Pour écrire des fichiers Shapefile, on utilise le pilote **"ESRI Shapefile"**. Avec cette fonction le fichier de projection (".prj") des Shapefile est créé. Comme lors de l'import, sous Windows, l'argument **dsn** ne doit pas présenter de slash terminal, et le nom de la couche ne doit pas comporter l'extension ".shp".

```
writeOGR(WD_ctr_W84_po, dsn = ".", layer = "WD_ctr_W84_po", driver = "ESRI Shapefile")
```

2.3 Données matricielles

Outre les données vectorielles, on peut importer des données matricielles sous R. De nombreux formats de rasters sont gérés. La fonction **getGDALDriverNames()** du package **rgdal** (Bivand *et al.*, 2014) permet de lister les formats matriciels que l'on peut importer et exporter dans et depuis R (tab. 2.2).

```
head(rgdal::getGDALDriverNames())
##      name                      long_name create  copy
## 1 AAIGrid           Arc/Info ASCII Grid  FALSE  TRUE
## 2    ACE2                           ACE2  FALSE FALSE
## 3    ADRG ARC Digitized Raster Graphics   TRUE FALSE
## 4     AIG          Arc/Info Binary Grid  FALSE FALSE
## 5  AirSAR     AirSAR Polarimetric Image  FALSE FALSE
## 6     ARG     Azavea Raster Grid format  FALSE  TRUE
```

TABLEAU 2.2 – Liste des formats de fichiers matriciels importables et exportables avec les pilotes GDAL du package **rgdal**.

Format	Écriture	Copie	Format	Écriture	Copie	Format	Écriture	Copie
AAIGrid	non	oui	GRIB	non	non	NITF	oui	oui
ACE2	non	non	GS7BG	oui	oui	NTv2	oui	non
ADRG	oui	non	GSAG	non	oui	NWT_GRC	non	non
AIG	non	non	GSBG	oui	oui	NWT_GRD	non	non
AirSAR	non	non	GSC	non	non	OGDI	non	non
ARG	non	oui	GTiff	oui	oui	OZI	non	non
BAG	non	non	GTX	oui	non	PAux	oui	non
BIGGIF	non	non	GXF	non	non	PCIDSK	oui	non
BLX	non	oui	HDF4	non	non	PCRaster	non	oui
BMP	oui	non	HDF4Image	oui	non	PDF	non	oui
BSB	non	non	HDF5	non	non	PDS	non	non
BT	oui	non	HDF5Image	non	non	PNG	non	oui
CEOS	non	non	HF2	non	oui	PNM	oui	non
COASP	non	non	HFA	oui	oui	PostGISRaster	non	oui
COSAR	non	non	HTTP	non	non	R	non	oui
CPG	non	non	IDA	oui	non	Rasterlite	non	oui
CTable2	oui	non	ILWIS	oui	oui	RIK	non	non
CTG	non	non	INGR	oui	oui	RMF	oui	non
DIMAP	non	non	IRIS	non	non	RPFTOC	non	non
DIPEx	non	non	ISIS2	oui	non	RS2	non	non
DODS	non	non	ISIS3	non	non	RST	oui	oui
DOQ1	non	non	JAXAPALSAR	non	non	SAGA	oui	oui
DOQ2	non	non	JDEM	non	non	SAR_CEOS	non	non
DTED	non	oui	JPEG	non	oui	SDTS	non	non
E00GRID	non	non	JPEG2000	non	oui	SGI	oui	non
ECRGTOC	non	non	KMLSUPEROVERLAY	non	oui	SNODAS	non	non
EHdr	oui	oui	KRO	oui	non	SRP	non	non
EIR	non	non	L1B	non	non	SRTMHGT	non	oui
ELAS	oui	non	LAN	oui	non	Terragen	oui	non
ENVI	oui	non	LCP	non	oui	TIL	non	non
EPSILON	non	oui	Leveller	oui	non	TSX	non	non
ERS	oui	non	LOSLAS	non	non	USGSDEM	non	oui
ESAT	non	non	MAP	non	non	VRT	oui	oui
FAST	non	non	MBTiles	non	non	WCS	non	non
FIT	non	oui	MEM	oui	non	WEBP	non	oui
FujiBAS	non	non	MFF	oui	oui	WMS	non	oui
GenBin	non	non	MFF2	oui	oui	XPM	non	oui
GFF	non	non	MSGN	non	non	XYZ	non	oui
GIF	non	oui	NDF	non	non	ZMap	non	oui
GMT	non	oui	netCDF	oui	oui			
GRASSASCIIGrid	non	non	NGSGEOID	non	non			

2.3.1 Package rgdal

Le package **rgdal** (Bivand *et al.*, 2014) implémente les outils GDAL dans R. Il permet donc de lire et d'écrire de nombreux formats de fichiers spatiaux de données matricielles.

```
library(rgdal)
```

2.3.1.1 Import

Pour importer un raster, une première solution consiste en l'utilisation de la fonction **readGDAL()**. L'objet créé est alors de la classe **SpatialGridDataFrame**. Ici, on renseigne le chemin complet avec l'extension du fichier.

```
FR_MNT1000_L93 <- readGDAL(fname = "../00_data/raster/MNT1000_L93_FRANCE.ASC", p4s = "+init=epsg:2154")
## ../00_data/raster/MNT1000_L93_FRANCE.ASC has GDAL driver AAIGrid
## and has 1081 rows and 1161 columns
class(FR_MNT1000_L93)
## [1] "SpatialGridDataFrame"
## attr(,"package")
## [1] "sp"
mode(FR_MNT1000_L93)
## [1] "S4"
summary(FR_MNT1000_L93)
## Object of class SpatialGridDataFrame
## Coordinates:
##       min     max
## x   89500 1250500
## y 6039500 7120500
## Is projected: TRUE
## proj4string :
## [+init=epsg:2154 +proj=lcc +lat_1=49 +lat_2=44 +lat_0=46.5 +lon_0=3 +x_0=700000
```

```
## +y_0=6600000 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs]
## Grid attributes:
##   cellcentre.offset cellsize cells.dim
## x             90000     1000      1161
## y           6040000     1000      1081
## Data attributes:
##      band1
##  Min.   :   0.0
##  1st Qu.:   0.0
##  Median :  78.0
##  Mean   : 209.7
##  3rd Qu.: 240.0
##  Max.   :4736.0
##  NA's   :346116
```

2.3.1.2 Export

L'export des objets **SpatialGridDataFrame** se fait au moyen de la fonction **writeGDAL()**, et ce, qu'il y ait une seule ou plusieurs bandes. On peut choisir d'écrire :

- des entiers 16 bits signés **Int16** $[-(2^{15})$ à $2^{15}-1]$ ou non signés **UInt16** $[0$ à $2^{16}-1]$;
- des entiers 32 bits signés **Int32** $[-(2^{31})$ à $2^{31}-1]$ ou non signés **UInt32** $[0$ à $2^{32}-1]$;
- des flottants 32 bits (simple précision) **Float32** [1 bit pour le signe, 8 bits pour l'exposant et 23 bits pour la mantisse] ;
- des flottants 64 bits (double précision) **Float64** [1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse].

```
writeGDAL(FR_MNT1000_L93, fname = "FR_MNT1000_L93_Uint16.tif",
          drivername = "GTiff", type = "UInt16", mvFlag = 2L)
```

2.3.2 Package raster

Le package **raster** (Hijmans, 2015) utilise les pilotes proposés par le package **rgdal** (Bivand *et al.*, 2014) pour lire et d'écrire les fichiers spatiaux de données matricielles.

```
library(raster)
```

2.3.2.1 Import

Pour importer un raster, on peut également utiliser la fonction **raster()** du package **raster**. L'objet créé est de la classe **RasterLayer**.

```
FR_MNT1000_L93 <- raster("../00_data/raster/MNT1000_L93_FRANCE.ASC")
mode(FR_MNT1000_L93)
## [1] "S4"
class(FR_MNT1000_L93)
## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"
FR_MNT1000_L93
## class       : RasterLayer
## dimensions  : 1081, 1161, 1255041  (nrow, ncol, ncell)
## resolution  : 1000, 1000  (x, y)
## extent      : 89500, 1250500, 6039500, 7120500  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## data source : /home/irstea/Samba/smbe/Data/boulot/misc/scientific_doc/R/formation/irstea/033-geomatique_LIV/01_r
## names       : MNT1000_L93_FRANCE
## values      : -2147483648, 2147483647  (min, max)
```

Pour les rasters à plusieurs bandes, on utilisera les fonctions **stack()** ou **brick()**.

2.3.2.2 Export

Avec le package **raster**, l'export se fait au moyen de la fonction **writeRaster()**, que ce soit pour des objets de classe **RasterLayer**, **RasterBrick** ou **RasterStack**. On peut choisir d'écrire :

- des booléens **LOG1S** ;
- des entiers 8 bits signés **INT1S** $[-(2^{7})$ à $2^{7}-1]$ ou non signés **INT1U** $[0$ à $2^{8}-1]$;
- des entiers 16 bits signés **INT2S** $[-(2^{15})$ à $2^{15}-1]$ ou non signés **INT2U** $[0$ à $2^{16}-1]$;
- des entiers 32 bits signés **INT4S** $[-(2^{31})$ à $2^{31}-1]$ ou non signés **INT4U** $[0$ à $2^{32}-1]$;
- des flottants 32 bits (simple précision) **FLT4S** [1 bit pour le signe, 8 bits pour l'exposant et 23 bits pour la mantisse] ;
- des flottants 64 bits (double précision) **FLT8S** [1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse].

```
writeRaster(FR_MNT1000_L93, filename = "FR_MNT1000_L93_Uint16b.tif",
            format = "GTiff", datatype = "INT2S")
```

Chapitre 3

Gestion du système de coordonnées

3.1 Classe sp

Pour afficher ou définir (si cela n'était pas déjà fait) le système de coordonnées, on utilise la fonction **proj4string()**, du package **sp**(Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a).

Ici, on dispose d'un objet correspondant au contour de la France, qui ne présente pas de système de référence.

```
sp::proj4string(FR_ctr_L2E)
## [1] NA
```

Avec la fonction **proj4string()**, on le définit *a posteriori*, comme étant de Lambert 2 étendu (code EPSG 27572), et l'on effectue la vérification avec la même fonction.

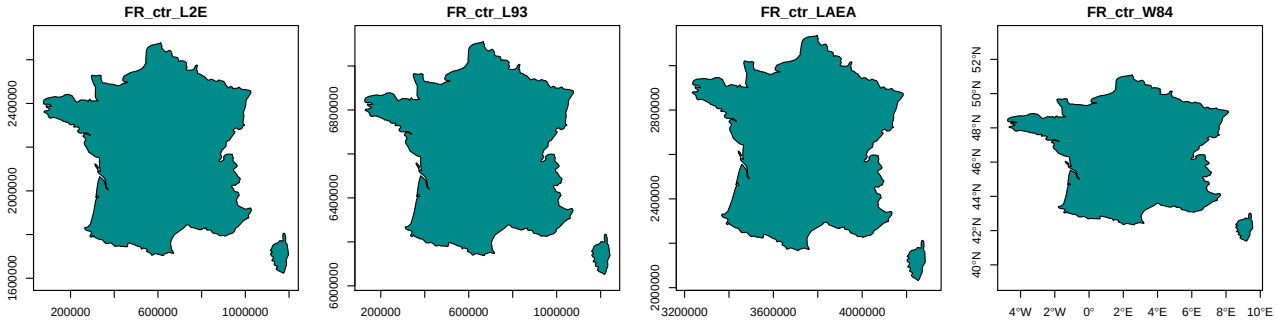
```
sp::proj4string(FR_ctr_L2E) <- CRS("+init=epsg:27572")
sp::proj4string(FR_ctr_L2E)
## [1] "+init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_0=2200000 +a=637
```

Pour modifier le système de coordonnées, on utilise la fonction **spTransform()** du package **rgdal**(Bivand *et al.*, 2014). Dans l'exemple, on transforme le contour de France du Lambert 2 étendu vers le Lambert 93 (code EPSG 2154), le LAEA (code EPSG 3035) et le WGS 84 (code EPSG 4326).

```
FR_ctr_L93  <- rgdal::spTransform(FR_ctr_L2E, CRS("+init=epsg:2154"))
FR_ctr_LAEA <- rgdal::spTransform(FR_ctr_L2E, CRS("+init=epsg:3035"))
FR_ctr_W84  <- rgdal::spTransform(FR_ctr_L2E, CRS("+init=epsg:4326"))
```

```
sp::proj4string(FR_ctr_L93)
## [1] "+init=epsg:2154 +proj=lcc +lat_1=49 +lat_2=44 +lat_0=46.5 +lon_0=3 +x_0=700000 +y_0=6600000 +ellps=GRS80 +t
sp::proj4string(FR_ctr_LAEA)
## [1] "+init=epsg:3035 +proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +towgs84=0,0,0,0,0,0,
sp::proj4string(FR_ctr_W84)
## [1] "+init=epsg:4326 +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

```
par(mfrow = c(1, 4))
for(layer in c("FR_ctr_L2E", "FR_ctr_L93", "FR_ctr_LAEA", "FR_ctr_W84")) {
        plot(get(layer), col = "cyan4", axes = TRUE, main = layer, asp = 1)
}
```



3.2 Classe raster

Pour afficher ou définir (si cela n'était pas déjà fait) le système de coordonnées, on utilise la fonction **projection()** du package **raster** (Hijmans, 2015) (cela fonctionne également avec la fonction **proj4string()**).

On dispose, ici, d'un MNT de la France dont le système de référence n'est pas défini.

```
raster::projection(FR_MNT1000_L93)
## [1] NA
```

À l'aide de la fonction **projection()**, on lui en assigne donc le Lambert 93 comme système de coordonnées, et l'on vérifie que tout s'est bien passé avec la même fonction.

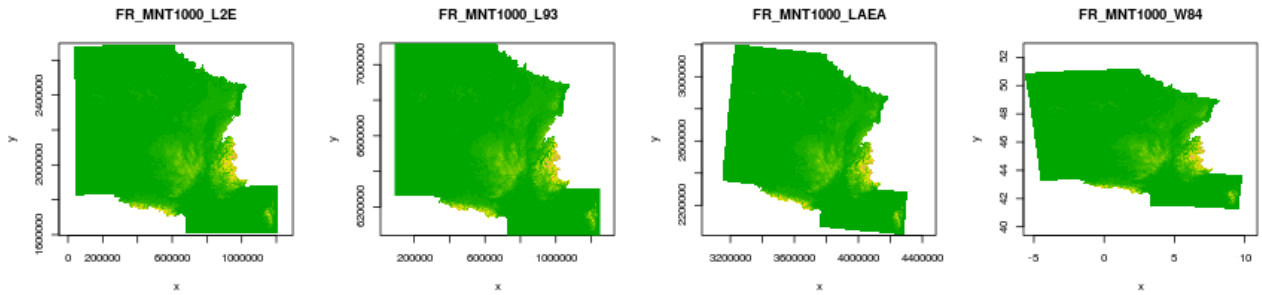
```
raster::projection(FR_MNT1000_L93) <- CRS("+init=epsg:2154")
raster::projection(FR_MNT1000_L93)
## [1] "+init=epsg:2154 +proj=lcc +lat_1=49 +lat_2=44 +lat_0=46.5 +lon_0=3 +x_0=700000 +y_0=6600000 +ellps=GRS80 +t
```

Pour modifier le système de coordonnées, on utilise la fonction **projectRaster()** du package **raster**. À partir de du MNT initial en Lambert 93, on crée donc trois objets, en transformant le système de coordonnées en Lambert 2 étendu, en LAEA et en WGS 84.

```
FR_MNT1000_L2E  <- raster::projectRaster(FR_MNT1000_L93, crs = "+init=epsg:27572")
FR_MNT1000_LAEA <- raster::projectRaster(FR_MNT1000_L93, crs = "+init=epsg:3035")
FR_MNT1000_W84  <- raster::projectRaster(FR_MNT1000_L93, crs = "+init=epsg:4326")
```

```
raster::projection(FR_MNT1000_L2E)
## [1] "+init=epsg:27572 +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=600000 +y_0=2200000 +a=637
raster::projection(FR_MNT1000_LAEA)
## [1] "+init=epsg:3035 +proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +towgs84=0,0,0,0,0,0,
raster::projection(FR_MNT1000_W84)
## [1] "+init=epsg:4326 +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

```
par(mfrow = c(1, 4))
for(layer in c("FR_MNT1000_L2E", "FR_MNT1000_L93", "FR_MNT1000_LAEA", "FR_MNT1000_W84")) {
        image(get(layer), col = terrain.colors(20), main = layer, asp = 1)
}
```



3.3 Divers

3.3.1 Comparer des systèmes de coordonnées

Il est possible de tester l'égalité des systèmes de coordonnées entre deux objet.

```
proj4string(FR_MNT1000_LAEA)
## [1] "+init=epsg:3035 +proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +towgs84=0,0,0,0,0,0,
proj4string(FR_ctr_LAEA)
## [1] "+init=epsg:3035 +proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +towgs84=0,0,0,0,0,0,
```

Pour cela, on peut utiliser la fonction **identicalCRS()** du package **sp**, qui prend en argument les deux objets que l'on souhaite comparer.

```
sp::identicalCRS(FR_MNT1000_LAEA, FR_ctr_LAEA)
## [1] TRUE
```

Il est également possible d'utiliser la fonction **compareCRS()** du package **raster**, qui compare directement les systèmes de coordonnées des objets.

```
raster::compareCRS(proj4string(FR_MNT1000_LAEA), proj4string(FR_ctr_LAEA))
## [1] TRUE
```

3.3.2 Convertir le système de coordonnées dans une matrice, depuis et vers le WGS 84

On peut convertir des coordonnées exprimées en WGS 84, et contenues dans un objet **matrix**, vers un autre système (ou inversement ; on choisit grâce à l'argument **inv**), sans passer par un objet de classe **sp** ; c'est ce que permet la fonction **project()** du package **rgdal**. La matrice d'entrée ne doit comporter que deux colonnes, la première correspondant aux longitudes et le seconde aux latitudes.

Dans l'exemple suivant, on convertit des coordonnées exprimées en Lambert 2 étendu (code EPSG 27572) vers le WGS 84 (code EPSG 4326) ; l'argument **inv** est donc défini à la valeur **TRUE**. L'objet initial étant un **data.frame**, on force sa conversion en objet de classe **matrix**.

```
head(Pt_tab[, c("X", "Y")])
##        X       Y
## 1 281138 1823492
## 2  78629 2413857
## 3 559159 2657554
## 4 971042 2472207
## 5 957312 1809762
## 6 665562 1703359
head(rgdal::project(xy = as.matrix(Pt_tab[, c("X", "Y")]), proj = "+init=epsg:27572", inv = TRUE))
##               X        Y
## [1,] -3.9272879 43.34201
## [2,] -7.0635488 48.51256
## [3,] -0.5791948 50.91000
## [4,]  5.0847098 49.13979
## [5,]  4.3904901 43.20053
## [6,]  0.7932000 42.33158
```

3.3.3 Passer facilement des degrés, minutes, secondes aux degrés décimaux

On peut aisément passer de coordonnées exprimées en degrés, minutes, secondes (contenus dans un vecteur de chaînes de caractères) vers des coordonnées en degrés décimaux grâce à la fonction **char2dms()** du package **sp**. Pour cela, il suffit de spécifier à cette fonction quels sont les symboles des degrés (**chd**), des minutes (**chm**) et des secondes (**chs**).

```
point_X_CHA <- c("4°31'00 E", "2°15'42 E")
point_X_CHA
## [1] "4°31'00 E" "2°15'42 E"
point_X_DMS  <- sp::char2dms(point_X_CHA, chd = "°", chm = "'", chs = " ")
point_X_DMS
## [1] 4d31'E    2d15'42"E
```

Le résultat renvoyé par la fonction **char2dms()** est exprimé dans la classe **DMS**.

```
class(point_X_DMS)
## [1] "DMS"
## attr(,"package")
## [1] "sp"
```

Pour récupérer un résultat exploitable, il suffit d'effectuer une conversion en **numeric**.

```
point_X_DD <- as.numeric(point_X_DMS)
point_X_DD
## [1] 4.516667 2.261667
```

La conversion des degrés décimaux vers les degrés, minutes, secondes se fait grâce à la fonction **dd2dms()** du package **sp**. L'argument **NS** (Nord-Sud) permet de préciser si les coordonnées sont des longitudes (**NS = FALSE**) ou des latitudes (**NS = TRUE**).

```
point_X_DMS <- sp::dd2dms(point_X_DD, NS = FALSE)
point_X_DMS
## [1] 4d31'E    2d15'42"E
```

Le résultat renvoyé par la fonction **dd2dms()** est exprimé dans la classe **DMS**.

```
class(point_X_DMS)
## [1] "DMS"
## attr(,"package")
## [1] "sp"
```

On retrouve une chaîne de caractères en passant par la fonction **as.character()**.

```
point_X_CHA <- as.character(point_X_DMS)
point_X_CHA
## [1] "4d31'E"     "2d15'42\"E"
```

Troisième partie

Géotraitements

Chapitre 4

Traitements internes à R

Dans cette partie, sont présentées quelques possibilités de traitements géomatiques sous R. Seuls quelques packages pouvant être très utiles seront présentés. Il ne s'agit pas de présenter ici toutes les fonctionnalités de ces packages, mais seulement les plus courantes. L'approfondissement de la connaissance des packages et de leurs fonctions est laissée aux bons soins des utilisateurs.

Rappelons ici que pour pouvoir effectuer des traitements spatiaux, toutes les couches spatiales utilisées doivent obligatoirement être dans le même système de coordonnées. Avant tout géotraitement impliquant plusieurs objets spatiaux, il convient donc de vérifier s'ils présentent le même système de coordonnées (§ 3.3.1). Pour alléger le code, nous ne ferons néanmoins pas le test dans les exemples ci-après.

4.1 Package `raster`

Comme son nom l'indique, le package **raster** (Hijmans, 2015) est spécialisé dans les traitement de données spatiales matricielles. C'est le package à privilégier pour manipuler ce format de données.

```
library(raster)
```

4.1.1 Calculs sur un raster

Il est possible de réaliser des opérations arithmétiques sur des rasters (et entre rasters), au moyen de simples opérateurs arithmétiques.

```
summary(values(FR_MNT1000_LAEA))
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##     0.0     0.0    79.4   210.6   241.0  4699.0  602970
summary(values(FR_MNT1000_LAEA + 1e4))
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   10000   10000   10080   10210   10240   14700  602970
summary(values(FR_MNT1000_LAEA + FR_MNT1000_LAEA))
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##     0.0     0.0   158.9   421.3   481.9  9397.0  602970
```

On peut aussi utiliser les fonctions **calc()** et **overlay()**.

```
summary(calc(FR_MNT1000_LAEA, fun = function(x) x * 2))
##                 layer
## Min.           0.0000
## 1st Qu.        0.0000
## Median       158.8712
## 3rd Qu.      481.9441
## Max.        9397.0485
## NA's      602970.0000
summary(overlay(FR_MNT1000_LAEA, FR_MNT1000_LAEA, fun = function(x, y) x + y))
##                 layer
## Min.           0.0000
## 1st Qu.        0.0000
## Median       158.8712
## 3rd Qu.      481.9441
## Max.        9397.0485
## NA's      602970.0000
```

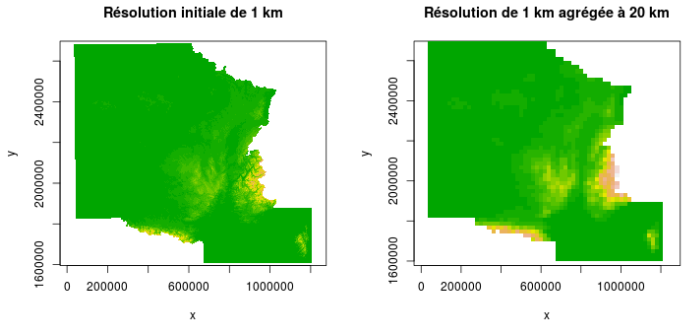
4.1.2 Modifier la résolution d'un raster

Agréger les valeurs d'un raster. Le package **raster** permet de réduire la résolution d'un raster à l'aide de la fonction **aggregate()** ; pour cela, il faut jouer avec l'argument **fact**, qui s'exprime dans l'unité de la

résolution du raster. Dans l'exemple suivant, on transforme un raster d'1 km de résolution en un raster de 20 km de résolution.

```
FR_MNT20000_L2E <- aggregate(FR_MNT1000_L2E, fact = 20, fun = mean)
```

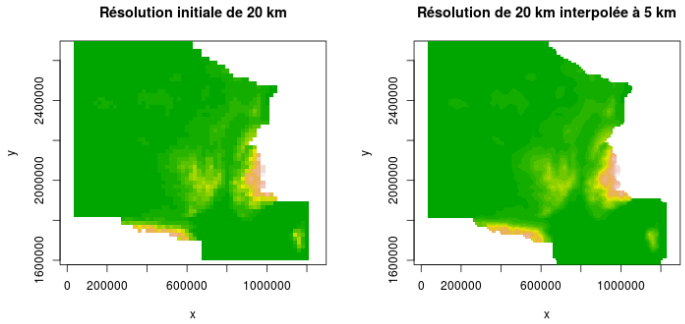
```
par(mfrow = c(1, 2))
image(FR_MNT1000_L2E, col = terrain.colors(20),
        main = "Résolution initiale de 1 km", asp = 1)
image(FR_MNT20000_L2E, col = terrain.colors(20),
        main = "Résolution de 1 km agrégée à 20 km", asp = 1)
```



Interpoler les valeurs d'un raster. On peut également augmenter la résolution d'un raster en interpolant les valeurs à l'aide de la fonction **`disaggregate()`**. Plusieurs méthodes d'interpolation sont disponible ; dans notre cas, on choisit la méthode bilinéaire. Dans l'exemple suivant, on interpole un raster de 20 km de résolution en un raster de 5 km de résolution.

```
FR_MNT20000_L2E_inter <- disaggregate(FR_MNT20000_L2E, fact = c(5, 5), method = "bilinear")
```

```
par(mfrow = c(1, 2))
image(FR_MNT20000_L2E, col = terrain.colors(20),
      main = "Résolution initiale de 20 km", asp = 1)
image(FR_MNT20000_L2E_inter, col = terrain.colors(20),
      main = "Résolution de 20 km interpolée à 5 km", asp = 1)
```



4.1.3 Intersecter une couche matricielle par une couche vectorielle

Il est possible de découper des rasters par l'emprise d'un polygone. Ici, on souhaite découper le MNT de la France par le polygone du bassin versant de la Seine.

On peut réaliser cette opération grâce à la fonction **`mask()`**. Notez bien que l'emprise du raster n'est pas modifiée, mais les valeurs dépassant le contour du polygone valent à présent **`NA`**.

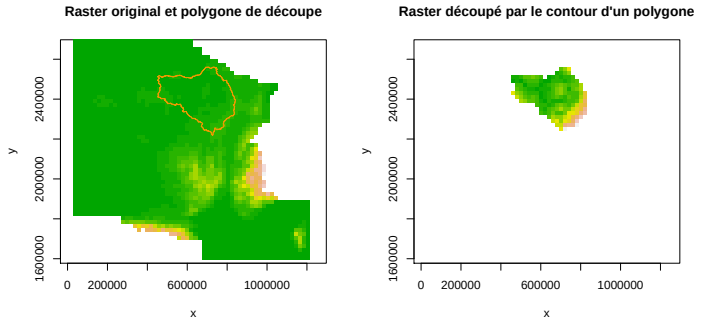
```
FR_MNT20000_L2E_clip_SE <- mask(FR_MNT20000_L2E, mask = SE_ctr_L2E)
summary(FR_MNT20000_L2E_clip_SE)
##          MNT1000_L93_FRANCE
## Min.               59.91534
## 1st Qu.           109.59334
## Median            142.12657
## 3rd Qu.           201.77174
## Max.              482.29238
## NA's             3172.00000
```

On peut également utiliser la fonction **`rasterize()`** en utilisant l'argument **`mask = TRUE`**. Attention, l'ordre des arguments de la fonction **`rasterize()`** n'est pas le même que celui de la fonction **`mask()`**.

```
FR_MNT20000_L2E_clip_SE <- rasterize(SE_ctr_L2E, FR_MNT20000_L2E, mask = TRUE)
summary(FR_MNT20000_L2E_clip_SE)
##               layer
## Min.       59.91534
## 1st Qu.   109.59334
## Median    142.12657
## 3rd Qu.   201.77174
```

```
## Max.     482.29238
## NA's    3172.00000
```

```
par(mfrow = c(1, 2))
image(FR_MNT20000_L2E, col = terrain.colors(20), asp = 1,
      main = "Raster original et polygone de découpe")
plot(SE_ctr_L2E, col = NA, border = "orange2", add = TRUE)
image(FR_MNT20000_L2E_clip_SE, col = terrain.colors(20), asp = 1,
      main = "Raster découpé par le contour d'un polygone")
```

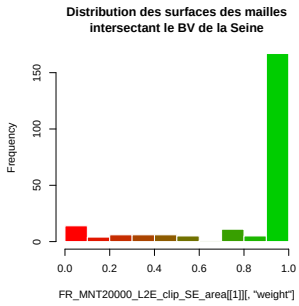


Par ailleurs, la fonction **extract()**, permet d'extraire, par exemple, le numéro des mailles (**cellnumbers**), leurs valeurs (**weights**) ou la surface (ou la proportion) couverte par la couche vectorielle (**normalizeWeights**). L'objet renvoyé est une liste.

```
FR_MNT20000_L2E_clip_SE_area<- extract(FR_MNT20000_L2E, SE_ctr_L2E,
                                       cellnumbers = TRUE, weights = TRUE, normalizeWeights = FALSE)
head(FR_MNT20000_L2E_clip_SE_area[[1]])
##      cell    value weight
## [1,]  394 162.8203   0.10
## [2,]  395 196.1143   0.02
## [3,]  396 208.3554   0.05
## [4,]  453 122.2771   0.40
## [5,]  454 159.5542   1.00
## [6,]  455 197.4723   0.93
```

Dans notre exemple, on a récupéré la proportion de chaque maille recoupant le bassin versant de la Seine. Logiquement, la plupart des valeurs sont égales à 1, seuls les pixels marginaux présentent des valeurs inférieures.

```
area_brk <- seq(0, 1, by = 0.1)
area_col <- colorRampPalette(c("red", "green3"))(10)
hist(FR_MNT20000_L2E_clip_SE_area[[1]][, "weight"],
     breaks = area_brk,
     main = "Distribution des surfaces des mailles\nintersectant le BV de la Seine",
     col = area_col, border = "white")
```

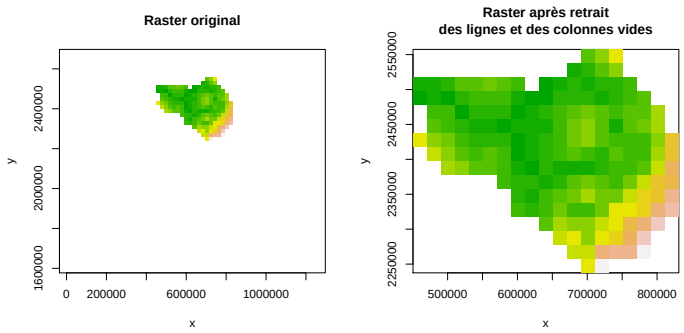


4.1.4 Retirer les valeurs vides d'un raster

Pour nettoyer un raster des ses lignes et colonnes comportant uniquement des pixels de mêmes valeurs (e.g. **NA**), on peut utiliser la fonction **trim()**. Cette fonction propose un argument **value** définissant explicitement les valeurs que l'on souhaite retirer (**NA**, par défaut). L'argument **padding**, quant à lui, correspond au nombre de lignes et de colonnes extérieures à conserver (**0**, par défaut).

```
FR_MNT20000_L2E_clip_SE_trim <- trim(FR_MNT20000_L2E_clip_SE, padding = 0, values = NA)
summary(FR_MNT20000_L2E_clip_SE_trim)
##             layer
## Min.     59.91534
## 1st Qu. 109.59334
## Median  142.12657
## 3rd Qu. 201.77174
## Max.    482.29238
## NA's    116.00000
```

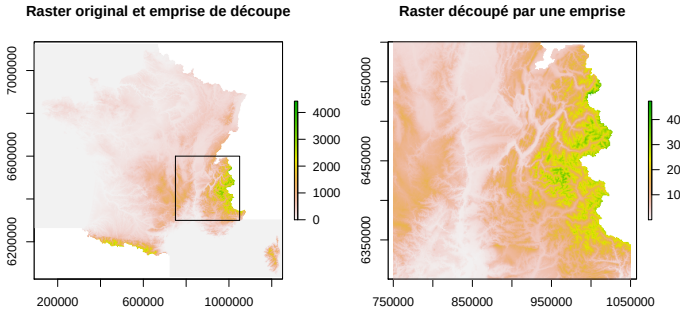
```
par(mfrow = c(1, 2))
image(FR_MNT20000_L2E_clip_SE, col = terrain.colors(20), asp = 1,
      main = "Raster original")
image(FR_MNT20000_L2E_clip_SE_trim, col = terrain.colors(20), asp = 1,
      main = "Raster après retrait\ndes lignes et des colonnes vides")
```



4.1.5 Découper un raster par une emprise géographique

À l'aide de la fonction **crop()**, on peut découper un raster selon une emprise géographique définie par la fonction **extent()**.

```
EXT_sub <- extent(7.5e5, 10.5e5, 6.3e6, 6.6e6)
FR_MNT1000_L93_clip_EXT <- crop(FR_MNT1000_L93, EXT_sub)
par(mfrow = c(1, 2))
plot(FR_MNT1000_L93, main = "Raster original et emprise de découpe")
plot(EXT_sub, add = TRUE)
plot(FR_MNT1000_L93_clip_EXT, main = "Raster découpé par une emprise")
```



4.1.6 Assembler des rasters

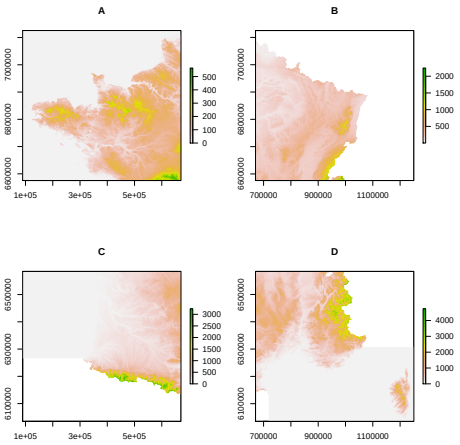
Pour assembler des rasters, on peut utiliser deux fonctions différentes : **merge()** ou **mosaic()**. Les objets à assembler doivent impérativement avoir la même origine, la même résolution, ainsi que le même système de coordonnées.

Dans l'exemple suivant, on commence par découper un raster en quatre parties selon des emprises géographiques.

```
EXT_subA <- extent( 89500,  670000, 6039500, 6580000)
EXT_subB <- extent( 89500,  670000, 6580000, 7120500)
EXT_subC <- extent(670000, 1250500, 6039500, 6580000)
EXT_subD <- extent(670000, 1250500, 6580000, 7120500)
FR_MNT1000_L93_clip_EXT_A <- crop(FR_MNT1000_L93, EXT_subA)
FR_MNT1000_L93_clip_EXT_B <- crop(FR_MNT1000_L93, EXT_subB)
FR_MNT1000_L93_clip_EXT_C <- crop(FR_MNT1000_L93, EXT_subC)
FR_MNT1000_L93_clip_EXT_D <- crop(FR_MNT1000_L93, EXT_subD)
```

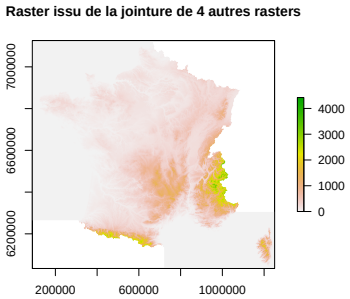
Graphiquement, on obtient :

```
par(mfrow = c(2, 2), oma = c(0, 0, 0, 1))
plot(FR_MNT1000_L93_clip_EXT_B, main = "A")
plot(FR_MNT1000_L93_clip_EXT_D, main = "B")
plot(FR_MNT1000_L93_clip_EXT_A, main = "C")
plot(FR_MNT1000_L93_clip_EXT_C, main = "D")
```



On assemble alors les quatre sous-rasters créés en un seul à l'aide la fonction **merge()**.

```
FR_MNT1000_L93_merge <- merge(FR_MNT1000_L93_clip_EXT_A, FR_MNT1000_L93_clip_EXT_B,
                              FR_MNT1000_L93_clip_EXT_C, FR_MNT1000_L93_clip_EXT_D)
plot(FR_MNT1000_L93_merge, main = "Raster issu de la jointure de 4 autres rasters")
```

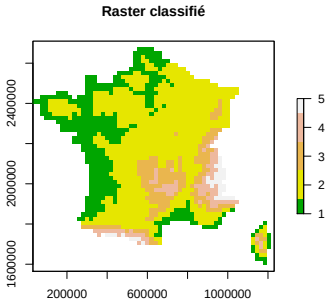


La différence entre les fonctions **merge()** et **mosaic()** réside dans leur comportement pour traiter les pixels qui se chevauchent. La première fonction prend simplement la valeur du premier raster donné. La seconde permet de faire la moyenne des pixels ou de définir soi-même la fonction de calcul à appliquer en cas de chevauchement de pixels.

4.1.7 Classifier un raster

On part ici du MNT de la France que l'on découpe en 5 classes d'altitudes à l'aide de la fonction **cut()**. Les bornes des 5 classes sont définies dans l'argument **breaks**.

```
myseq <- c(0, 100, 500, 1000, 2000, 5000)
FR_MNT20000_L2E_cut <- cut(FR_MNT20000_L2E, breaks = myseq)
class(FR_MNT20000_L2E_cut)
## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"
summary(FR_MNT20000_L2E_cut)
##         layer
## Min.        1
## 1st Qu.     1
## Median      2
## 3rd Qu.     2
## Max.        5
## NA's     1820
plot(FR_MNT20000_L2E_cut, main = "Raster classifié",
     col = terrain.colors(max(values(FR_MNT20000_L2E_cut), na.rm = TRUE)))
```



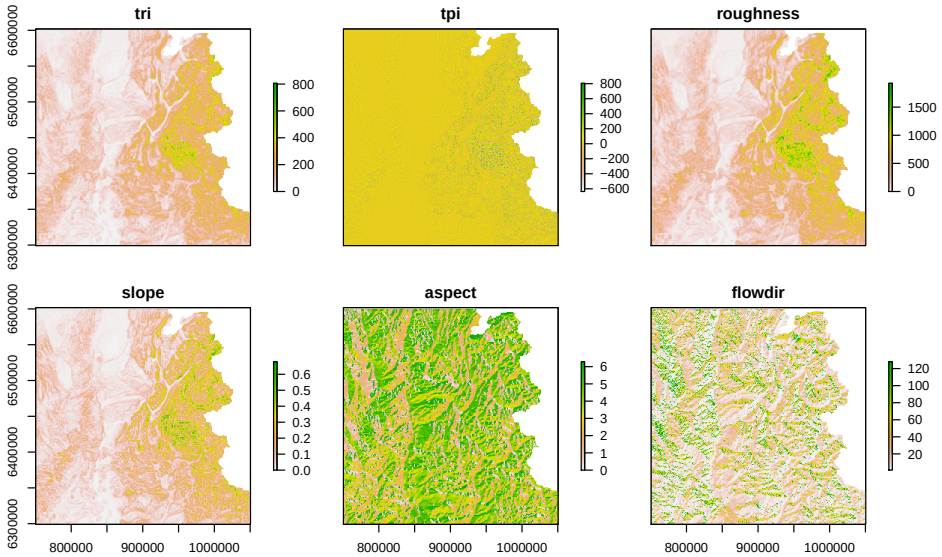
4.1.8 Effectuer des analyses de terrain

Il est possible de calculer la pente, l'aspect et d'autres caractéristiques de terrain à partir d'un raster d'élévation. Les données d'altitude doivent être exprimées dans l'unité du système de coordonnées de la couche (typiquement le mètre) pour des données projetées (planes). Elles doivent être exprimées en mètres lorsque le système de coordonnées de référence est non projeté.

Avec la fonction **`terrain()`**, il est possible de calculer :

- **`slope`** : la pente, dérivée première de l'élévation donnée en degré d'inclinaison [degrés ou radians] (0° indique un terrain plat) ;
- **`aspect`** : l'aspect, qui correspond l'orientation du terrain [degrés ou radians] (0° et 360° pour une face orientée vers le Nord, 90° vers l'Ouest, 180° vers le Sud, 270° vers l'Est) ;
- **`TPI`** : l'indice de position topographique, qui est défini comme la différence entre la valeur de la cellule centrale et la moyenne de celles de ses voisines ;
- **`TRI`** : l'indice de rugosité, qui correspond à la moyenne des dénivelés entre la cellule centrale et ses voisines (l'unité est donc celle du raster), et mesure l'hétérogénéité du terrain ; le résultat obtenu peut sembler proche de celui de la pente, mais la rugosité accentue les ruptures du relief ;
- **`roughness`** : la rugosité, qui correspond à la plus grande différence entre la valeur de la cellule centrale et celles de ses 8 voisines (à la différence de l'indice de rugosité qui est une moyenne de ces 8 différences) ;
- **`flowdir`** : la direction de flux, qui correspond au sens de circulation de chaque cellule, défini comme la direction de l'un de ses huit voisins adjacents (ou diagonaux) présentant la pente descendante raide.

```
TER1000_L93_clip_EXT <- terrain(FR_MNT1000_L93_clip_EXT,
                                opt = c("TRI", "TPI", "roughness", "slope", "aspect", "flowdir"))
par(oma = c(0, 0, 0, 1))
plot(TER1000_L93_clip_EXT)
```

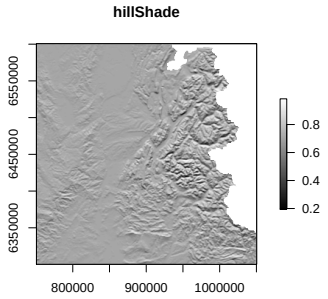


La fonction **`focal()`** permet de définir ses propres calculs de voisinage, et vient donc complémenter les fonctionnalités de **`terrain()`**.

À partir de la pente et de l'aspect, il est possible de calculer l'ombrage, grâce à la fonction **`hillShade()`**. Par défaut, l'aspect des ombres du terrain est supposé éclairé par une source lumineuse située à 0° d'azimut et à

45° d'élévation.

```
par(oma = c(0, 0, 0, 1))
HSH1000_L93_clip_EXT <- hillShade(slope = TER1000_L93_clip_EXT[["slope"]],
                                  aspect = TER1000_L93_clip_EXT[["aspect"]],
                                  angle = 45, direction = 0)
plot(HSH1000_L93_clip_EXT, col = grey(0:100/100), main = "hillShade")
```



4.1.9 Rasteriser des données vectorielles

Il est possible de rasteriser des données vectorielles. Dans l'exemple suivant, on souhaite transformer en raster le réseau hydrographique du bassin versant de l'Orgeval[1].

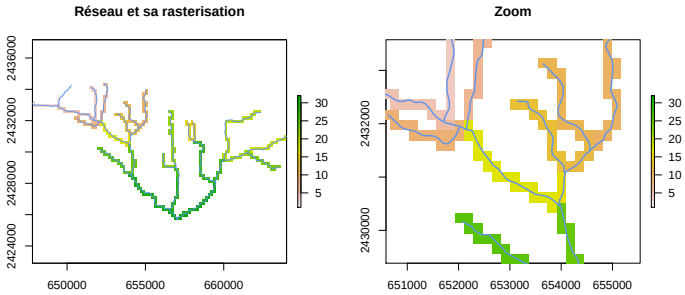
```
OR_RIV_L2E <- readShapeLines("01_data/vecteur/orgeval_reseau_hydro.shp",
                             proj4string = CRS("+init=epsg:27572"))
```

Au préalable, il convient de créer une grille sur laquelle on veut caler le réseau. On doit définir l'emprise de cette grille, le nombre de lignes et de colonnes qu'elle comporte, ainsi que son système de coordonnées (qui doit être le même que celui du réseau que l'on souhaite transformer). Plus le nombre de cellules du raster sera élevé et plus la raserisation sera précise, mais plus le raster créé sera volumineux. Il faut donc trouver un compromis entre la précision que l'on souhaite et la taille de l'objet que l'on crée. Notez qu'il est aussi possible de se servir d'un raster déjà existant où tous ces paramètres sont déjà définis.

```
myRaster <- raster(ncols = 90, nrows = 45,
                   xmn = OR_RIV_L2E@bbox["x", "min"], xmx = OR_RIV_L2E@bbox["x", "max"],
                   ymn = OR_RIV_L2E@bbox["y", "min"], ymx = OR_RIV_L2E@bbox["y", "max"],
                   crs = "+init=epsg:27572")
```

Puis, on réalise l'opération de rasterisation à l'aide de la fonction **rasterize()**.

```
OR_RIV_L2E_rast <- rasterize(OR_RIV_L2E, myRaster)
par(mfrow = c(1, 2))
plot(OR_RIV_L2E_rast, main = "Réseau et sa rasterisation")
plot(OR_RIV_L2E, col = "cornflowerblue", asp = 1, add = TRUE)
plot(0, col = NA, col.lab = NA, xlim = c(650768.3, 655356.5), ylim = c(2429540, 2433420), main = "Zoom")
plot(OR_RIV_L2E_rast, add = TRUE)
plot(OR_RIV_L2E, col = "cornflowerblue", lwd = 2, add = TRUE)
```



4.1.10 Vectoriser un raster

Bien évidemment, il est également possible de réaliser l'opération inverse, c'est-à-dire de transformer un raster en données vectorielles. On part ici du MNT de la France découpé en 5 classes d'altitudes, puis on le transforme en polygones grâce à la fonction **rasterToPolygons()**. On obtient alors un objet de classe

1. Pour plus d'information sur l'observatoire ORACLE et le bassin versant de l'Orgeval : SitewebduGISORACLE:https://gisoracle.cemagref.fr/.

SpatialPolygonsDataFrame. L'argument **dissolve** permet de définir si l'on souhaite fusionner ou non les polygones contigus de mêmes valeurs.

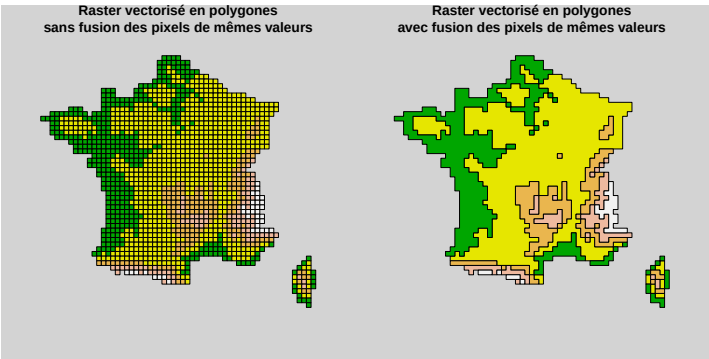
```
FR_MNT20000_L2E_cut_po1 <- rasterToPolygons(FR_MNT20000_L2E_cut, dissolve = FALSE)
FR_MNT20000_L2E_cut_po2 <- rasterToPolygons(FR_MNT20000_L2E_cut, dissolve = TRUE)
class(FR_MNT20000_L2E_cut_po1)
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
class(FR_MNT20000_L2E_cut_po2)
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
```

En cas de fusions des polygones, on constate bien que le **SpatialPolygonsDataFrame** produit présente bien une seule entrée par classe possible dans la table attributaire ; chaque entrée pouvant toutefois être associée à plusieurs sous-polygones, dans le cas ou des entités présentent la même valeur mais ne sont pas contiguës.

```
table(FR_MNT20000_L2E_cut_po1@data)
##
##   1   2   3   4   5
## 405 824 187  88  36
table(FR_MNT20000_L2E_cut_po2@data)
##
## 1 2 3 4 5
## 1 1 1 1 1
```

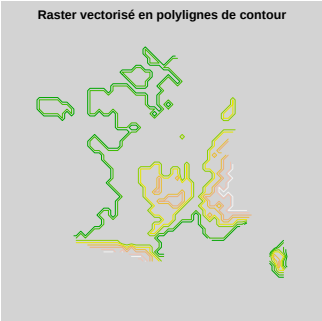
Graphiquement, on obtient la représentation suivante :

```
par(mfrow = c(1, 2), bg = "lightgrey")
plot(FR_MNT20000_L2E_cut_po1,
     col = terrain.colors(max(FR_MNT20000_L2E_cut_po1@data$layer))[FR_MNT20000_L2E_cut_po1@data$layer],
     main = "Raster vectorisé en polygones\nsans fusion des pixels de mêmes valeurs")
plot(FR_MNT20000_L2E_cut_po2,
     col = terrain.colors(max(FR_MNT20000_L2E_cut_po2@data$layer))[FR_MNT20000_L2E_cut_po2@data$layer],
     main = "Raster vectorisé en polygones\navec fusion des pixels de mêmes valeurs")
```



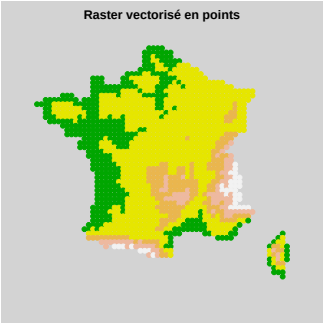
On peut aussi créer un objet de classe **SpatialLinesDataFrame** en utilisant la fonction **rasterToContour()**.

```
FR_MNT20000_L2E_cut_li <- rasterToContour(FR_MNT20000_L2E_cut)
class(FR_MNT20000_L2E_cut_li)
## [1] "SpatialLinesDataFrame"
## attr(,"package")
## [1] "sp"
par(bg = "lightgrey")
plot(FR_MNT20000_L2E_cut_li,
     col = terrain.colors(nlevels(FR_MNT20000_L2E_cut_li@data$level))[FR_MNT20000_L2E_cut_li@data$level],
     main = "Raster vectorisé en polylignes de contour")
```



On peut également récupérer un objet de classe **SpatialPointsDataFrame** grâce à la fonction **rasterToPoints()**. Notez que, par défaut, cette fonction renvoie un objet **matrix**. En revanche, si l'on souhaite récupérer un objet **sp**, il faut utiliser l'argument **spatial = TRUE**.

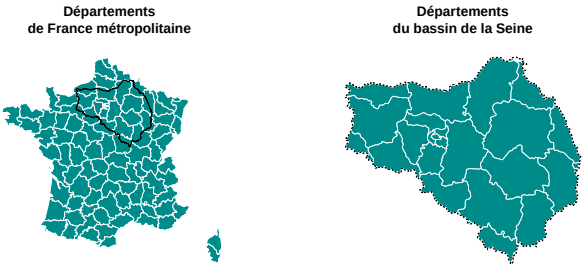
```
FR_MNT20000_L2E_cut_pt <- rasterToPoints(FR_MNT20000_L2E_cut, spatial = TRUE)
class(FR_MNT20000_L2E_cut_pt)
## [1] "SpatialPointsDataFrame"
## attr(,"package")
## [1] "sp"
par(bg = "lightgrey")
plot(FR_MNT20000_L2E_cut_pt, pch = 19,
     col = terrain.colors(max(FR_MNT20000_L2E_cut_pt@data$layer))[FR_MNT20000_L2E_cut_pt@data$layer],
     main = "Raster vectorisé en points")
```



4.1.11 Intersecter des entités vectorielles

Le package **raster** est spécialisé dans le traitement des données matricielles, mais il permet tout de même de réaliser certaines opération purement vectorielles, comme l'intersection entre deux couches différentes. Ceci est rendu possible par la fonction **intersect()**, dépendant des fonctionnalités du package **rgeos** (Bivand & Rundel, 2014) (§ 4.2.3).

```
SE_dpt_L2E <- intersect(FR_dpt_L2E, SE_ctr_L2E)

par(mfrow = c(1, 2))
plot(FR_dpt_L2E, col = "cyan4", border = "white", main = "Départements\nde France métropolitaine")
plot(SE_ctr_L2E, lty = 1, add = TRUE)
plot(SE_dpt_L2E, col = "cyan4", border = "white", main = "Départements\ndu bassin de la Seine")
plot(SE_ctr_L2E, lty = 2, add = TRUE)
```



4.2 Package rgeos

Le package **rgeos** (Bivand & Rundel, 2014) est un des principaux packages d'analyse de données spatiales dans R. Il utilise la bibliothèque GEOS, qui est une interface moteur de l'Open source utilisant l'API C pour les opérations de topologie sur des géométries.

```
library(rgeos)
```

4.2.1 Agréger des entités vectorielles

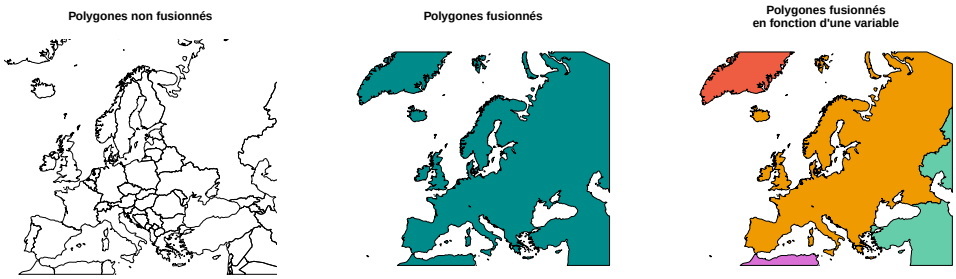
Il est possible d'agréger l'ensemble les entités géographiques d'une couche avec la fonction **gUnaryUnion()**, ou une partie seulement, en utilisant l'argument **id** de cette fonction, ce qui permet de réaliser l'agrégation en fonction d'un critère qualitatif.

On dispose d'une couche géographique des pays d'Europe (en LAEA).

```
head(EU_ctr1_LAEA@data, 4)
##   id iso_a2 iso_a3    name continent            part       area pop pop_dens gdpps gdpps_pop
## 1  1     AL    ALB Albania    Europe Southern Europe 27400.0000  NA       NA    NA        NA
## 2  2     AX    ALA   Aland    Europe Northern Europe   674.3813  NA       NA    NA        NA
## 3  3     AD    AND Andorra    Europe Southern Europe   470.0000  NA       NA    NA        NA
## 4  4     AM    ARM Armenia      Asia            <NA>         NA  NA       NA    NA        NA
##   birth death birth_rate death_rate
## 1    NA    NA         NA         NA
## 2    NA    NA         NA         NA
## 3    NA    NA         NA         NA
## 4    NA    NA         NA         NA
```

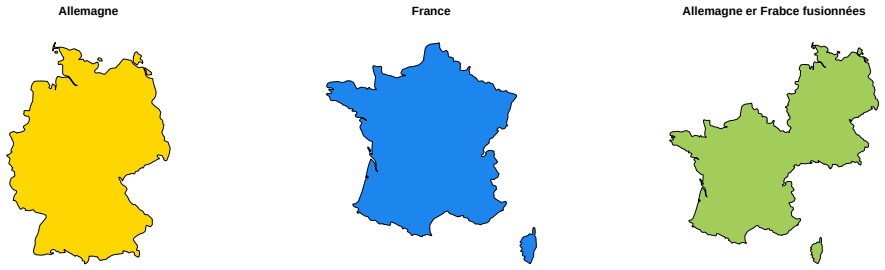
On fusionne tout d'abord l'ensemble des polygones des pays. D'autre part, on fusionne les pays par groupes, selon une variable de la table attributaire.

```
EU_tot_LAEA <- gUnaryUnion(EU_ctr1_LAEA)
EU_uni_LAEA <- gUnaryUnion(EU_ctr1_LAEA, id = EU_ctr1_LAEA@data$continent)
par(mfrow = c(1, 3))
plot(EU_ctr1_LAEA, main = "Polygones non fusionnés")
plot(EU_tot_LAEA, col = "cyan4", main = "Polygones fusionnés")
plot(EU_uni_LAEA, col = c("orchid", "aquamarine3", "orange2", "tomato2"),
     main = "Polygones fusionnés\nen fonction d'une variable")
```



On peut aussi agréger des entités provenant de 2 objets différents avec la fonction **gUnion()**. Dans l'exemple, on dispose de deux couches stockées dans deux objets distincts : un polygone de l'Allemagne et un autre de la France. On fusionne ces deux objets pour former un objet constitué d'un seul polygone représentant une union de ces deux pays.

```
DE_ctrUE_LAEA   <- EU_ctr1_LAEA[EU_ctr1_LAEA@data$name == "Germany", ]
FR_ctrUE_LAEA   <- EU_ctr1_LAEA[EU_ctr1_LAEA@data$name == "France" , ]
DEFR_ctrUE_LAEA <- gUnion(DE_ctrUE_LAEA, FR_ctrUE_LAEA)
par(mfrow = c(1, 3))
plot(DE_ctrUE_LAEA  , col = "gold", main = "Allemagne")
plot(FR_ctrUE_LAEA  , col = "dodgerblue2", main = "France")
plot(DEFR_ctrUE_LAEA, col = "darkolivegreen3", main = "Allemagne er Frabce fusionnées")
```



4.2.2 Différence entre des entités vectorielles

Il est également possible d'effectuer la différence de deux couches à l'aide de la fonction **gDifference()**.

Ici, on soustrait le polygone de la France métropolitaine par le contour du bassin versant de la Seine.

```
FR_L2E_ctr_SE_clip <- gDifference(FR_ctr_L2E, SE_ctr_L2E)
```

On obtient la carte suivante, où la France comporte un trou correspondant au contour du bassin versant de la Seine :

```
plot(FR_ctr_L2E_SE_clip, col = "cyan4", main = "France sans\nle bassin de la Seine")
```

4.2.3 Intersecter des entités vectorielles

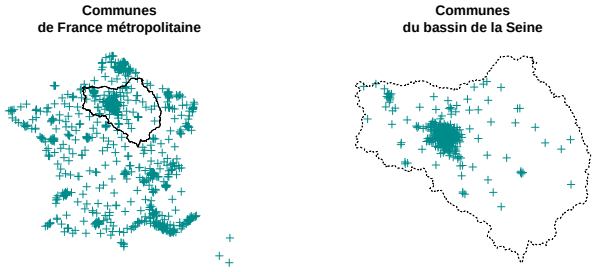
On peut également découper une couche vectorielle (points, lignes ou polygones) par l'emprise d'un polygone. Pour cela, le package **rgeos** propose la fonction **gIntersection()**.

Dans l'exemple suivant, on souhaite sélectionner les positions des principales communes de France situées dans le bassin versant de la Seine.

```
SE_com_L2E <- gIntersection(SE_ctr_L2E, FR_com_L2E)
```

On obtient le résultat suivant :

```
par(mfrow = c(1, 2))
plot(FR_com_L2E, col = "cyan4", main = "Communes\nde France métropolitaine")
plot(SE_ctr_L2E, lty = 1, add = TRUE)
plot(SE_com_L2E, col = "cyan4", main = "Communes\ndu bassin de la Seine")
plot(SE_ctr_L2E, lty = 2, add = TRUE)
```

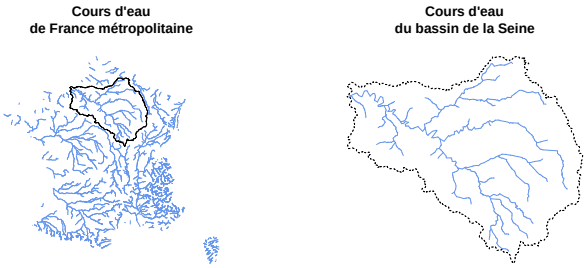


De la même manière, on souhaite sélectionner les cours d'eau de France situés dans le bassin versant de la Seine.

```
SE_riv_L2E <- gIntersection(SE_ctr_L2E, FR_riv_L2E)
```

Ici, si une ligne dépasse du bassin, elle sera tronquée. Par définition, ça ne doit pas être le cas dans l'exemple présenté, puisqu'on s'intéresse aux cours d'eau d'un bassin versant.

```
par(mfrow = c(1, 2))
plot(FR_riv_L2E, col = "cornflowerblue", main = "Cours d'eau\nde France métropolitaine")
plot(SE_ctr_L2E, lty = 1, add = TRUE)
plot(SE_riv_L2E, col = "cornflowerblue", main = "Cours d'eau\ndu bassin de la Seine")
plot(SE_ctr_L2E, lty = 2, add = TRUE)
```



À présent, on souhaite découper une couche des départements de France par le contour du bassin de la Seine. Si un département déborde du bassin, son contour sera découpé pour suivre ce dernier.

En ce qui concerne l'intersection entre deux couches de polygones, c'est un peu plus compliqué qu'avec des points ou des lignes. Il faut tout d'abord tester si les polygones des départements intersectent ou non le bassin versant de la Seine.

```
gI <- gIntersects(FR_dpt_L2E, SE_ctr_L2E, byid = TRUE)
```

Puis, on réalise une boucle sur chacun des départements pour laquelle cette proposition est vraie, et on réalise alors l'intersection avec la fonction **gIntersection()**.

```
SE_dpt_L2E_list <- lapply(which(gI), function(x) {
  gIntersection(FR_dpt_L2E[x, ], SE_ctr_L2E)
  })
table(sapply(SE_dpt_L2E_list, class))
##
## SpatialPolygons
##              28
```

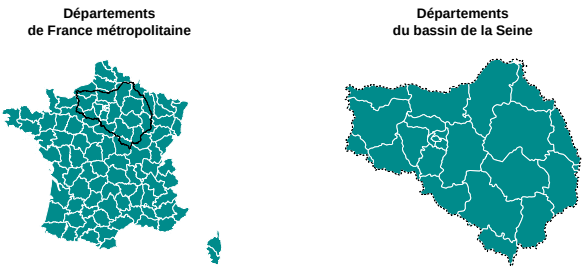
Ensuite, il faut ne conserver que les objets de classe **SpatialPolygons**.

```
keep <- sapply(SE_dpt_L2E_list, class)
SE_dpt_L2E_list <- SE_dpt_L2E_list[keep == "SpatialPolygons"]
```

Enfin, on les regroupe tous ensemble, dans un seul et unique objet **SpatialPolygons**.

```
SE_dpt_L2E <- SpatialPolygons(lapply(seq_along(SE_dpt_L2E_list),
                                     function(x) {
                                       Pol <- slot(SE_dpt_L2E_list[[x]], "polygons")[[1]]
                                       slot(Pol, "ID") <- as.character(x)
                                       Pol
                                     }
                              ))
```

```
par(mfrow = c(1, 2))
plot(FR_dpt_L2E, col = "cyan4", border = "white", main = "Départements\nde France métropolitaine")
plot(SE_ctr_L2E, lty = 1, add = TRUE)
plot(SE_dpt_L2E, col = "cyan4", border = "white", main = "Départements\ndu bassin de la Seine")
plot(SE_ctr_L2E, lty = 2, add = TRUE)
```



4.2.4 Définir une zone tampon

À l'aide de la fonction **gBuffer()**, on peut définir des zones "tampon" autour de points, lignes ou polygones. La distance par rapport à l'entité d'origine s'exprime *via* l'argument **width**, dans l'unité du système de coordonnées. Plusieurs formes de zones "tampon" sont disponibles *via* l'argument **capStyle** :

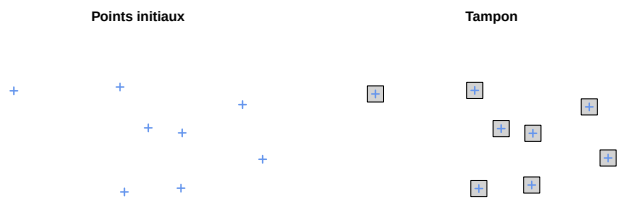
- **"ROUND"** : le contour est arrondi (valeur par défaut) ;
- **"SQUARE"** : le contour est anguleux ;
- **"FLAT"** : le contour est anguleux sans zone "tampon" aux extrémités des lignes (ne fonctionne pas pour les points) .

Si l'on dispose de points, la zone "tampon" se définit comme des objets **SpatialPolygons**, dont les centroïdes sont les coordonnées des points d'origine fournis à la fonction.

```
OR_sta_L2E <- data.frame(X = c( 654948.4,  658227.6,  658300.5,  656332.9,  654693.3,
                                661798.3,  662964.2,  648535.7),
                         Y = c(2426805.0, 2427023.0, 2430230.0, 2430521.0, 2432890.0,
                               2431869.0, 2428699.0, 2432671.0))
coordinates(OR_sta_L2E) <- ~ X + Y
proj4string(OR_sta_L2E) <- CRS("+init=epsg:27572")
OR_sta_L2E_buf <- gBuffer(OR_sta_L2E, width = 500, capStyle = "SQUARE")
```

Graphiquement, on obtient la représentation suivante :

```
par(mfrow = c(1, 2))
plot(OR_sta_L2E, col = "cornflowerblue", lwd = 2, main = "Points initiaux")
plot(OR_sta_L2E_buf, col = "lightgrey", main = "Tampon")
plot(OR_sta_L2E    , col = "cornflowerblue", lwd = 2, add = TRUE)
```



Il est évidemment possible de réaliser la même opération sur des polylignes.

```
OR_RIV_L2E_buf <- gBuffer(OR_RIV_L2E, width = 500, capStyle = "FLAT")
```

Dans ce cas, le "tampon" correspond à une zone de part et d'autre des lignes fournies à la fonction **gBuffer()**.

```
par(mfrow = c(1, 2))
plot(OR_RIV_L2E    , col = "cornflowerblue", lwd = 2, main = "Lignes initiales")
plot(OR_RIV_L2E_buf, col = "lightgrey", main = "Tampon")
plot(OR_RIV_L2E    , col = "cornflowerblue", lwd = 2, add = TRUE)
```

En ce qui concerne les polygones, il est possible de définir une zone "tampon" interne ou externe. Pour une zone interne, on fournit une valeur négative à l'argument **width**, pour une zone externe, on lui fournit une valeur positive.

```
SE_L2E_buf_1 <- gBuffer(SE_ctr_L2E, width = -1e4, capStyle = "ROUND")
SE_L2E_buf_2 <- gBuffer(SE_ctr_L2E, width = +1e4, capStyle = "ROUND")
```

Comme on peut le voir, la zone interne correspond au polygone de départ dont les bords ont été rognés. La zone externe, quant à elle, correspond au polygone de départ dont les frontières ont été étendues au-delà du contour initial.

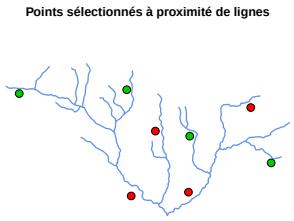
```
par(mfrow = c(1, 3))
plot(SE_ctr_L2E  , col = "cyan4", main = "Polygone initial")
plot(SE_ctr_L2E  , col = "cyan4", main = "Tampon interne")
plot(SE_L2E_buf_1, col = "lightgrey", add = TRUE)
plot(SE_L2E_buf_2, col = "lightgrey", main = "Tampon externe")
plot(SE_ctr_L2E  , col = NA, lty = "dotted", add = TRUE)
```

4.2.5 Trouver des entités dans un périmètre de recherche

Il est possible de trouver les points situés dans un périmètre de recherche défini. Pour cela, on peut utiliser la fonction **gWithinDistance()** du package **rgeos**. Ici, on recherche les points à proximité d'un cours d'eau, situés dans un périmètre de 350 mètres de celui-ci.

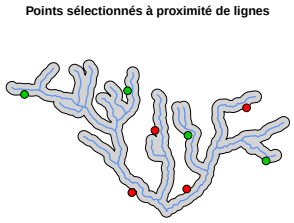
```
plot(OR_RIV_L2E, col = "cornflowerblue", lwd = 2, main = "Points sélectionnés à proximité de lignes")
points(OR_sta_L2E[which(rowSums(gWithinDistance(OR_RIV_L2E, OR_sta_L2E, 350, byid = TRUE)) != 0)],
       pch = 21, bg = "green3", cex = 1.6)
```

```
points(OR_sta_L2E[which(rowSums(gWithinDistance(OR_RIV_L2E, OR_sta_L2E, 350, byid = TRUE)) == 0)],
       pch = 21, bg = "red", cex = 1.6)
```



D'autres manières de faire sont possibles. On peut, par exemple, chercher les points intersectant une zone "tampon" de 350 mètres autour du réseau hydrographique. Dans ce cas là, on utilise les fonctions **gBuffer()** et **gIntersection()**.

```
OR_RIV_L2E_buf350 <- gBuffer(OR_RIV_L2E, width = 350)
plot(OR_RIV_L2E_buf350, col = "lightgrey", main = "Points sélectionnés à proximité de lignes")
plot(OR_RIV_L2E, col = "cornflowerblue", lwd = 2, add = TRUE)
points(gIntersection(OR_sta_L2E, OR_RIV_L2E_buf350), pch = 21, bg = "green3", cex = 1.6)
points(gDifference(OR_sta_L2E, OR_RIV_L2E_buf350), pch = 21, bg = "red", cex = 1.6)
```



4.2.6 Simplifier des entités vectorielles

Il est possible de simplifier des données vectorielles avec la fonction **gSimplify()**. Le paramètre de tolérance (**tol**) s'exprime dans l'unité du système de coordonnées de la couche. Attention toutefois, par défaut, la topologie ne préserve par la géométrie d'origine des entités. Il convient donc de se servir des fonctions qui permettent cette simplification avec précaution si l'on souhaite réaliser des analyses spatiales. Pour la représentation cartographique, en revanche, il n'y a pas de problème.

```
par(mfrow = c(1, 4))
for(i in c(0, 1e2, 2e2, 5e2)) {
  plot(gSimplify(OR_RIV_L2E, tol = i), col = "cornflowerblue", lwd = 2, main = paste("Tol =", i))
}
```

4.2.7 Calculer l'aire d'un polygone

On a vu que l'aire des polygones est directement disponible dans la structure des objets de classe **SpatialPolygons(DataFra**

```
SPoD[2, ]@polygons[[1]]@area
## [1] 185162660204
```

Avec cette manière de faire, comme déjà mentionné (§ 1.1.4), le problème est que la surface renvoyée est fausse dans les cas ou le polygone comporte un trou, car l'aire de ce dernier est ignorée au lieu d'être soustraite.

```
SPoD[1, ]@polygons[[1]]@area
## [1] 6.32985e+11
```

Dans ce cas, il faut extraire les aires des sous-polygones afin soustraire ou additionner les valeurs, selon le fait que le sous-polygone considéré est un trou ou non.

```
sapply(seq_along(SPoD[1, ]@polygons[[1]]@Polygons), function(x) {
  SPoD[1, ]@polygons[[1]]@Polygons[[x]]@area
  })
## [1] 626087257104   6897791544  76095778318
```

```
sapply(seq_along(SPoD[1, ]@polygons[[1]]@Polygons), function(x) {
  SPoD[1, ]@polygons[[1]]@Polygons[[x]]@ringDir
  })
## [1]  1  1 -1
sum(sapply(seq_along(SPoD[1, ]@polygons[[1]]@Polygons), function(x) {
  SPoD[1, ]@polygons[[1]]@Polygons[[x]]@area * SPoD[1, ]@polygons[[1]]@Polygons[[x]]@ringDir
  }))
## [1] 556889270330
```

De manière plus simple, on peut utiliser la fonction **gArea()** fournie par le package **rgeos** qui prend bien en le fait que certains polygones correspondent à des trous. Notez que cette fonction peut travailler sur des points ou des lignes, mais l'aire calculée sera alors logiquement nulle.

```
gArea(SPoD[1, ])
## [1] 556889270330
```

L'argument **byid = TRUE** permet de réaliser le calcul pour chacune des entités de la couche.

4.2.8 Calculer la longueur d'une entité

On peut utiliser la fonction **gLength()**, dont le fonctionnement est très semblable à **gArea()**. Par défaut, appliquée à des objets de classe **SpatialLines(DataFrame)**, elle renvoie la somme des longueurs des polylignes, pour des objets de classe **SpatialPolygons(DataFrame)**, elle calcule la somme des périmètres des polygones, et pour des points, elle renvoie logiquement une valeur nulle. L'argument **byid = TRUE** permet de réaliser le calcul pour chacune des entités de la couche considérée.

4.2.9 Autres fonctionnalités

Par ailleurs, le package **rgeos**, permet de regrouper des entités (**gUnion()**...), tester si elles se touchent, se superposent en partie ou complètement (**gTouches()**, **gContains()**, **gCrosses()**, **gIntersects()**), obtenir le polygone convexe (**gConvexHull()**), etc.

4.3 Package maptools

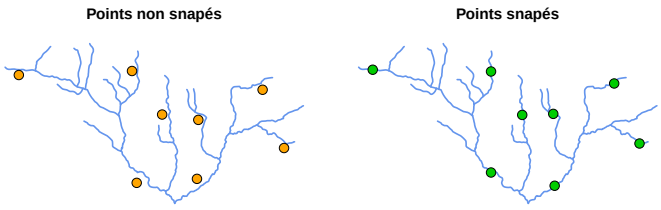
Le package **maptools** (Bivand & Lewin-Koh, 2015) permet principalement de réaliser des conversions de formats d'objets spatiaux (§ 1.3.1, 1.3.2 et 1.3.3) et d'importer ou d'exporter des données vectorielles au format Shapefile (§ 2.2.1), mais il propose également de nombreuses fonctionnalités d'analyse spatiale.

```
library(maptools)
```

4.3.1 Snaper des points sur un réseau

Pour snaper des points sur un réseau, on peut utiliser la fonction **project2segment()**. Dans l'exemple suivant, on cherche à raccorder des stations de mesure à un réseau hydrologique, ici celui du bassin versant de l'Orgeval. Les points et les lignes doivent, bien évidemment, êtres exprimés dans le même système de coordonnées.

```
OR_sta_L2E_snap <- snapPointsToLines(OR_sta_L2E, OR_RIV_L2E)
par(mfrow = c(1, 2))
plot(OR_RIV_L2E, col = "cornflowerblue", lwd = 2, main = "Points non snapés")
plot(OR_sta_L2E, pch = 21, bg = "orange", cex = 1.6, add = TRUE)
plot(OR_RIV_L2E, col = "cornflowerblue", lwd = 2, main = "Points snapés")
points(OR_sta_L2E_snap, pch = 21, bg = "green3", cex = 1.6)
```



4.3.2 Simplifier des entités vectorielles

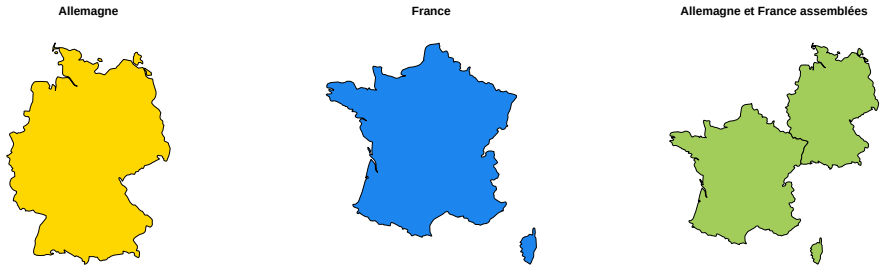
Il est possible de simplifier des données vectorielles avec la fonction **thinnedSpatialPoly()**. Attention, par défaut la topologie n'est pas respectée. Le paramètre de tolérance s'exprime dans l'unité du système de coordonnées de la couche.

```
par(mfrow = c(1, 4))
for(i in c(0, 1e4, 2e4, 5e4)) {
        plot(thinnedSpatialPoly(SE_ctr_L2E, tol = i), col = "cyan4", main = paste("Tol =", i))
}
```

4.3.3 Assembler des couches vectorielles

La fonction **spRbind()** permet d'assembler deux couches vectorielles (**SpatialPoints(DataFrame)**, **SpatialLines(DataFr** ou **SpatialPolygons(DataFrame)**). Ici, on ne fusionne pas les entités ; ces dernières restent bien distinctes les unes des autres.

```
DEFR_ctrUEr_LAEA <- spRbind(DE_ctrUE_LAEA, FR_ctrUE_LAEA)
par(mfrow = c(1, 3))
plot(DE_ctrUE_LAEA   , col = "gold", main = "Allemagne")
plot(FR_ctrUE_LAEA   , col = "dodgerblue2", main = "France")
plot(DEFR_ctrUEr_LAEA, col = "darkolivegreen3", main = "Allemagne et France assemblées")
```



Concernant les tables attributaires, elles sont également assemblées.

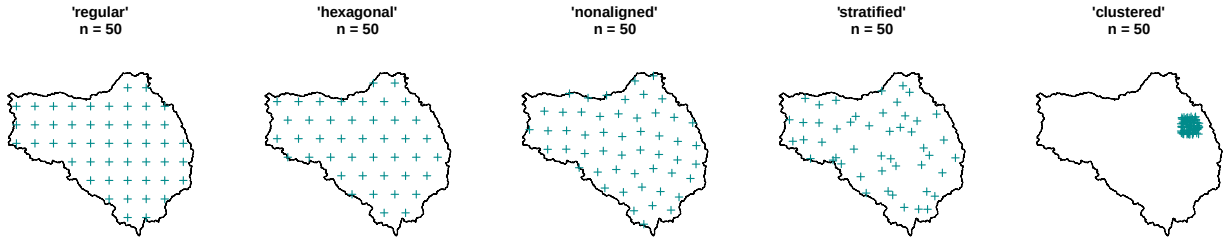
```
DEFR_ctrUEr_LAEA@data
##    id iso_a2 iso_a3    name continent           part   area      pop pop_dens        gdpps
## 14 14     DE    DEU Germany    Europe Western Europe 348540 82217837 235.8921 2.381851e+12
## 20 20     FR    FRA  France    Europe Western Europe 547561 63961956 116.8125 1.713350e+12
##     gdpps_pop  birth  death birth_rate death_rate
## 14 0.02897000 682514 844439   8.301289  10.270752
## 20 0.02678702 828404 542575  12.951511   8.482777
```

4.3.4 Tirer des points au hasard

Il est possible de réaliser un tirage aléatoire de positions géographiques sur des objets de classes **sp** et ce, grâce à la fonction **spsample()**. On peut, par exemple, tirer des points au hasard sur des lignes ou des polygones. L'objet renvoyé sera de classe **SpatialPoints**. Le nombre de points en sorti est approximé et ne correspond donc pas toujours exactement à ce qui a été demandé par l'utilisateur.

Plusieurs types de tirages aléatoires sont proposés :
- *regular* : alignement régulier ;
- *hexagonal* : alignement sur un treillis hexagonal ;
- *nonaligned* : les coordonnées X et Y sont aléatoires (indisponible pour les lignes) ;
- *stratified* : l'espace est découpé en cellules régulières et un point est tiré dans chacune de ces cellules ;
- *clustered* : tirage aléatoire d'une grappe de points ;
- *Fibonacci* : tirage de Fibonacci sur une sphère.

```
par(mfrow = c(1, 5))
for (sType in c("regular", "hexagonal", "nonaligned", "stratified", "clustered")) {
  plot(SE_ctr_L2E, main = paste(shQuote(sType), "n = 50", sep = "\n"))
  plot(spsample(SE_ctr_L2E, n = 50, type = sType), col = "cyan4", add = TRUE)
}
```



4.4 Package spatstat

Le package **spatstat** (Baddeley & Turner, 2005 ; Baddeley *et al.*, 2013) est spécialisé dans l'analyse de données spatiales des semis de points. Comme mentionné dans la partie relative aux formats de données (§ 1.3.3), les objets qu'il permet de manipuler sont de mode **S3**.

```
library(spatstat)
```

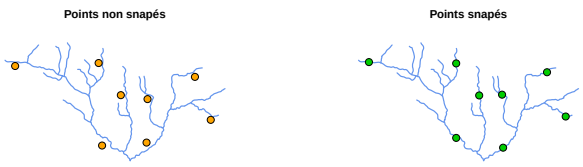
4.4.1 Snaper des points sur un réseau

En premier lieu, on définit une fenêtre de la classe **owin**, correspondant à l'emprise de la carte que l'on désire. On transforme le tableau de coordonnées de station dans la classe **ppp** et le réseau dans la classe **psp**.

```
winbox <- owin(OR_RIV_L2E@bbox["x", ], OR_RIV_L2E@bbox["y", ])
OR_sta_L2E_p <- ppp(OR_sta_L2E@coords[, "X"], OR_sta_L2E@coords[, "Y"], window = winbox)
OR_RIV_L2E_l <- as.psp(OR_RIV_L2E, winbox)
OR_sta_L2E_snap2 <- project2segment(OR_sta_L2E_p, OR_RIV_L2E_l)
str(OR_sta_L2E_snap2)
## List of 4
##  $ Xproj:List of 5
##   ..$ window    :List of 4
##   .. ..$ type  : chr "rectangle"
##   .. ..$ xrange: Named num [1:2] 647790 664016
##   .. .. ..- attr(*, "names")= chr [1:2] "min" "max"
##   .. ..$ yrange: Named num [1:2] 2425687 2434379
##   .. .. ..- attr(*, "names")= chr [1:2] "min" "max"
##   .. ..$ units :List of 3
##   .. .. ..$ singular  : chr "unit"
##   .. .. ..$ plural    : chr "units"
##   .. .. ..$ multiplier: num 1
##   .. .. ..- attr(*, "class")= chr "units"
##   .. ..- attr(*, "class")= chr "owin"
##   ..$ n         : int 8
##   ..$ x         : num [1:8] 654992 658453 658377 656689 654992 ...
##   ..$ y         : num [1:8] 2427368 2426647 2430557 2430500 2432863 ...
##   ..$ markformat: chr "none"
##   ..- attr(*, "class")= chr "ppp"
##  $ mapXY: int [1:8] 850 931 741 526 171 378 588 28
##  $ d    : num [1:8] 565 438 336 357 300 ...
##  $ tp   : num [1:8] 1 1 0.307 1 0.146 ...
```

On peut alors snaper les points sur le réseau avec la fonction **project2segment()**, qui renvoie une liste, et l'on peut dessiner le graphe avec la fonction **plot()**, en lui fournissant l'élément **$Xproj**.

```
par(mfrow = c(1, 2))
plot(OR_RIV_L2E, col = "cornflowerblue", lwd = 2, main = "Points non snapés")
plot(OR_sta_L2E, pch = 21, bg = "orange", cex = 1.6, add = TRUE)
plot(OR_RIV_L2E, col = "cornflowerblue", lwd = 2, main = "Points snapés")
plot(OR_sta_L2E_snap2$Xproj, pch = 21, bg = "green3", cex = 1.6, add = TRUE)
```



Notez que l'on peut récupérer les coordonnées en fournissant l'élément **$Xproj** à la fonction **coords()**.

```
coords(OR_sta_L2E_snap2$Xproj)
##          x       y
## 1 654992.0 2427368
## 2 658453.0 2426647
```

```
## 3 658377.1 2430557
## 4 656689.0 2430500
## 5 654991.8 2432863
## 6 661691.4 2432211
## 7 663069.8 2428945
## 8 648556.3 2432959
```

4.5 Package PBSmapping

Le package **PBSmapping** (Schnute *et al.*, 2015, 2004) est package d'analyse de données spatiales. Le cœur du code est en C ou en C++, ce qui lui confère une rapidité certaine. Comme mentionné dans la partie relative aux formats de données (§ 1.3.2), les objets qu'il permet de manipuler sont de mode **S3**, et il ne travaille pas dans tous les systèmes de coordonnées, mais seulement en coordonnées géographiques WGS 84 (code EPSG 4326) ou avec les projections UTM et Mercator (code EPSG 3857).

```
library(PBSmapping)
```

4.5.1 Agréger des entités vectorielles

On convertit tout d'abord le système de coordonnées des objets **sp** du LAEA vers le WGS 84.

```
EU_W84_DE <- spTransform(DE_ctrUE_LAEA, CRS("+init=epsg:4326"))
EU_W84_FR <- spTransform(FR_ctrUE_LAEA, CRS("+init=epsg:4326"))
```

Puis, on convertit les objets **sp** en objets **PolySet**.

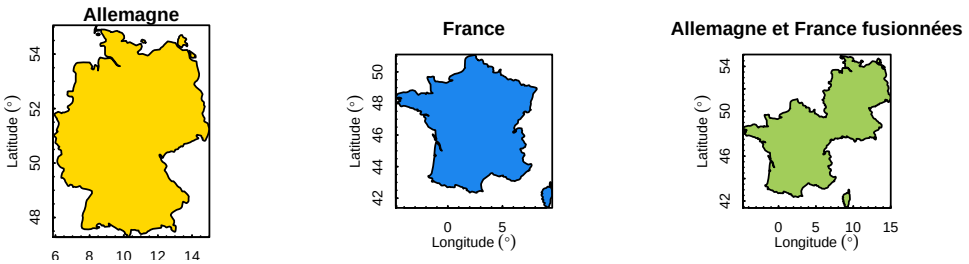
```
EU_W84_DE_ps <- SpatialPolygons2PolySet(EU_W84_DE)
EU_W84_FR_ps <- SpatialPolygons2PolySet(EU_W84_FR)
```

On réalise l'agrégation avec la fonction **joinPolys()** et l'argument **operation = "UNION"**.

```
EU_W84_DEFRu_ps <- joinPolys(EU_W84_DE_ps, EU_W84_FR_ps, operation = "UNION")
```

La représentation cartographique se fait par l'utilisation de la fonction **plotMap()**.

```
par(mfrow = c(1, 3))
plotMap(EU_W84_DE_ps   , col = "gold", plt = NULL, main = "Allemagne")
plotMap(EU_W84_FR_ps   , col = "dodgerblue2", plt = NULL, main = "France")
plotMap(EU_W84_DEFRu_ps, col = "darkolivegreen3", plt = NULL, main = "Allemagne et France fusionnées")
```



4.5.2 Différence d'entités vectorielles

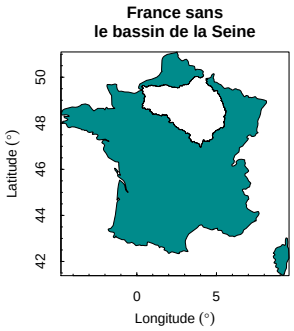
La différence entre des entités se fait également avec la fonction **joinPolys()**, mais avec l'argument **operation = "DIFF"**.

On prépare les données en créant des objets **PolySet** en WGS 84.

```
FR_ctr_W84 <- spTransform(FR_ctr_L2E, CRS("+init=epsg:4326"))
SE_ctr_W84 <- spTransform(SE_ctr_L2E, CRS("+init=epsg:4326"))
FR_ctr_W84_ps <- SpatialPolygons2PolySet(FR_ctr_W84)
SE_ctr_W84_ps <- SpatialPolygons2PolySet(SE_ctr_W84)
```

Graphiquement, on obtient le résultat suivant :

```
FR_ctr_W84_SE_clip <- joinPolys(FR_ctr_W84_ps, SE_ctr_W84_ps, operation = "DIFF")
plotMap(FR_ctr_W84_SE_clip, col = "cyan4", plt = NULL, main = "France sans\nle bassin de la Seine")
```



4.5.3 Intersection d'entités vectorielles

On prépare les données en créant des objets **PolySet** en WGS 84.

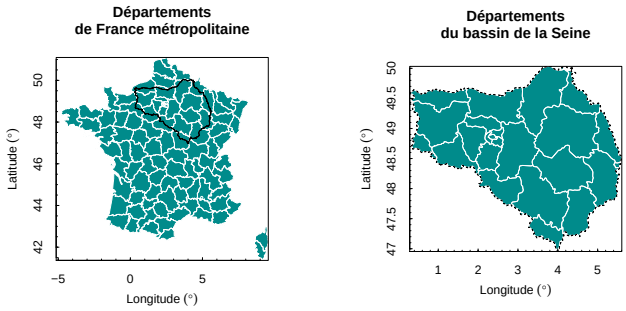
```
FR_dpt_W84 <- spTransform(FR_dpt_L2E, CRS("+init=epsg:4326"))
SE_ctr_W84 <- spTransform(SE_ctr_L2E, CRS("+init=epsg:4326"))
FR_dpt_W84_ps <- SpatialPolygons2PolySet(FR_dpt_W84)
SE_ctr_W84_ps <- SpatialPolygons2PolySet(SE_ctr_W84)
```

On réalise l'intersection avec la fonction **joinPolys()** et l'argument **operation = "INT"**.

```
SE_dpt_W84_ps <- joinPolys(FR_dpt_W84_ps, SE_ctr_W84_ps, operation = "INT")
```

On obtient le résultat suivant :

```
par(mfrow = c(1, 2))
plotMap(FR_dpt_W84_ps, col = "cyan4", border = "white", plt = NULL,
        main = "Départements\nde France métropolitaine")
addLines(SE_ctr_W84_ps, lty = 1)
plotMap(SE_dpt_W84_ps, col = "cyan4", border = "white", plt = NULL,
        main = "Départements\ndu bassin de la Seine")
addLines(SE_ctr_W84_ps, lty = 2, add = TRUE)
```



4.6 Package sp

Comme expliqué plus haut (§ 1.1 et 1.2), le package **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a) définit les classes et méthodes des données spatiales dont le format est le plus couramment utilisé dans R. Même si ce n'est pas le but premier de ce package, il est possible de réaliser quelques analyses spatiales avec ce package.

```
library(sp)
```

4.6.1 Assembler des couches vectorielles

La fonction **rbind()** permet d'assembler deux couches vectorielles. Bien entendu, cette fonction peut s'appliquer à divers types d'objets de classe **sp**.

```
apropos("rbind.Spatial")
##  [1] "rbind.SpatialLines"              "rbind.SpatialLinesDataFrame"
##  [3] "rbind.SpatialMultiPoints"        "rbind.SpatialMultiPointsDataFrame"
##  [5] "rbind.SpatialPixels"             "rbind.SpatialPixelsDataFrame"
##  [7] "rbind.SpatialPoints"             "rbind.SpatialPointsDataFrame"
##  [9] "rbind.SpatialPolygons"           "rbind.SpatialPolygonsDataFrame"
```

Ici, on ne fusionne pas les entités. En effet, ces dernières restent bien distinctes les unes des autres.

```
DEFR_ctrUEr_LAEA <- rbind(DE_ctrUE_LAEA, FR_ctrUE_LAEA)
par(mfrow = c(1, 3))
plot(DE_ctrUE_LAEA   , col = "gold", main = "Allemagne")
plot(FR_ctrUE_LAEA   , col = "dodgerblue2", main = "France")
plot(DEFR_ctrUEr_LAEA, col = "darkolivegreen3", main = "Allemagne et France assemblées")
```

Concernant les tables attributaires, elles sont, également assemblées.

```
DEFR_ctrUEr_LAEA@data
##    id iso_a2 iso_a3    name continent           part   area      pop pop_dens        gdpps
## 14 14     DE    DEU Germany    Europe Western Europe 348540 82217837 235.8921 2.381851e+12
## 20 20     FR    FRA  France    Europe Western Europe 547561 63961956 116.8125 1.713350e+12
##     gdpps_pop  birth  death birth_rate death_rate
## 14 0.02897000 682514 844439   8.301289  10.270752
## 20 0.02678702 828404 542575  12.951511   8.482777
```

Par ailleurs, il arrive parfois que l'on souhaite assembler des couches contenues dans une liste. C'est typiquement le cas lorsqu'on lit des fichiers en masse à l'aide de la fonction **`lapply()`**. Pour assembler l'ensemble des couches d'une liste, il suffit d'utiliser la fonction **`Reduce()`**, qui est une fonction de base de R. Pour que cela fonctionne, il faut bien veiller à ce que les couches présentent des identifiants différents les uns des autres (élément **`@polygons[[1]]@ID`**). Si tel n'est pas le cas, il faut utiliser au préalable la fonction **`spChFIDs()`** du package **`sp`**, qui permet de renommer facilement les identifiants des objets **`sp`**.

```
DE_ctrUE_LAEA <- spChFIDs(DE_ctrUE_LAEA, "1")
FR_ctrUE_LAEA <- spChFIDs(FR_ctrUE_LAEA, "2")
DE_ctrUE_LAEA@polygons[[1]]@ID
## [1] "1"
FR_ctrUE_LAEA@polygons[[1]]@ID
## [1] "2"
DEFR_ctrUEr_LAEA_list <- list(DE_ctrUE_LAEA, FR_ctrUE_LAEA)
str(DEFR_ctrUEr_LAEA_list, max.level = 3)
## List of 2
##  $ :Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
##   .. ..@ data       :'data.frame': 1 obs. of  15 variables:
##   .. ..@ polygons   :List of 1
##   .. ..@ plotOrder  : int 1
##   .. ..@ bbox       : num [1:2, 1:2] 4030570 2685042 4670922 3551503
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##  $ :Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
##   .. ..@ data       :'data.frame': 1 obs. of  15 variables:
##   .. ..@ polygons   :List of 1
##   .. ..@ plotOrder  : int 1
##   .. ..@ bbox       : num [1:2, 1:2] 3234114 2031912 4284216 3136297
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
DEFR_ctrUEr_LAEA <- Reduce("rbind", DEFR_ctrUEr_LAEA_list)
str(DEFR_ctrUEr_LAEA, max.level = 2)
## Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
##   ..@ data       :'data.frame': 2 obs. of  15 variables:
##   ..@ polygons   :List of 2
##   ..@ plotOrder  : int [1:2] 2 1
##   ..@ bbox       : num [1:2, 1:2] 3234114 2031912 4670922 3551503
##   .. ..- attr(*, "dimnames")=List of 2
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
```

4.6.2 Ajouter des données dans la table attributaire d'une couche vectorielle

Le package **`sp`** permet d'ajouter des colonnes à la table attributaire d'un objet **`sp`**. Dans l'exemple suivant, on dispose du **`data.frame`** contenant les noms des capitales de l'Allemagne et de la France, ainsi que leurs populations respectives.

```
DEFR_ctrUEr_LAEA_cap <- data.frame(name     = c("Germany", "France"),
                                   cam_name = c( "Berlin",  "Paris"),
                                   cap_pop  = c(  3452911,  2240621))
```

La fonction **merge()** permet de joindre les données de ce **data.frame** directement à l'objet spatiale **sp**. Les arguments de la fonction **merge()** classique sont disponibles, ce qui permet, par exemple de choisir la colonne qui permet de réaliser la jointure, de joindre l'ensemble des lignes ou seulement les lignes communes entre les deux tables, etc.

```
DEFR_ctrUEr_LAEA <- merge(DEFR_ctrUEr_LAEA, DEFR_ctrUEr_LAEA_cap, by = "name")
DEFR_ctrUEr_LAEA@data
##      name id iso_a2 iso_a3 continent           part   area      pop pop_dens        gdpps
## 2 Germany 14     DE    DEU    Europe Western Europe 348540 82217837 235.8921 2.381851e+12
## 1  France 20     FR    FRA    Europe Western Europe 547561 63961956 116.8125 1.713350e+12
##    gdpps_pop  birth  death birth_rate death_rate cam_name cap_pop
## 2 0.02897000 682514 844439   8.301289  10.270752   Berlin 3452911
## 1 0.02678702 828404 542575  12.951511   8.482777    Paris 2240621
```

4.6.3 Intersecter des couches vectorielles

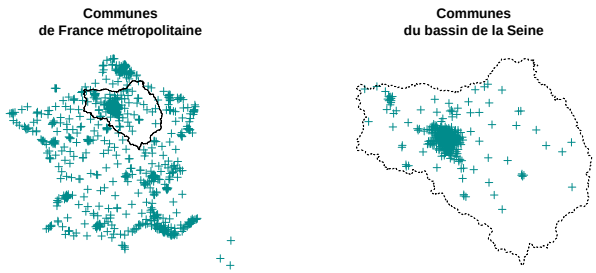
À l'aide des propriétés d'indexation des objets **sp**, il est possible d'intersecter des couches vectorielles de manière très simple et très rapide. La solution utilisée ici peut paraître particulièrement contre-intuitive au premier abord, mais elle est très efficace. En fait, la solution consiste à indexer une couche par une autre. Au lieu d'utiliser la fonction **gIntersection()** du package **rgeos** (Bivand & Rundel, 2014), on utilise donc les propriétés même des objets **sp**. Cependant, le résultat obtenu diffère un peu de celui renvoyé par la fonction **gIntersection()** (§ 4.2.3). Ici, on récupère les entités qui sont le résultat de l'intersection entre les deux couches. Elles restent intègres, c'est-à-dire que l'on n'a pas découpé une couche par une autre, mais juste réalisé une sélection des entités se chevauchant.

Ici, on souhaite à nouveau sélectionner les principales communes françaises qui se situent dans le bassin versant de la Seine. Il s'agit d'indexer la couche de points par la couche du polygones, afin de ne conserver que les points inclus dans ce dernier. Quand il s'agit de points, le résultat est identique à celui obtenu en utilisant la fonction **gIntersection()**, car les points n'ont pas de dimension.

```
SE_com_L2Esp <- FR_com_L2E[SE_ctr_L2E, ]
```

Graphiquement, on obtient la représentation suivante :

```
par(mfrow = c(1, 2))
plot(FR_com_L2E, col = "cyan4", main = "Communes\nde France métropolitaine")
plot(SE_ctr_L2E, lty = 1, add = TRUE)
plot(SE_com_L2Esp, col = "cyan4", main = "Communes\ndu bassin de la Seine")
plot(SE_ctr_L2E, lty = 2, add = TRUE)
```

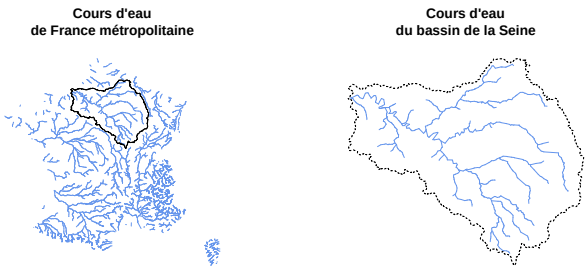


Cette solution fonctionne également si l'on souhaite intersecter des polylignes par un polygone. On reprend l'exemple où l'on souhaite ne conserver que les cours d'eau du bassin versant de la Seine. On indexe la couche des cours d'eau par celle du bassin versant.

```
SE_riv_L2Esp <- FR_riv_L2E[SE_ctr_L2E, ]
```

Graphiquement, on obtient la représentation suivante :

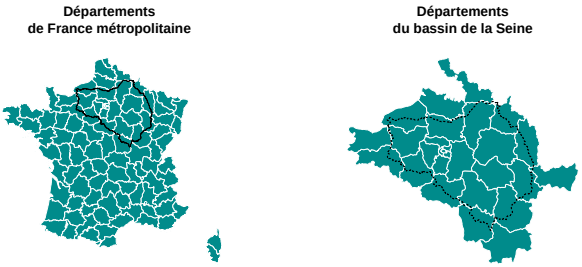
```
par(mfrow = c(1, 2))
plot(FR_riv_L2E, col = "cornflowerblue", main = "Cours d'eau\nde France métropolitaine")
plot(SE_ctr_L2E, lty = 1, add = TRUE)
plot(SE_riv_L2Esp, col = "cornflowerblue", main = "Cours d'eau\ndu bassin de la Seine")
plot(SE_ctr_L2E, lty = 2, add = TRUE)
```



Il est également possible d'intersecter des polygones par un autre polygone. Dans l'exemple suivant, on cherche à conserver les départements qui appartiennent au bassin versant de la Seine. On indexe la couche des départements par celle du bassin versant. On voit bien que les départements sont restés intacts, leurs contours n'ont pas été découpés par celui du bassin versant, ce qui n'était pas le cas avec **gIntersection()**.

```
SE_dpt_L2Esp <- FR_dpt_L2E[SE_ctr_L2E, ]
```

```
par(mfrow = c(1, 2))
plot(FR_dpt_L2E, col = "cyan4", border = "white", main = "Départements\nde France métropolitaine")
plot(SE_ctr_L2E, lty = 1, add = TRUE)
plot(SE_dpt_L2Esp, col = "cyan4", border = "white", main = "Départements\ndu bassin de la Seine")
plot(SE_ctr_L2E, lty = 2, add = TRUE)
```



Cette manière d'intersecter des couches est très simple et permet de se passer de l'utilisation d'un package supplémentaire. En revanche, comme elle ne découpe pas les entités, si on souhaite le faire, il faudra passer par la fonction **gIntersection()** du package **rgeos**. Cette dernière pouvant être assez lente, une solution peut consister à associer les deux manières de faire. On peut présélectionner les entités en réalisant l'intersection en indexant une couche par l'autre, puis on les découpe en utilisant **gIntersection()**.

4.7 Package spdep

Le package **spdep** (Bivand & Piras, 2015 ; Bivand *et al.*, 2013b) est spécialisé en statistique et en modélisation spatiales.

```
library(spdep)
```

4.7.1 Identifier les voisins d'une liste de polygones

On peut trouver les voisins contigus des polygones d'un **SpatialPolygons** grâce à la fonction **poly2nb()**. L'objet renvoyé est de classe **nb**.

```
EU_ctr1_LAEA_nb <- poly2nb(EU_ctr1_LAEA)
class(EU_ctr1_LAEA_nb)
## [1] "nb"
```

Il correspond à une liste qui compte autant d'éléments que de **Polygons** dans l'objet **SpatialPolygons**, et chacun de ces éléments est un vecteur correpondant aux indices des polygones contigus du **Polygons** considéré.

```
summary(EU_ctr1_LAEA_nb)
## Neighbour list object:
## Number of regions: 68
## Number of nonzero links: 230
## Percentage nonzero weights: 4.974048
## Average number of links: 3.382353
```

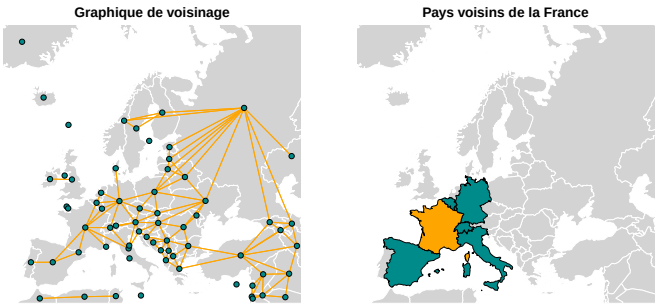
```
## 10 regions with no links:
## 2 12 21 24 26 29 33 36 49 68
## Link number distribution:
##
##  0  1  2  3  4  5  7  8  9 11
## 10  9 10  5 12 13  3  4  1  1
## 9 least connected regions:
## 15 22 30 38 45 46 54 59 65 with 1 link
## 1 most connected region:
## 57 with 11 links
```

```
head(EU_ctr1_LAEA_nb)
```

```
## [[1]]
## [1] 25 39 48 50
##
## [[2]]
## [1] 0
##
## [[3]]
## [1] 17 20
##
## [[4]]
## [1]  6 23 31 66
##
## [[5]]
## [1] 11 13 14 28 35 41 61 62
##
## [[6]]
## [1]  4 23 31 57 66
```

Un appel à la fonction **plot()** sur un objet **nb** permet de tracer le graphique de voisinage.

```
nb_id <- which(EU_ctr1_LAEA@data$name == "France")
nb_pal <- rep("lightgrey", length(EU_ctr1_LAEA))
nb_pal[nb_id] <- "orange"
nb_pal[EU_ctr1_LAEA_nb[[nb_id]]] <- "cyan4"
par(mfrow = c(1, 2))
plot(EU_ctr1_LAEA, col = "lightgrey", border = "white", main = "Graphique de voisinage")
plot(EU_ctr1_LAEA_nb, coordinates(EU_ctr1_LAEA),
     col = "orange", bg = "cyan4", pch = 21, add = TRUE)
plot(EU_ctr1_LAEA, col = "lightgrey", border = "white", main = "Pays voisins de la France")
plot(EU_ctr1_LAEA[nb_id, ], col = "orange", add = TRUE)
plot(EU_ctr1_LAEA[EU_ctr1_LAEA_nb[[nb_id]], ], col = "cyan4", add = TRUE)
```



4.8 Package dismo

Le package **dismo** (Hijmans *et al.*, 2014) est spécialisé dans la modélisation de la distribution d'espèces (prédiction de la répartition géographiques à partir d'occurrences à des stations). Dans ce cadre, il propose quelques fonctions d'analyse spatiales intéressantes.

```
library(dismo)
```

4.8.1 Déterminer des polygones de Thiessen

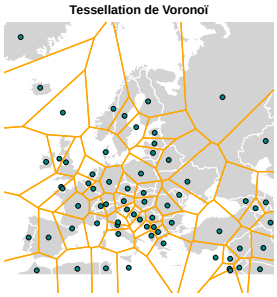
La fonction **voronoi()** permet de calculer les polygones de Thiessen. D'autres packages le permettent, c'est notamment le cas de **tripack** (Renka R.J. (Fortran code) *et al.*, 2013) et de sa fonction **voronoi.mosaic()**, mais l'avantage de la fonction disponible dans la package **dismo** est qu'elle renvoie directement un objet de classe **sp**.

La fonction **voronoi()** prend en entrée un objet **SpatialPoints(DataFrame)** ou une matrice de coordonnées. La fonction **coordinates()**, permet de récupérer les centroïdes d'un objet **SpatialPolygons(DataFrame)**

```
EU_ctr1_LAEA_vor <- voronoi(coordinates(EU_ctr1_LAEA))
print(EU_ctr1_LAEA_vor)
```

```
## class       : SpatialPolygonsDataFrame
## features    : 68
## extent      : 2144426, 7962304, 929707.5, 6365506  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## variables   : 1
## names       : id
## min values  :  1
## max values  : 68
```

```
plot(EU_ctr1_LAEA, col = "lightgrey", border = "white", main = "Tessellation de Voronoï")
plot(EU_ctr1_LAEA_vor, border = "orange", lwd = 2, add = TRUE)
points(coordinates(EU_ctr1_LAEA), pch = 21, bg = "cyan4")
```



4.9 Package geosphere

Le package **geosphere** (Hijmans, 2014) permet de réaliser des opérations trigonométriques sphériques.

```
library(geosphere)
```

4.9.1 Calculer les coordonnées sur un grand cercle

La fonction **greatCircle()** permet de calculer des coordonnées du chemin le plus court sur un grand cercle à partir des coordonnées de deux points exprimées en WGS 84. Dans l'exemple suivant, on connaît les coordonnées de deux villes (Paris et New York).

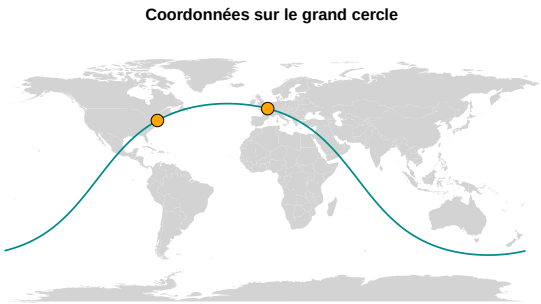
```
xy_PA <- c(+ 2.35, +48.85)
xy_NY <- c(-73.78, +40.63)
```

L'objet renvoyé est une matrice comportant les coordonnées.

```
gc_PANY <- greatCircle(xy_PA, xy_NY)
head(gc_PANY)
##       lon       lat
## [1,] -179 -49.19120
## [2,] -178 -48.94042
## [3,] -177 -48.67837
## [4,] -176 -48.40482
## [5,] -175 -48.11953
## [6,] -174 -47.82226
```

On obtient le grand cercle suivant :

```
plot(WD_ctr_W84_po, col = "lightgrey", border = NA, main = "Coordonnées sur le grand cercle")
lines(gc_PANY, col = "cyan4", lwd = 2)
points(rbind(xy_PA, xy_NY), bg = "orange", pch = 21, cex = 2)
```



4.9.2 Calculer les coordonnées d'un trajet sur le grand cercle

La fonction **gcIntermediate()** permet de calculer les coordonnées d'un trajet sur un grand cercle entre les coordonnées de deux points exprimées en WGS 84. Dans l'exemple suivant, on connaît les coordonnées de trois villes (Paris, New York et Johannesburg).

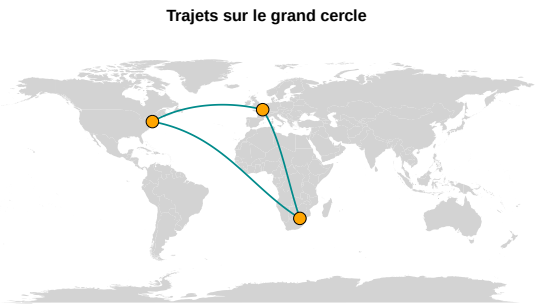
```
xy_PA <- c(+ 2.35, +48.85)
xy_NY <- c(-73.78, +40.63)
xy_JB <- c(+28.04, -26.20)
```

L'objet renvoyé est une matrice comportant les coordonnées du trajet.

```
gi_PANY <- gcIntermediate(xy_PA, xy_NY)
gi_NYJB <- gcIntermediate(xy_NY, xy_JB)
gi_JBPA <- gcIntermediate(xy_JB, xy_PA)
head(gi_JBPA)
##           lon       lat
## [1,] 27.54682 -24.72689
## [2,] 27.06519 -23.25220
## [3,] 26.59410 -21.77607
## [4,] 26.13262 -20.29864
## [5,] 25.67987 -18.82002
## [6,] 25.23501 -17.34032
```

On peut alors représenter les différents trajets calculés entre les trois villes.

```
plot(WD_ctr_W84_po, col = "lightgrey", border = NA, main = "Trajets sur le grand cercle")
lines(gi_PANY, col = "cyan4", lwd = 2)
lines(gi_NYJB, col = "cyan4", lwd = 2)
lines(gi_JBPA, col = "cyan4", lwd = 2)
points(rbind(xy_PA, xy_NY, xy_JB), bg = "orange", pch = 21, cex = 2)
```



Chapitre 5

Traitements externalisés

Divers packages permettent de réaliser des opérations de géomatique directement dans l'environnement de R, c'est notamment le cas de **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a), **maptools** (Bivand & Lewin-Koh, 2015), **rgdal** (Bivand *et al.*, 2014), **raster** (Hijmans, 2015), **rgeos** (Bivand & Rundel, 2014), etc. Cette façon de manier les données spatiales est extrêmement souple, car elle permet de ne pas multiplier les fichiers géographiques en tant que tel ; les entrées et sorties de fonctions sont simplement manipulées sous forme d'objets dans la mémoire de R, et non sous la forme de fichiers physiques sur le disque dur de la machine. Une des limites d'utilisation de ces packages est que, dans les circonstances où les données sont volumineuses, les imports et exports peuvent être très longs, et c'est bien évidemment aussi le cas pour les traitements de géomatique. R n'est, en effet, pas réputé pour être un langage rapide.

Pour contourner ces problèmes, les packages wrappers permettent d'attaquer directement des bibliothèques de fonctions ou les logiciels de géomatique les plus courants, sans réaliser les traitements directement dans R. Seules les lignes de codes sont écrites dans R, ce dernier se chargeant alors de communiquer avec les outils externes. Ceci permet de rester dans l'environnement de travail de R, et évite de passer par les interfaces ou les lignes de commandes de chaque logiciel. Tout en restant dans l'environnement de R, on bénéficie ainsi de la rapidité des langages utilisés par ces bibliothèques et logiciels (C, C++, Python, etc.). Chaque fonction nécessitera au moins un fichier d'entrée et créera un ou plusieurs fichiers de sortie, ce qui multipliera les fichiers produits, mais la rapidité est à ce prix !

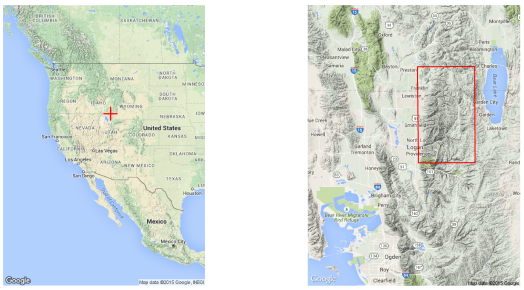
Ici, on expliquera par l'exemple comment réaliser des traitements grâce aux différents logiciels ou bibliothèques de géomatique que sont GDAL/OGR (GDAL Development Team, 2012), GRASS GIS (GRASS Development Team, 2015), SAGA GIS (SAGA Development Team, 2008), ArcGIS (ESRI, 2013), TauDEM (Tarboton, 1997, 2013) et QGIS (QGIS Development Team, 2016).

Nous n'aborderons ici pas le procédé inverse, qui consiste à attaquer R depuis des logiciels de géomatique, comme le propose QGIS, grâce à Sextante, ou ArcGIS *via* le R ArcGIS Bridge [1]. On notera l'existence du package **arcgisbinding** (ESRI, 2016a), que l'on peut installer depuis son dépôt GitHub [2], et qui fournit des classes pour le chargement, la conversion et l'exportation de jeu données ArcGIS et des couches dans R.

Description des données utilisées

On dispose d'une représentation de la topographie sous la forme d'un modèle numérique de terrain (au format GeoTIFF [3], d'un réseau hydrographique (au format vectoriel Shapefile) [4], ainsi que de la position de l'exutoire d'un bassin versant dont le contour reste à déterminer (au format vectoriel Shapefile). Les données sont projetées en NAD83 / UTM zone 12N, dont le code EPSG correspondant est le 26912. La zone d'étude se situe aux États-Unis, dans l'état de l'Utah, sur le territoire de la commune de Logan, située au nord de Salt Lake City.

1. Procédure d'installation de R sous ArcGIS : https://github.com/R-ArcGIS/r-bridge-install/.
2. Sources du package arcgisbinding : https://github.com/R-ArcGIS/r-bridge/tree/master/src/package/.
3. Source : D. Tarboton, Utah Univesitity (http://hydrology.usu.edu/taudem/taudem5/index.html).
4. Source : EPA, USGS, National Operational Hydrologic Remote Sensing Center (http://www.nws.noaa.gov/geodata/catalog/hydro/html/rivers.htm).



Modèle numérique de terrain

On dispose d'un fichier contenant un modèle numérique de terrain. À l'aide du package **raster** (Hijmans, 2015), on peut charger dans R le fichier matriciel contenant le MNT ("logan_MNT.tif").

```
RAS_mnt <- raster::raster("01_data/logan_MNT.tif")
```

Les caractéristiques du raster d'élévation sont les suivantes :

```
print(RAS_mnt)
## class       : RasterLayer
## dimensions  : 1660, 985, 1635100  (nrow, ncol, ncell)
## resolution  : 30, 30  (x, y)
## extent      : 432404, 461954, 4612592, 4662392  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=12 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs
## data source : E:\Data\boulot\misc\scientific_doc\R\formation\irstea\033-geomatique_LIV\01_manuscrit_KNITR\03_par
## names       : logan_MNT
## values      : 1363.818, 2998.83  (min, max)
```

Réseau hydrographique observé

On dispose également d'un fichier contenant un réseau hydrographique observé. À l'aide des packages **maptools** (Bivand & Lewin-Koh, 2015) ou **rgdal** (Bivand *et al.*, 2014), on peut charger le fichier Shapefile du réseau hydrographique ("rv13my07.shp").

```
VEC_riv <- rgdal::readOGR("01_data", "rv13my07", p4s = "+init=epsg:26912")
## OGR data source with driver: ESRI Shapefile
## Source: "01_data", layer: "rv13my07"
## with 8 features
## It has 17 fields
```

Les caractéristiques des données vectorielles du réseau hydrographique sont les suivantes :

```
print(VEC_riv)
## class       : SpatialLinesDataFrame
## features    : 8
## extent      : 432404, 461954, 4621037, 4662392  (xmin, xmax, ymin, ymax)
## coord. ref. : +init=epsg:26912 +proj=utm +zone=12 +datum=NAD83 +units=m +no_defs +ellps=GRS80 +towgs84=0,0,0
## variables   : 17
## names       : IHABBSRF_I,           RR,      HUC, TYPE, PMILE,     PNAME, OWNAME,       PNMCD, OWNMCD,       DSRE
## min values  :      57252, 16010201019, 16010201,    R,  78.5, BEAVER CR,      0, 16010201009,     NA, 2147483647
## max values  :      57608, 16010203015, 16010203,    S, 237.2, SUMMIT CR,      0, 16010203008,     NA, 2147483647
```

Position de l'exutoire du bassin versant à définir

Par ailleurs, on dispose d'un fichier contenant la position de l'exutoire d'un bassin versant que l'on souhaite définir. On s'intéresse également au bassin versant défini par l'exutoire.

```
EXU_pos <- rgdal::readOGR("01_data", "exu_POS", verbose = FALSE)
```

Les caractéristiques du fichier vectoriel de la position de l'exutoire du bassin versant à définir sont les suivantes :

```
print(EXU_pos)
## class       : SpatialPointsDataFrame
## features    : 1
## extent      : 453217.4, 453217.4, 4638915, 4638915  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=12 +datum=NAD83 +units=m +no_defs +ellps=GRS80 +towgs84=0,0,0
## variables   : 1
## names       : id
## min values  :  1
## max values  :  1
```

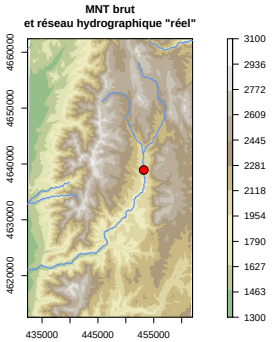
Quant aux coordonnées, elles sont les suivantes :

```
coordinates(EXU_pos)
##      coords.x1 coords.x2
## [1,]  453217.4   4638915
```

Représentation cartographique

On peut afficher les objets géographiques, qu'ils soient matriciels ou vectoriels, par de simples appels à la fonction **plot()**. Au besoin, on peut superposer les couches en utilisant l'argument **add = TRUE**.

```
plot(RAS_mnt, legend.shrink = 1,
     breaks = ALT_brk, col = ALT_pal,
     main = "MNT brut\net réseau hydrographique \"réel\"")
plot(VEC_riv, col = "cornflowerblue", lwd = 2, add = TRUE)
plot(EXU_pos, pch = 21, bg = "red", cex = 2, add = TRUE)
```



Avertissement

> D'une manière générale, quel que soit le logiciel considéré, les noms des paramètres des fonctionnalités peuvent changer d'une version à l'autre. Au besoin, il faudra donc les modifier, et assurer ainsi le bon fonctionnement des commandes.

5.1 Bibliothèques GDAL/OGR

GDAL (*Geospatial Data Abstraction Library*) est une bibliothèque de fonctions, libre, existant depuis 2007, et permettant de lire et de traiter un très grand nombre de formats de données matricielles. (Wikipedia, 2015b.)

OGR est une bibliothèque de fonctions permettant d'accéder à la plupart des formats courants de données vectorielles.

Ces deux bibliothèques sont désormais rassemblées sous la simple appellation "GDAL", ou parfois "GDAL/OGR" (GDAL Development Team, 2012). Elles permettent de réaliser des opérations depuis des langages de programmation tels que C, C++, C sharp/.NET, Java, Ruby, VB6, Perl, Python, ou encore le langage R. GDAL/OGR est utilisé par de très nombreux logiciels de géomatique, parmi lesquels : ArcGIS, Google Earth, GRASS GIS, OSSIM, QGIS ou SAGA GIS. La bibliothèque GDAL/OGR est un des piliers des systèmes d'information géographique libres, car elle permet d'assurer la compatibilité avec de nombreux systèmes commerciaux reposant sur des formats propriétaires, tout autant que sur les normes de l'*Open Geospatial Consortium*. (Wikipedia, 2015b.)

La version binaire inclut de nombreux utilitaires de conversion, de transformation et de reprojection pour traiter directement les rasters ou les vecteurs.

Pour établir la connexion entre GDAL/OGR et R, il faut utiliser le package **gdalUtils** (Greenberg & Mattiuzzi, 2014).

```
library(gdalUtils)
```

5.1.1 Création de la connexion

On initialise la connexion à l'aide de la fonction **gdal_setInstallation()**. Pour cela, on fournit le chemin du répertoire contenant les modules GDAL/OGR à l'argument **search_path**. Sous Windows, ils peuvent, par exemple, se trouver dans OSGeo4W ou dans QGIS. Si aucun chemin n'est spécifié en argument, la fonction ira chercher s'il est écrit dans le path de l'ordinateur. Sous Windows, il est préférable d'aller inscrire le répertoire contenant les exécutables et les DLL dans le path. Par ailleurs, selon les installations, il sera peut-être nécessaire d'ajouter certaines DLL qui peuvent être manquantes sur la machine.

```
gdal_setInstallation(search_path = "C:/OSGeo4W64/bin",
                     rescan = FALSE, ignore.full_scan = TRUE, verbose = TRUE)
```

On peut obtenir les options de connexion à GDAL/OGR *via* la commande suivante :

```
str(getOption("gdalUtils_gdalPath"))
## List of 1
##  $ :List of 5
##   ..$ path            : chr "C:/OSGEO4~1/bin/"
##   ..$ version         : Named chr "1.11.2"
##   .. ..- attr(*, "names")= chr "version"
##   ..$ date            : Named chr "2015-02-10"
##   .. ..- attr(*, "names")= chr "date"
##   ..$ drivers         :'data.frame': 123 obs. of  7 variables:
##   .. ..$ format_code: Factor w/ 123 levels "AAIGrid","ACE2",..: 26 65 37 80 81 118 47 86 104 25 ...
##   .. ..$ read       : logi [1:123] TRUE TRUE TRUE TRUE TRUE TRUE ...
##   .. ..$ write      : logi [1:123] FALSE FALSE TRUE FALSE FALSE TRUE ...
##   .. ..$ update     : logi [1:123] FALSE FALSE TRUE FALSE FALSE TRUE ...
##   .. ..$ virtualIO  : logi [1:123] TRUE TRUE FALSE FALSE TRUE TRUE ...
##   .. ..$ subdatasets: logi [1:123] FALSE FALSE TRUE FALSE FALSE FALSE ...
##   .. ..$ format_name: Factor w/ 122 levels "ACE2","AirSAR Polarimetric Image",..: 25 28 87 70 72 119 38 75 99 19
##   ..$ python_utilities: chr [1:23, 1] "epsg_tr.py" "esri2wkt.py" "gcps2vec.py" "gcps2wld.py" ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr "C:/OSGEO4~1/bin/"
```

La connexion étant créée, il est désormais possible de réaliser des traitements de géomatique.

5.1.2 Traitement des données

Nous allons commencer par présenter deux exemples de fonctionnalités GDAL/OGR implantées dans des fonctions du package **gdalUtils**, puis nous nous intéresserons à l'utilisation de la fonction générique qui permet d'utiliser n'importe quelle commande GDAL/OGR.

5.1.2.1 Transformation du système de coordonnées

Dans l'exemple suivant, on souhaite transformer la projection de la carte du NAD83 / UTM zone 12N (EPSG : 26912), vers les projections équivalentes pour les zones UTM 11N (EPSG : 26911) et 13N (EPSG : 26913).

5.1.2.1.a Données matricielles

Pour les données matricielles, il faut utiliser la fonction **gdalwarp()**. Parmi les nombreux arguments que propose cette fonction, ceux qui sont indispensables sont ceux qui définissent la source des données (**srcfile**), la destination des données (**dstfile**), ainsi que le système de coordonnées de destination (**t_srs**). Ici, le système de coordonnées de la source (**s_srs**) a été défini, mais c'est inutile si les données de départ sont déjà géoréférencées (ce qui est en réalité le cas avec les données de l'exemple).

Ci-dessous, on convertit le raster du MNT de la zone 12N à la zone 11N.

```
gdalwarp(srcfile = "01_data/logan_MNT.tif",
         dstfile = "01_data/logan_MNT_UTM11N.tif",
         s_srs = projection(CRS("+init=epsg:26912")),
         t_srs = projection(CRS("+init=epsg:26911")),
         overwrite = TRUE)
## NULL
```

De la même manière, on convertit le raster de la zone 12N à la zone 13N.

```
gdalwarp(srcfile = "01_data/logan_MNT.tif",
         dstfile = "01_data/logan_MNT_UTM13N.tif",
         s_srs = projection(CRS("+init=epsg:26912")),
         t_srs = projection(CRS("+init=epsg:26913")),
         overwrite = TRUE)
## NULL
```

Notez que si l'on assigne le tout dans un objet R, tout en définissant l'argument **output_Raster = TRUE**, on récupérera alors un objet de la classe **RasterBrick**. Ceci permet donc de disposer du nouveau raster directement dans la mémoire de R, sans avoir à utiliser une autre fonction pour importer les données depuis le fichier de sortie ; on ne peut toutefois pas se passer de créer ce dernier.

```
RAS_mnt_UTM11N <- gdalwarp(srcfile = "01_data/logan_MNT.tif",
                           dstfile = "01_data/logan_MNT_UTM11N.tif",
                           t_srs   = projection(CRS("+init=epsg:26911")),
                           overwrite = TRUE, output_Raster = TRUE)
print(RAS_mnt_UTM11N)
## class       : RasterBrick
## dimensions  : 1725, 1099, 1895775, 1  (nrow, ncol, ncell, nlayers)
## resolution  : 30.07214, 30.07214  (x, y)
## extent      : 928507, 961556.3, 4625290, 4677164  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=11 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs
## data source : E:\Data\boulot\misc\scientific_doc\R\formation\irstea\033-geomatique_LIV\01_manuscrit_KNITR\03_par
## names       : logan_MNT_UTM11N
```

5.1.2.1.b Données vectorielles

Pour les données vectorielles, il faut utiliser la fonction **ogr2ogr()**. Le principe est identique à celui appliqué pour les rasters ; il faut définir la source des données (**src_datasource_name**), la destination des données (**dst_datasource_name**), ainsi que le système de coordonnées de destination (**t_srs**). Le système de coordonnées de la source (**s_srs**) a été défini, mais ce n'est pas obligatoire (pour les mêmes raisons que celles expliquées précédemment).

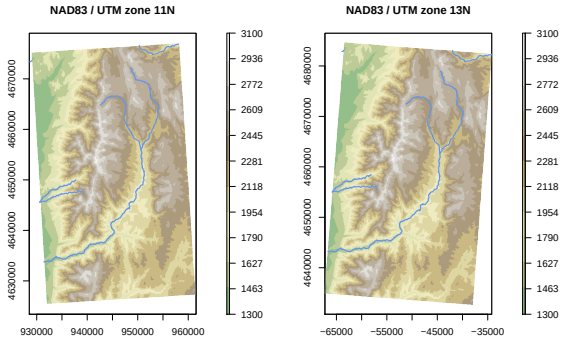
On convertit le fichier vectoriel, contenant la couche des cours d'eau, de la zone 12N à la zone 11N.

```
ogr2ogr(src_datasource_name = "01_data/rv13my07.shp",
        dst_datasource_name = "01_data/rv13my07_UTM11N.shp",
        s_srs = projection(CRS("+init=epsg:26912")),
        t_srs = projection(CRS("+init=epsg:26911")),
        overwrite = TRUE)
## character(0)
```

De la même façon, on le convertit aussi de la zone 12N à la zone 13N.

```
ogr2ogr(src_datasource_name = "01_data/rv13my07.shp",
        dst_datasource_name = "01_data/rv13my07_UTM13N.shp",
        s_srs = projection(CRS("+init=epsg:26912")),
        t_srs = projection(CRS("+init=epsg:26913")),
        overwrite = TRUE)
## character(0)
```

Au final, on obtient les deux nouvelles cartes présentées ci-dessous.



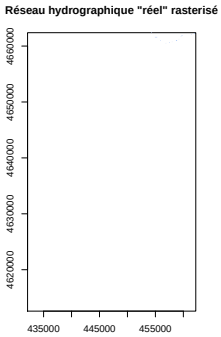
5.1.2.2 Rasterisation

Ici, on souhaite effectuer une rasterisation du réseau hydrographique. On réalise cette opération avec la fonction **gdal_rasterize()**. En entrée (**src_datasource**), on fournit le Shapefile du réseau hydrographique, et en sortie (**dst_filename**), on donne le nom du fichier raster. Ce dernier peut pré-exister ; dans le cas contraire, il sera créé par la fonction. On doit également identifier le nom de la couche (**l**) et la colonne de la table attributaire (**a**), dont les données seront assignées au futur raster. La colonne sélectionnée doit obligatoirement contenir des valeurs numériques. Si l'on s'intéresse à une variable qualitative, elle devra donc tout d'abord être convertie, afin d'être codée numériquement. Si l'on fournit plusieurs colonnes d'attributs, on créera alors un raster à plusieurs bandes. Si le fichier raster de sortie ne pré-existe pas, il conviendra de définir les caractéristiques du raster que l'on souhaite créer, à savoir : l'emprise géographique (**te**), la résolution (**tr**), ainsi que le système de coordonnées (**a_srs**). Dans le cas contraire, lorsque le fichier raster existe déjà, ce dernier doit être vide, c'est à dire qu'il ne

doit contenir que des valeurs vides ou des valeurs nulles (ou toute autre valeur que l'on ne pourra pas confondre avec une valeur de la table attributaire que l'on souhaite affecter à la matrice).

```
RAS_net <- gdal_rasterize(src_datasource = "01_data/rv13my07.shp",
                          dst_filename   = "01_data/rv13my07.tif",
                          a              = "PMILE",
                          l              = "rv13my07",
                          te             = c(RAS_mnt@extent@xmin, RAS_mnt@extent@ymin,
                                             RAS_mnt@extent@xmax, RAS_mnt@extent@ymax),
                          tr             = res(RAS_mnt),
                          a_srs          = projection(RAS_mnt),
                          ot             = "Int16",
                          verbose        = TRUE,
                          output_Raster  = TRUE)
print(RAS_net)
## class       : RasterBrick
## dimensions  : 1660, 985, 1635100, 1  (nrow, ncol, ncell, nlayers)
## resolution  : 30, 30  (x, y)
## extent      : 432404, 461954, 4612592, 4662392  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=12 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs
## data source : E:\Data\boulot\misc\scientific_doc\R\formation\irstea\033-geomatique_LIV\01_manuscrit_KNITR\03_par
## names       : rv13my07
## min values  :   -32768
## max values  :    32767
```

Attention, le module de rasterisation de GDAL/OGR, **gdal_rasterize**, présente un bug. En effet, ce module recalcule mal les valeurs minimale et maximale du raster.



5.1.2.3 Syntaxe des commandes

Étant donné que l'ensemble des fonctionnalités de GDAL/OGR n'ont pas été implémentées dans le package **gdalUtils**, pour accéder à celles-ci, il est nécessaire de passer par les propres lignes de commande de la bibliothèque.

5.1.2.3.a Assistance à l'écriture de commandes

Pour cela, le package **gdalUtils**, propose la fonction générique **gdal_cmd_builder()**, qui est une fonction d'aide à l'écriture des commandes GDAL/OGR. Comme nous allons le voir, l'inconvénient de cette fonction est qu'elle est très lourde à mettre en œuvre. Elle nécessite de définir les arguments suivants :

- **executable** : le nom de la fonction GDAL/OGR ;
- **parameter_variables** : la liste, organisée par type, de tous les paramètres de la fonction ;
- **parameter_values** : la liste des valeurs de paramètres que l'on souhaite définir ;
- **parameter_order** : le vecteur de tous les paramètres ordonnés ;
- **parameter_noflags** : le vecteur des paramètres qui ne présentent pas de *flag*.

Ci-dessous, nous allons présenter un exemple dans lequel on souhaite récupérer les isoclines du modèle numérique de terrain, et ce, tous les 500 m.

On commence par définir la commande GDAL/OGR que l'on souhaite utiliser, à savoir **gdal_contour**.

```
gdal_exe <- "gdal_contour"
```

On définit ensuite la liste de l'ensemble des paramètres de la commande et ce, par type de données, à savoir :

- **logical** : les booléens ;
- **vector** : les vecteurs numériques ;
- **scalar** : les scalaires numériques ;

– **character** : les chaînes de caractères ;
– **repeatable** : les paramètres multiples.

```
param_var <- list(logical    = list(c("3d","inodata")),
                  vector     = list(c("i")),
                  scalar     = list(c("snodata", "off")),
                  character  = list(c("a","f","fl","nln")),
                  repeatable = list(c("b")))
```

On crée ensuite le vecteur des noms de tous les paramètres, et ce, dans l'ordre, tel qu'il a été défini dans la documentation de la commande GDAL/OGR.

```
param_ord <- c("b", "a", "3d", "inodata", "snodata", "f", "i", "off", "fl", "nln",
               "src_filename", "dst_filename")
```

On crée ensuite le vecteur des noms de paramètres qui ne présentent pas de *flag* dans la ligne de commande GDAL/OGR (typiquement les noms des fichiers d'entrée et de sortie).

```
param_nof <- c("src_filename", "dst_filename")
```

Enfin, on définit la liste des valeurs des paramètres. L'ordre n'importe pas, et seuls les paramètres qui nous intéressent sont définis. Les noms des éléments de la liste doivent être ceux des noms des paramètres de la commande GDAL/OGR.

```
param_val <- list(src_filename = "01_data/logan_MNT.tif",
                  dst_filename = "01_data/logan_ISO.shp",
                  a = "elev",
                  i = 500)
```

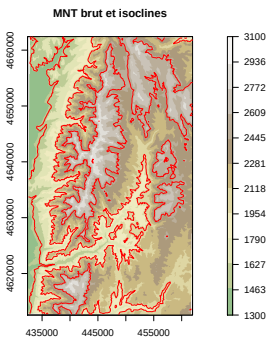
Une fois tous ces objets créés, on peut enfin faire appel à la fonction **gdal_cmd_builder()**. Celle-ci renvoie la commande GDAL/OGR sous la forme d'une chaîne de caractères.

```
gdal_cmd <- gdal_cmd_builder(executable          = gdal_exe,
                             parameter_variables = param_var,
                             parameter_values    = param_val,
                             parameter_order     = param_ord,
                             parameter_noflags   = param_nof)
gdal_cmd
## [1] "\"C:\\OSGeo4W64\\bin\\gdal_contour.exe\" -a \"elev\" -i \"500\" 01_data/logan_MNT.tif 01_data/logan_ISO.shp
```

Il ne reste plus qu'à lancer la commande à l'aide de la fonction **system()**.

```
system(gdal_cmd, intern = TRUE)
## [1] "0...10...20...30...40...50...60...70...80...90...100 - done."
```

Au final, on obtient les lignes de niveaux présentées sur la carte suivante.



5.1.2.3.b Écriture de commandes sans assistance

Comme on vient de le voir, l'utilisation de la fonction **gdal_cmd_builder()** est extrêmement verbeuse. On peut toutefois écrire soi-même la ligne de commande à moindres frais, même si toutes les vérifications ne seront pas réalisées. On peut ainsi très simplement définir une liste dont le premier élément représentera le chemin de la commande GDAL/OGR et les éléments suivants représenteront les valeurs des paramètres de cette commande. L'ordre des paramètres mentionnés dans la documentation de la commande GDAL/OGR devra impérativement être respecté. Par ailleurs, seuls les éléments définissant les paramètres présentant un *flag* dans la ligne de commande présenteront des noms, et ces derniers devront débuter par le signe moins ("**-**").

```
gdal_cmd_v <- list(paste(getOption("gdalUtils_gdalPath")[[1]]$path, "gdal_contour", sep = "/"),
                   '-a'  = "elev",
                   '-i'  = 500,
                   "01_data/logan_MNT.tif",
                   "01_data/logan_ISO.shp")
```

Il convient alors de concaténer les noms des éléments et leurs valeurs, afin de créer la ligne de commande. Les valeurs des paramètres devront être entourés de guillemets présentés sous une syntaxe compréhensible par la fonction **system()**.

```
gdal_cmd <- paste(names(gdal_cmd_v),
                  ifelse(is.na(gdal_cmd_v), '', qm(gdal_cmd_v)),
                  sep = " ", collapse = " ")
gdal_cmd
## [1] " \"C:/OSGEO4~1/bin//gdal_contour\" -a \"elev\" -i \"500\"  \"01_data/logan_MNT.tif\"  \"01_data/logan_ISO.s
```

Une fois cette ligne de commande rédigée, comme précédemment, il suffit de la lancer avec la fonction **system()**.

```
system(gdal_cmd, intern = TRUE)
## [1] "0...10...20...30...40...50...60...70...80...90...100 - done."
```

5.2 Logiciel GRASS GIS

GRASS GIS (*Geographic Resources Analysis Support System*) (GRASS Development Team, 2015) est un logiciel de système d'information géographique libre, développé depuis 1982 par le *Construction Engineering Research Laboratory* de l'U.S. Army, puis repris, en 1999, par le *GRASS Development Team*, une équipe internationale de développeurs. La première version stable date de 1984. (Wikipedia, 2015c.)

GRASS GIS est de conception modulaire, c'est-à-dire qu'à chaque fonction du logiciel correspond un module, ce qui permet d'économiser la mémoire et la CPU de l'ordinateur en ne lançant que les modules dont l'utilisateur a besoin. Ces modules sont regroupés en familles qui sont identifiables par leurs préfixes respectifs :

- **g.*** : fonctions générales (manipulation/suppression/renommage de fichiers ou paramétrage du secteur, de la région ou du jeu de données, etc.) ;
- **i.*** : fonctions de traitement d'images ;
- **r.*** : fonctions de traitement de rasters ;
- **r3.*** : fonctions de traitement de rasters tri-dimensionnels ;
- **v.*** : fonctions de traitement de vecteurs ;
- **db.*** : fonctions ayant trait aux bases de données ;
- **pg.*** : fonctions ayant trait aux bases de données Postgres ;
- **d.*** : fonctions d'affichage ;
- **p.*** : production de cartes ;
- **ps.*** : production de cartes Postscript ;
- **m.*** : fonctions diverses.

GRASS GIS accepte d'importer une multitude de formats propriétaires ou d'échange, et il peut également exporter une grande variété de formats. Ceci est rendu possible grâce à son interface avec les bibliothèques autonomes GDAL et OGR.

Plusieurs packages permettent l'établissement d'une communication entre R et différentes versions du logiciel GRASS GIS. Le package **spgrass6** (Bivand, 2007 ; Bivand *et al.*, 2008 ; Bivand, 2014) permet d'établir la connexion entre R et la version 6 de GRASS GIS. Il est également compatible avec la version 7, mais pour travailler avec celle-ci, il est préférable d'utiliser le package **rgrass7** (Bivand, 2015). Le package **GRASS**, quant à lui, permettait d'établir la connexion avec la version 5, mais il n'est plus maintenu ; il est cependant toujours disponible dans les archives du CRAN [5]. L'exemple ci-après présente le package **rgrass7**, dont les fonctions sont extrêmement proches de celles du package **spgrass6**.

```
library(rgrass7)
```

5.2.1 Préparation des données

Avant toute chose, il convient de définir quelques objets nécessaires au fonctionnement de GRASS GIS :

5. http://cran.r-project.org/src/contrib/Archive/GRASS/.

- **location** : le secteur, pour lequel sont définis le système de géoréférencement choisi pour le projet ou la résolution des fichiers rasters ;
- **region** : la région, c'est-à-dire la vue du projet, mais aussi la zone sur laquelle porteront les calculs, les analyses, ou les exportations ;
- **mapset** : le jeu de données, correspondant au répertoire de travail de l'utilisateur principal, sur lequel il a droit de lecture et d'écriture, tandis que les autres utilisateurs ont le seul droit de lecture.

5.2.1.1 Création de la base de données

La première étape consiste à créer un projet GRASS GIS. Il faut donc définir :
- le répertoire où GRASS GIS stockera les fichiers importés et/ou créés ;
- le système de coordonnées dans lequel l'utilisateur principal souhaite travailler ;
- l'étendue géographique du projet (qui concerne l'ensemble des fichiers qui seront importés par la suite par l'utilisateur).

Avant d'importer les données, on commence donc par créer la base de données GRASS GIS. Pour cela, on indique à la fonction **initGRASS()** dans quel répertoire est installé le logiciel GRASS GIS (**gisBase**). On lui définit aussi le nom du répertoire qui stockera des données temporaires (**home**), ainsi que celui qui contiendra la base de données (**gisDbase**). Sous Windows, il est possible d'appeler GRASS GIS depuis son emplacement propre, ou bien depuis OSGeo4W, ou encore depuis QGIS.

```
LOC_grs <- initGRASS(gisBase  = "C:/Prog/GRASS_GIS/7.0.1",
                     home     = tempdir(),
                     gisDbase = "01_data/GRASS_BD",
                     override = TRUE, use_g.dirseps.exe = TRUE)
```

5.2.1.2 Syntaxe des commandes

De manière générale, avec les packages **spgrass6** et **rgrass7**, on appelle chaque fonctionnalité de GRASS GIS *via* la fonction générique **execGRASS()**. Pour cela, on fournit à cette fonction le nom du module GRASS GIS que l'on souhaite utiliser, les différents paramètres de ce module, ainsi que les *flags*, qui correspondent à des paramètres relatifs au mode d'exécution (i.e. autorisation de réécriture du fichier, affichage explicite des opérations dans la console, etc.).

5.2.1.3 Définition du projet

On souhaite travailler en NAD83 / UTM zone 12N. Pour cela, on utilise le module **g.proj** avec **flags = "c"** pour créer un nouveau projet en définissant le code EPSG de la projection, et l'on déplace notre espace de travail vers celui-ci à l'aide du module **g.mapset**. On peut vérifier le résultat à chaque étape en affichant le système de coordonnées dans la console, en utilisant le module **g.proj** avec **flags = "g"**.

```
execGRASS("g.proj"  , flags = c("c"),
                      parameters = list(epsg = 26912, location = "logan_PROJ"))
execGRASS("g.proj"  , flags = c("g", "verbose"))
execGRASS("g.mapset", flags = c("c"),
                      parameters = list(mapset = "PERMANENT", location = "logan_PROJ"))
execGRASS("g.proj"  , flags = c("g", "verbose"))
```

5.2.1.4 Import des données dans la base

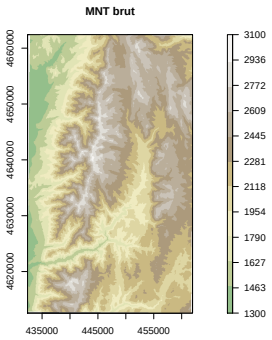
Une des particularités du logiciel GRASS GIS est qu'il ne travaille qu'avec son propre format de données. On doit donc toujours convertir les données au format adéquat. Par ailleurs, comme nous venons de le voir, les données doivent impérativement se trouver dans une base présentant une structure bien définie, spécifique à GRASS GIS.

Maintenant que la base de données est définie, on peut donc importer les fichiers géographiques dans cette dernière. On souhaite travailler avec le modèle numérique de terrain que nous possédons au format GeoTIFF. L'importation d'un raster se fait grâce au module **r.in.gdal**.

```
execGRASS("r.in.gdal", flags = c("o", "overwrite"),
          parameters = list(input = "01_data/logan_MNT.tif", output = "logan_MNT"))
```

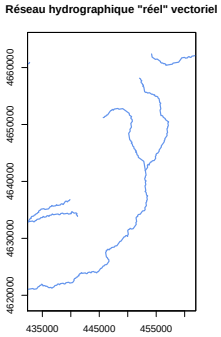
Il est indispensable de définir l'emprise géographique du projet sur lequel on travaille avec le module **g.region**. Dans notre exemple, il sera le même que celui de notre MNT.

```
execGRASS("g.region", parameters = list(raster = "logan_MNT"))
```



Par ailleurs, on souhaite importer un réseau hydrographique observé, qui est sous forme vectorielle. Pour cela, on utilise le module **`v.in.ogr`**.

```
execGRASS("v.in.ogr", flags = c("verbose", "r", "o", "overwrite"),
          parameters = list(input = "01_data/rv13my07.shp", output = "rv13my07"))
```



De la même manière, on peut aussi importer la position de l'exutoire du bassin que l'on souhaite définir.

```
execGRASS("v.in.ogr", flags = c("verbose", "r", "o", "overwrite"),
          parameters = list(input = "01_data/exu_POS.shp", output = "exu_POS"))
```

Maintenant que la base de données GRASS GIS est créée et que les données sont prêtes, on peut réaliser les traitements de géomatique.

5.2.2 Traitements des données

Nous allons réaliser ici les traitements sur le modèle numérique de terrain, en vue de la détermination du contour d'un bassin versant.

5.2.2.1 Correction du fichier d'élévation et calcul de la direction d'écoulement

On souhaite effectuer une correction du fichier d'élévation, afin de combler les éventuelles cuvettes qui pourraient exister. Le module **`r.fill.dir`** permet de réaliser une correction automatique des cuvettes, sans présager de leur existence réelle ou non. Des options permettent d'importer, le cas échéant, un fichier contenant des cuvettes réelles (e.g. des lacs). On considérera ici que les cuvettes présentes, suite au traitement du MNT, n'ont pas d'existence réelle et sont donc liées à la résolution du MNT. Le module **`r.fill.dir`** renvoie un nouvel MNT nettoyé de ses cuvettes, ainsi qu'un raster de direction d'écoulement.

Attention, même si le module **`r.fill.dir`** effectue automatiquement plusieurs "passages" afin de combler les cuvettes, il est nécessaire de relancer la fonction autant de fois que nécessaire, afin qu'il ne subsiste aucune dépression à la fin de l'opération. À chaque itération, il faudra bien penser à fournir en entrée le nom du fichier de sortie obtenu au calcul précédent. L'option **`format = "agnps"`** permet d'obtenir le raster de direction d'écoulement avec des valeurs codées de 1 à 8 et non en degrés. Attention, ce n'est le raster qui devra être utilisé pour la déterminer des contours de bassins versants.

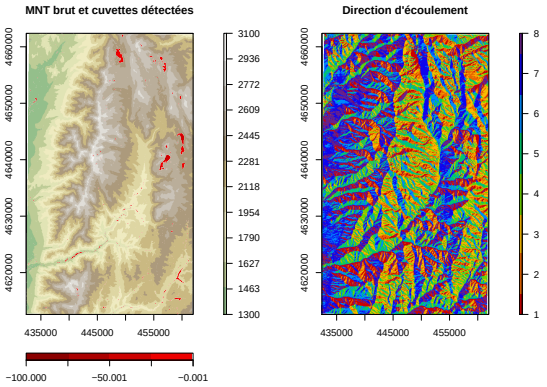
On effectue ici le premier passage.

```
execGRASS("r.fill.dir", flags = c("verbose", "overwrite"),
          parameters = list(input     = "logan_MNT",
                            output    = "logan_FIL",
                            direction = "logan_DIR",
                            format    = "agnps"))
```

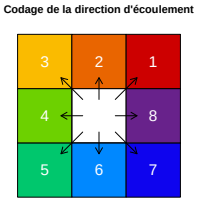
On récupère le MNT partiellement comblé et on le soumet à nouveau au même traitement. Ainsi a-t-on besoin de réaliser encore deux itérations, afin qu'il ne subsiste plus la moindre dépression.

```
replicate(2, execGRASS("r.fill.dir", flags = c("verbose", "overwrite"),
                       parameters = list(input     = "logan_FIL",
                                         output    = "logan_FIL",
                                         direction = "logan_DIR",
                                         format    = "agnps")))
## [1] 0 0
```

Une fois que l'on s'est assuré d'avoir réalisé suffisamment d'itérations, on peut récupérer le MNT exempt de cuvette, ainsi que le raster de direction d'écoulement.



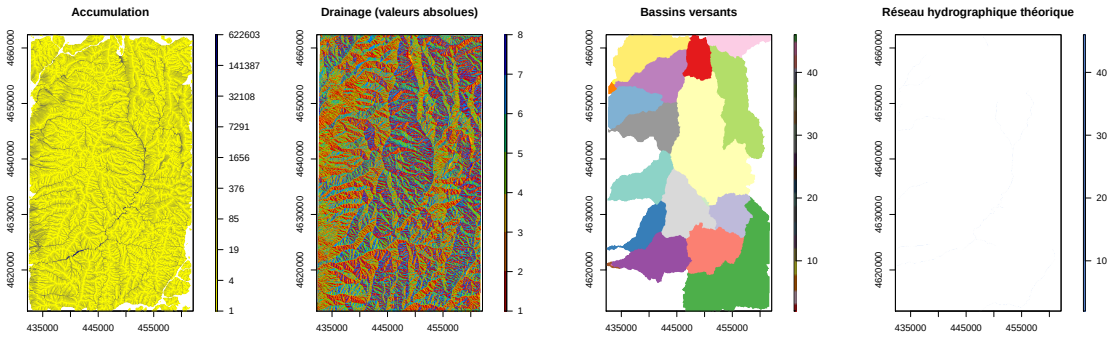
Le module **`r.fill.dir`** code la direction d'écoulement D8, comme présentée ci-dessous.



5.2.2.2 Détermination de l'accumulation d'écoulement, du drainage et du réseau hydrographique théorique

Une fois cette étape terminée, on dispose donc d'un modèle numérique de terrain topologiquement propre, c'est-à-dire sans cuvette, et qui est donc utilisable pour tracer des contours de bassin versant. On peut donc calculer le raster d'accumulation et le raster de drainage (direction d'écoulement D8) en fournissant le MNT comblé au module **`r.watershed`**. Noté que l'écoulement n'est pas le même que celui renvoyé par le module **`r.fill.dir`**. Par ailleurs, d'autres différences existent entre le raster renvoyé par ces différents modules : avec**`r.watershed`**, la valeur zéro indique que la cellule est une zone de dépression (définie la carte de dépression éventuellement fournie en entrée), et les valeurs négatives indiquent que le ruissellement de surface quitte les limites de la région géographique actuelle (la valeur absolue de ces cellules négatives indiquent la direction d'écoulement). C'est le raster produit par **`r.watershed`** qui devra être utilisé pour la déterminer des contours de bassins versants. On peut également déterminer des bassins versants en choisissant un seuil correspondant à une superficie minimale. Le raster du réseau hydrographique théorique obtenu présentera des valeurs correspondant aux surfaces des bassins.

```
execGRASS("r.watershed", flags = c("overwrite"),
          parameters = list(elevation    = "logan_FIL",
                            accumulation = "logan_ACC",
                            drainage     = "logan_DRA",
                            threshold    = 30000,
                            basin        = "logan_CAT",
                            stream       = "logan_NET"))
```



Si on le souhaite, le réseau hydrographique théorique, ainsi créé sous forme matricielle, pourra être vectorisé. Pour ce faire, il faudra utiliser le module **`r.to.vect`** après l'avoir toutefois aminci avec le module **`r.thin`**, afin qu'il n'excède pas un pixel de large.

5.2.2.3 Délimitation du bassin versant

Pour pouvoir déterminer le contour du bassin versant, il convient de s'assurer que l'exutoire qui nous intéresse est bien positionné sur le réseau hydrographique théorique. On peut réaliser automatiquement le repositionnement à l'aide du module **`r.stream.snap`** (il s'agit d'un plugin qui doit être préalablement installé). Ce module permet de snaper un point sur une grille. Dans notre cas, on se sert du raster du réseau hydrographique et du raster d'accumulation, et l'on recherche la position de l'exutoire dans un périmètre défini par l'argument **`radius`** (exprimé en nombre de pixels) et selon un seuil de surface maximale défini par l'argument **`threshold`** (exprimé dans le carré de l'unité de la carte).

```
execGRASS("r.stream.snap", flags = c("verbose", "overwrite"),
          parameters = list(input        = "exu_POS",
                            stream_rast  = "logan_NET",
                            accumulation = "logan_ACC",
                            radius       = 10,
                            threshold    = 10000,
                            output       = "exu_POS_MOV"))
```

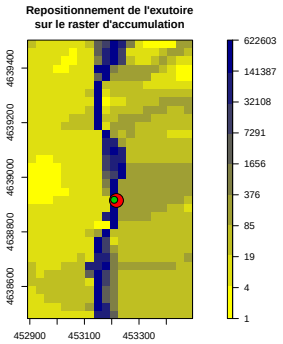
Utilisation de plugins GRASS GIS sous R

Sous le système d'exploitation Windows, les plugins ne s'installent pas dans le même répertoire que les binaires de base de GRASS GIS (/GRASS_GIS/7.0.1/bin), mais dans un répertoire spécifique ("/Users/prenom.nom/AppData/Roaming/GRASS7/addons/bin"). Pour que R puisse accéder aux plugins, il faut alors placer les exécutables de ces derniers parmi les binaires de base. Sous Linux, il n'y a pas de problème particulier.

Après repositionnement, les nouvelles coordonnées sont les suivantes :

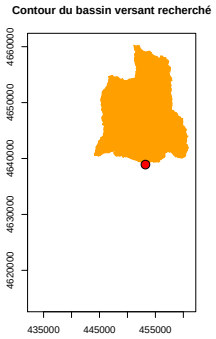
```
coordinates(EXU_pos_mov)
##      coords.x1 coords.x2 coords.x3
## [1,]    453209   4638918         0
```

Sur la carte suivante, on peut apprécier l'ampleur du déplacement de l'exutoire, depuis sa position d'origine (en rouge), vers sa nouvelle position sur le réseau hydrographique théorique (en vert).



Le module **r.water.outlet** permet de déterminer les limites d'un bassin versant à partir du couple de coordonnées de son exutoire. Au préalable, il faut bien évidemment veiller à ce que l'exutoire se situe sur le réseau de drainage théorique déterminé à partir du MNT nettoyé de ses cuvettes, comme nous venons de le faire.

```
execGRASS("r.water.outlet", flags = c("verbose", "overwrite"),
          parameters = list(input       = "logan_DRA",
                            output      = "logan_CBS",
                            coordinates = coordinates(EXU_pos_mov)[1:2]))
```



Le bassin versant ainsi obtenu présente une superficie d'environ 230.19 km^2.

5.2.3 Import des données GRASS GIS dans R

5.2.3.1 Données matricielles

Pour importer des données matricielles au format GRASS GIS dans l'environnement de R, il faut utiliser la fonction **readRAST()** (**readRAST6()** avec le package **spgrass6**). Cette dernière renvoie un objet matriciel de classe **sp** (**SpatialGridDataFrame**).

```
RAS_mnt <- readRAST("logan_MNT")
class(RAS_mnt)
## [1] "SpatialGridDataFrame"
## attr(,"package")
## [1] "sp"
```

Pour davantage de facilités de manipulation, on peut convertir les objets matriciels de classe **SpatialGridDataFrame** en des objets de classe **rasterLayer** ; cette opération peut être assurée par la fonction **raster()** du package du même nom.

```
RAS_mnt <- raster(readRAST("logan_MNT"))
print(RAS_mnt)
## class       : RasterLayer
## dimensions  : 1660, 985, 1635100  (nrow, ncol, ncell)
## resolution  : 30, 30  (x, y)
## extent      : 432404, 461954, 4612592, 4662392  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +no_defs +zone=12 +a=6378137 +rf=298.257222101 +towgs84=0.000,0.000,0.000 +to_meter=1
## data source : in memory
## names       : logan_MNT
## values      : 1358.335, 3031.443  (min, max)
```

On peut importer simultanément plusieurs rasters de mêmes caractéristiques et ce, dans un seul et unique objet.

```
RAS_wat <- readRAST(c("logan_FIL", "logan_ACC", "logan_DRA", "logan_CAT"))
nlayers(RAS_wat)
## [1] 4
```

Pour plus de facilité de manipulation, on peut convertir les objets matriciels multiples de classe **SpatialGridDataFrame** en des objets de classe **brick** ou **stack**, et ce à l'aide des fonctions **brick()** et **stack()** du package **raster**.

```
RAS_wat <- brick(RAS_wat)
print(RAS_wat)
## class       : RasterBrick
## dimensions  : 1660, 985, 1635100, 4  (nrow, ncol, ncell, nlayers)
## resolution  : 30, 30  (x, y)
## extent      : 432404, 461954, 4612592, 4662392  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +no_defs +zone=12 +a=6378137 +rf=298.257222101 +towgs84=0.000,0.000,0.000 +to_meter=1
## data source : in memory
## names       :   logan_FIL,   logan_ACC,   logan_DRA,   logan_CAT
## min values  :    1358.743, -623867.348,      -8.000,       2.000
## max values  :    3031.443,  622602.527,       8.000,      46.000
```

5.2.3.2 Données vectorielles

Pour importer des données vectorielles au format GRASS GIS dans l'environnement de R, il faut utiliser la fonction **readVECT()** (**readVECT6()** avec le package **spgrass6**). Cette dernière renvoie un objet vectoriel de classe **sp** (**SpatialPointsDataFrame**, **SpatialLinesDataFrame** ou **SpatialPolygonsDataFrame** selon les cas).

```
VEC_net <- readVECT("rv13my07")
## OGR data source with driver: SQLite
## Source: "01_data/GRASS_BD/logan_PROJ/PERMANENT/.tmp/unknown/366.0", layer: "rv13my07"
## with 8 features
## It has 18 fields
print(VEC_net)
## class       : SpatialLinesDataFrame
## features    : 8
## extent      : 432404, 461954, 4621037, 4662392  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=12 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs
## variables   : 18
## names       : cat, IHABBSRF_I,          RR,      HUC, TYPE, PMILE,     PNAME, OWNAME,       PNMCD, OWNMCD,
## min values  :   1,      57252, 16010201019, 16010201,    R,  78.5, BEAVER CR,      0, 16010201009,       , 1601
## max values  :   8,      57608, 16010203015, 16010203,    S, 237.2, SUMMIT CR,      0, 16010203008,       , 1601
```

5.3 Logiciel SAGA GIS

SAGA GIS (*System for Automated Geoscientific Analyses*) (SAGA Development Team, 2008) est un logiciel multiplate-forme, publié sous licence GPL. Il était à l'origine, en 2001, développé par une équipe du département de géographie physique de l'université de Göttingen, mais il est maintenant maintenu par une équipe internationale, dont le noyau est situé à l'université de Hambourg. La première version stable date de 2004. SAGA GIS est un outil destiné aux traitements spatiaux aussi bien de type matriciel que vectoriel. (Wikipedia, 2015d.)

Le logiciel est composé de plusieurs bibliothèques, ces dernières contenant différents modules.

La connexion entre R et SAGA GIS est assurée par le package **RSAGA** (Brenning, 2008).

```
library(RSAGA)
```

5.3.1 Préparation des données

5.3.1.1 Définition de l'environnement de travail

En préalable à tout traitement de données, il convient de définir l'environnement de travail permettant à R d'établir la connexion avec SAGA GIS. Pour cela, on utilise la fonction **rsaga.env()**. L'argument **workspace** corresond à l'espace de travail, c'est-à-dire au répertoire dans lequel SAGA GIS va exporter ses résultats. L'argument **path** correspond au chemin de l'application SAGA GIS.

```
ENV_sag <- rsaga.env(workspace = ".", path = "C:/OSGeo4W64/apps/saga")
ENV_sag
## $workspace
## [1] "."
##
## $cmd
## [1] "saga_cmd.exe"
##
## $path
## [1] "C:/OSGeo4W64/apps/saga"
##
## $modules
## [1] "C:/OSGeo4W64/apps/saga/modules"
##
## $version
## [1] "2.1.2"
##
## $cores
## [1] NA
##
## $parallel
## [1] FALSE
##
## $lib.prefix
## [1] ""
```

Il est possible de paralléliser certains fonctionnalités, de deux manières indépendantes et ce, en utilisant deux arguments distincts de la fonction **rsaga.env()** :

- **cores** : nombre de cœurs utilisés par certains modules du logiciel SAGA GIS ;
- **parallel** : parallélise certaines fonctions du package **RSAGA** (e.g. **pick.from.ascii.grid()** ou **rsaga.get.modules()**).

Par ailleurs, le chemin défini par défaut est renvoyé par la fonction **rsaga.default.path()**, et peut éventuellement permettre de définir l'argument **path** de la fonction **rsaga.env()**.

```
rsaga.default.path()
## [1] "C:/Progra~1/SAGA-GIS"
```

5.3.1.2 Syntaxe des commandes

Avant toute chose, il convient de mentionner que, dans SAGA GIS, chaque bibliothèque est identifiée par un nom. Les modules, quant à eux, sont identifiés par un nom et un code, ce dernier se présentant sous la forme d'un numéro.

5.3.1.2.a Liste des bibliothèques et des modules disponibles

La fonction **rsaga.get.libraries()** liste les différentes bibliothèques disponibles.

```
rsaga.get.libraries(path = ENV_sag$modules)
##  [1] "climate_tools"           "contrib_perego"          "db_odbc"
##  [4] "db_pgsql"                "docs_html"               "docs_pdf"
##  [7] "garden_3d_viewer"        "garden_fractals"         "garden_games"
## [10] "garden_learn_to_program" "garden_webservices"      "grid_analysis"
## [13] "grid_calculus"           "grid_calculus_bsl"       "grid_filter"
## [16] "grid_gridding"           "grid_spline"             "grid_tools"
## [19] "grid_visualisation"      "imagery_classification"  "imagery_opencv"
## [22] "imagery_photogrammetry"  "imagery_rga"             "imagery_segmentation"
## [25] "imagery_svm"             "imagery_tools"           "imagery_vigra"
## [28] "io_esri_e00"             "io_gdal"                 "io_gps"
## [31] "io_grid"                 "io_grid_grib2"           "io_grid_image"
## [34] "io_shapes"               "io_shapes_dxf"           "io_shapes_las"
## [37] "io_table"                "io_virtual"              "pj_georeference"
## [40] "pj_geotrans"             "pj_proj4"                "pointcloud_tools"
## [43] "pointcloud_viewer"       "shapes_grid"             "shapes_lines"
## [46] "shapes_points"           "shapes_polygons"         "shapes_tools"
## [49] "shapes_transect"         "sim_cellular_automata"   "sim_ecosystems_hugget"
## [52] "sim_erosion"             "sim_fire_spreading"      "sim_hydrology"
## [55] "sim_ihacres"             "statistics_grid"         "statistics_kriging"
## [58] "statistics_points"       "statistics_regression"   "ta_channels"
## [61] "ta_compound"             "ta_hydrology"            "ta_lighting"
## [64] "ta_morphometry"          "ta_preprocessor"         "ta_profiles"
## [67] "ta_slope_stability"      "table_calculus"          "table_tools"
## [70] "tin_tools"               "tin_viewer"
```

Les fonctions **rsaga.get.lib.modules()** et **rsaga.get.modules()** listent les différents modules disponibles au sein d'une bibliothèque donnée. Ceci permet de connaître les codes des modules, ainsi que leurs noms (qui correspondent souvent à un court descriptif).

```
rsaga.get.modules(lib = "pj_proj4", env = ENV_sag)
## $pj_proj4
##    code                                                  name interactive
## 1     0                        Set Coordinate Reference System       FALSE
## 2     1                 Coordinate Transformation (Shapes List)       FALSE
## 3     2                      Coordinate Transformation (Shapes)       FALSE
## 4     3                   Coordinate Transformation (Grid List)       FALSE
## 5     4                        Coordinate Transformation (Grid)       FALSE
## 6     5                 Proj.4 (Command Line Arguments, Shapes)       FALSE
## 7     6                                Proj.4 (Dialog, Shapes)       FALSE
## 8     7                   Proj.4 (Command Line Arguments, Grid)       FALSE
## 9     8                                  Proj.4 (Dialog, Grid)       FALSE
## 10    9 Proj.4 (Command Line Arguments, List of Shapes Layers)       FALSE
## 11   10                  Proj.4 (Dialog, List of Shapes Layers)       FALSE
## 12   11          Proj.4 (Command Line Arguments, List of Grids)       FALSE
## 13   12                         Proj.4 (Dialog, List of Grids)       FALSE
## 14   13                     Change Longitudinal Range for Grids       FALSE
## 15   14                           Latitude/Longitude Graticule       FALSE
## 16   15                       Coordinate Reference System Picker       FALSE
## 17   16                                  Tissot's Indicatrix       FALSE
```

5.3.1.2.b Consultation de la documentation

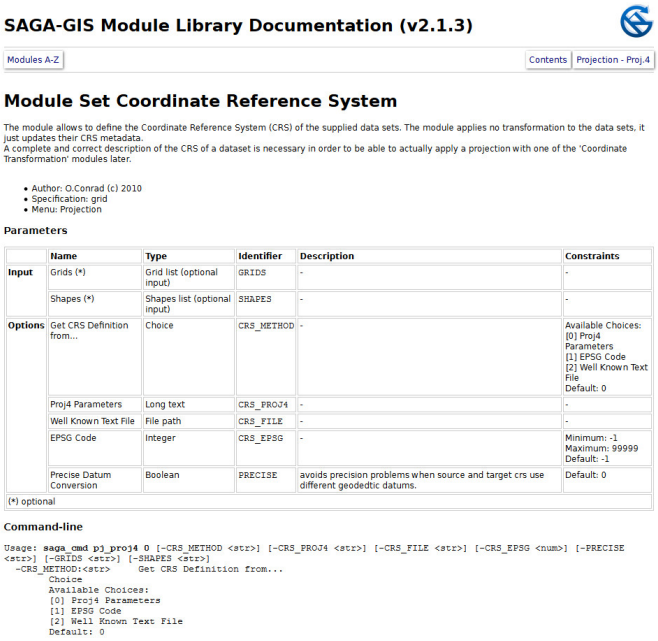
La fonction **rsaga.get.usage()** fournit des informations sur l'utilisation d'un module et de ses arguments.

```
rsaga.get.usage(lib = "pj_proj4", module = 0, env = ENV_sag)
## library path: C:\OSGEO4~1\apps\saga\modules\pj_proj4.dll
## library name: Projection - Proj.4
## Usage: saga_cmd pj_proj4 0 [-CRS_METHOD <str>] [-CRS_PROJ4 <str>] [-CRS_FILE <str>] [-CRS_EPSG <num>] [-PRECISE
##   -CRS_METHOD:<str>Get CRS Definition from...
##  Choice
```

```
##  Available Choices:
##  [0] Proj4 Parameters
##  [1] EPSG Code
##  [2] Well Known Text File
##  Default: 0
##   -CRS_PROJ4:<str>  Proj4 Parameters
##  Long text
##   -CRS_FILE:<str>   Well Known Text File
##  File path
##   -CRS_EPSG:<num>   EPSG Code
##  Integer
##  Minimum: -1
##  Maximum: 99999
##  Default: -1
##   -PRECISE          Precise Datum Conversion
##  Boolean
##  Default: 0
##   -GRIDS:<str>      Grids
##  Grid list (optional input)
##   -SHAPES:<str>     Shapes
##  Shapes list (optional input)
```

La fonction **`rsaga.html.help()`** renvoie l'aide d'une bibliothèque et de ses modules.

```
rsaga.html.help(lib = "pj_proj4", module = 0, env = ENV_sag)
```



SAGA-GIS Module Library Documentation (v2.1.3)

Modules A-Z | Contents | Projection - Proj.4

Module Set Coordinate Reference System

The module allows to define the Coordinate Reference System (CRS) of the supplied data sets. The module applies no transformation to the data sets, it just updates their CRS metadata.
A complete and correct description of the CRS of a dataset is necessary in order to be able to actually apply a projection with one of the 'Coordinate Transformation' modules later.

- Author: O.Conrad (c) 2010
- Specification: grid
- Menu: Projection

Parameters

	Name	Type	Identifier	Description	Constraints
Input	Grids (*)	Grid list (optional input)	GRIDS	-	-
	Shapes (*)	Shapes list (optional input)	SHAPES	-	-
Options	Get CRS Definition from...	Choice	CRS_METHOD	-	Available Choices: [0] Proj4 Parameters [1] EPSG Code [2] Well Known Text File Default: 0
	Proj4 Parameters	Long text	CRS_PROJ4	-	-
	Well Known Text File	File path	CRS_FILE	-	-
	EPSG Code	Integer	CRS_EPSG	-	Minimum: -1 Maximum: 99999 Default: -1
	Precise Datum Conversion	Boolean	PRECISE	avoids precision problems when source and target crs use different geodedtic datums.	Default: 0

(*) optional

Command-line

```
Usage: saga_cmd pj_proj4 0 [-CRS_METHOD <str>] [-CRS_PROJ4 <str>] [-CRS_FILE <str>] [-CRS_EPSG <num>] [-PRECISE
<str>] [-GRIDS <str>] [-SHAPES <str>]
  -CRS_METHOD:<str>     Get CRS Definition from...
        Choice
        Available Choices:
        [0] Proj4 Parameters
        [1] EPSG Code
        [2] Well Known Text File
        Default: 0
```

5.3.1.2.c Recherche de bibliothèque ou de module

On peut effectuer une recherche par mots clés sur les noms de bibliothèques ou de modules, grâce à la fonction **`rsaga.search.modules()`**.

```
rsaga.search.modules("gap", env = ENV_sag)
```

```
## $lib
## character(0)
##
## $modules
##              lib                                module
## 1     grid_tools                   Close One Cell Gaps
## 2     grid_tools                               Close Gaps
## 3     grid_tools                Close Gaps with Spline
## 4     grid_tools Close Gaps with Stepwise Resampling
## 5 table_calculus                  Fill Gaps in Records
```

5.3.1.2.d Fonction générique

Il existe une fonction générique permettant d'appeler n'importe quel module de n'importe quelle bibliothèque ; il s'agit de **`rsaga.geoprocessor()`**. Pour cela, on fournit à cette fonction le nom du module SAGA GIS que l'on souhaite utiliser, ainsi que les différents paramètres de celui-ci. Nous allons donner des exemples de son utilisation ci-après. Par ailleurs, le package **`RSAGA`** propose plusieurs fonctions correspondant à des implémentations directes de modules SAGA GIS.

5.3.1.3 Import des données dans l'environnement

5.3.1.3.a Données matricielles

SAGA GIS travaille avec son propre format de données matricielles. On doit donc toujours convertir les données au format adéquat. L'import de ce type de données est réalisé *via* un appel à la fonction **`rsaga.import.gdal()`**. On importe ici notre MNT au format GeoTIFF pour le convertir au format SAGA GIS (extension ".sgrd" par défaut).

```
rsaga.import.gdal(in.grid  = "01_data/logan_MNT.tif",
                  out.grid = "01_data/logan_MNT.sgrd",
                  env      = ENV_sag)
```

```
##
##
## library path: C:\OSGEO4~1\apps\saga\modules\io_gdal.dll
## library name: Import/Export - GDAL/OGR
## tool name   : GDAL: Import Raster
## author      : O.Conrad (c) 2007 (A.Ringeler)
##
##
##
## Parameters
##
##
## Grids: No objects
## Files: "01_data/logan_MNT.tif"
## Transformation: no
## Interpolation: B-Spline Interpolation
##
##
## loading: 01_data/logan_MNT.tif
##
##
##
## Driver: GTiff
##
## Bands: 1
##
## Rows: 985
##
## Columns: 1660
##
##
##
## Transformation:
##
##   x' = 432404.019091 + x * 30.000000 + y * 0.000000
##
##   y' = 4662392.446916 + x * 0.000000 + y * -30.000000
##
##
## loading band [1/1]
## Save grid: 01_data/logan_MNT.sgrd...
```

Il est possible de réaliser la même opération en utilisant la fonction générique **`rsaga.geoprocessor()`**, et en faisant appel à la bibliothèque **`io_gdal`** et à son module **`0`** (**`GDAL: Import Raster`**).

```
rsaga.geoprocessor(lib = "io_gdal", module = 0, env = ENV_sag,
                   param = list(FILES = "01_data/logan_MNT.tif",
                                GRIDS = "01_data/logan_MNT.sgrd"))
```

```
##
##
## library path: C:\OSGEO4~1\apps\saga\modules\io_gdal.dll
## library name: Import/Export - GDAL/OGR
## tool name   : GDAL: Import Raster
## author      : O.Conrad (c) 2007 (A.Ringeler)
##
##
##
## Parameters
##
##
## Grids: No objects
## Files: "01_data/logan_MNT.tif"
## Transformation: no
## Interpolation: B-Spline Interpolation
##
##
## loading: 01_data/logan_MNT.tif
##
##
##
## Driver: GTiff
##
## Bands: 1
##
## Rows: 985
##
## Columns: 1660
```

```
##
##
##
## Transformation:
##
##   x' = 432404.019091 + x * 30.000000 + y * 0.000000
##
##   y' = 4662392.446916 + x * 0.000000 + y * -30.000000
##
##
## loading band [1/1]
## Save grid: 01_data/logan_MNT.sgrd...
```

En réalité, la fonction **rsaga.import.gdal()** ne fait pas autre chose que d'appeler cette fonctionnalité de SAGA GIS.

```
rsaga.import.gdal
## function (in.grid, out.grid, env = rsaga.env(), ...)
## {
##     if (missing(out.grid)) {
##         out.grid = set.file.extension(in.grid, "")
##         out.grid = substr(out.grid, 1, nchar(out.grid) - 1)
##     }
##     if (env$version == "2.0.4") {
##         param = list(GRIDS = out.grid, FILE = in.grid)
##     }
##     else {
##         param = list(GRIDS = out.grid, FILES = in.grid)
##     }
##     module = "GDAL: Import Raster"
##     if (env$version == "2.2.3") {
##         module = "Import Raster"
##     }
##     rsaga.geoprocessor("io_gdal", module = module, param = param,
##         env = env, ...)
## }
## <environment: namespace:RSAGA>
```

5.3.1.3.b Données vectorielles

Concernant les données vectorielles, SAGA GIS travaille directement avec des fichiers au format Shapefile. S'agissant du format de données vectorielles le plus courant, la majorité du temps, aucune manipulation particulière n'est nécessaire. Dans le cas contraire, il faudra lire les données avec les modules de la bibliothèques **io_shapes**.

5.3.2 Traitements des données

Nous allons réaliser ici les traitements sur le modèle numérique de terrain, en vue de la détermination du contour d'un bassin versant.

5.3.2.1 Correction du fichier d'élévation

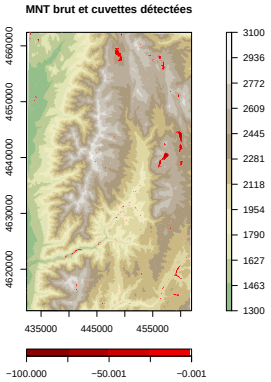
On souhaite effectuer une correction du fichier d'élévation en supprimant les éventuelles dépressions. La bibliothèque **ta_preprocessor** propose plusieurs algorithmes permettant de combler les cuvettes.

```
rsaga.get.modules("ta_preprocessor", env = ENV_sag)
## $ta_preprocessor
##   code                                name interactive
## 1    0                        Flat Detection       FALSE
## 2    1        Sink Drainage Route Detection       FALSE
## 3    2                          Sink Removal       FALSE
## 4    3 Fill Sinks (Planchon/Darboux, 2001)       FALSE
## 5    4              Fill Sinks (Wang & Liu)       FALSE
## 6    5          Fill Sinks XXL (Wang & Liu)       FALSE
## 7    6         Burn Stream Network into DEM       FALSE
```

Il est possible de prendre en compte l'existence de cuvettes réelles, c'est notamment ce que propose le module **0** (**Flat Detection**). Dans le cadre de notre exemple, comme nous ne disposons pas de telles données, nous allons utiliser le module **3** (**Fill Sinks (Planchon/Darboux, 2001)**), qui ne requiert que le choix d'une pente minimale.

```
rsaga.geoprocessor(lib = "ta_preprocessor", module = 3, env = ENV_sag,
                   param = list(DEM      = "01_data/logan_MNT.sgrd",
                                RESULT   = "01_data/logan_FIL.sgrd",
                                MINSLOPE = 0.01))
##
##
## library path: C:\OSGEO4~1\apps\saga\modules\ta_preprocessor.dll
## library name: Terrain Analysis - Preprocessing
```

```
## tool name   : Fill Sinks (Planchon/Darboux, 2001)
## author      : Copyrights (c) 2003 by Volker Wichmann
##
## Load grid: 01_data/logan_MNT.sgrd...
##
##
## Parameters
##
##
## Grid system: 30; 985x 1660y; 432419.019091x 4612607.446916y
## DEM: logan_MNT
## Filled DEM: Filled DEM
## Minimum Slope [Degree]: 0.010000
##
## Save grid: 01_data/logan_FIL.sgrd...
```



On peut noter que le package **RSAGA** propose la fonction **rsaga.sink.route()**, qui est une implémentation directe du module **1** (**Sink Drainage Route Detection**) de la bibliothèque **ta_preprocessor**. Par ailleurs, il propose également la fonction **rsaga.sink.removal()** qui est une implémentation du module **2** (**Sink Removal**). La fonction **rsaga.fill.sinks()**, quant à elle, est une implémentation des modules **3** (**Fill Sinks (Planchon/Darboux, 2001)**), **4** (**Fill Sinks (Wang & Liu)**) et **5** (**Fill Sinks XXL (Wang & Liu)**).

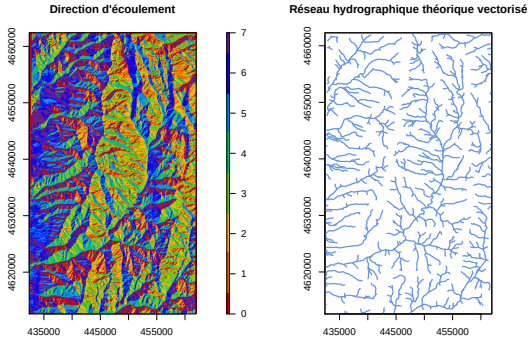
5.3.2.2 Détermination de la direction d'écoulement

La direction d'écoulement et le réseau hydrographique théorique sont renvoyés par le module **5** (**Channel Network and Drainage Basins**) de la bibliothèque **ta_channels**.

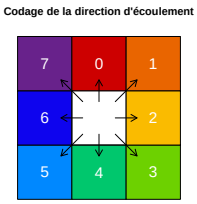
```
rsaga.geoprocessor(lib = "ta_channels", module = 5, env = ENV_sag,
                   param = list(DEM       = "01_data/logan_FIL.sgrd",
                                DIRECTION = "01_data/logan_DIR.sgrd",
                                SEGMENTS  = "01_data/logan_NET.shp"))
```

```
##
##
## library path: C:\OSGEO4~1\apps\saga\modules\ta_channels.dll
## library name: Terrain Analysis - Channels
## tool name   : Channel Network and Drainage Basins
## author      : O.Conrad (c) 2003
##
## Load grid: 01_data/logan_FIL.sgrd...
##
##
## Parameters
##
##
## Grid system: 30; 985x 1660y; 432419.019091x 4612607.446916y
## Elevation: logan_FIL
## Flow Direction: Flow Direction
## Flow Connectivity: <not set>
## Strahler Order: <not set>
## Drainage Basins: <not set>
## Channels: Channels
## Drainage Basins: Drainage Basins
## Junctions: <not set>
## Threshold: 5
##
## Flow Direction
## Stream Order
## Junctions
## Drainage Basins
## Vectorising Grid Classes
##
##
## Parameters
##
##
## Grid system: 30; 985x 1660y; 432419.019091x 4612607.446916y
## Grid: Drainage Basins
```

```
## Polygons: Polygons
## Class Selection: all classes
## Class Identifier: 1.000000
## Vectorised class as...: one single (multi-)polygon object
##
## class identification
## Create index: Drainage Basins
## edge detection
## edge collection
## Channels
## Save grid: 01_data/logan_DIR.sgrd...
## Save shapes: 01_data/logan_NET.shp...
```

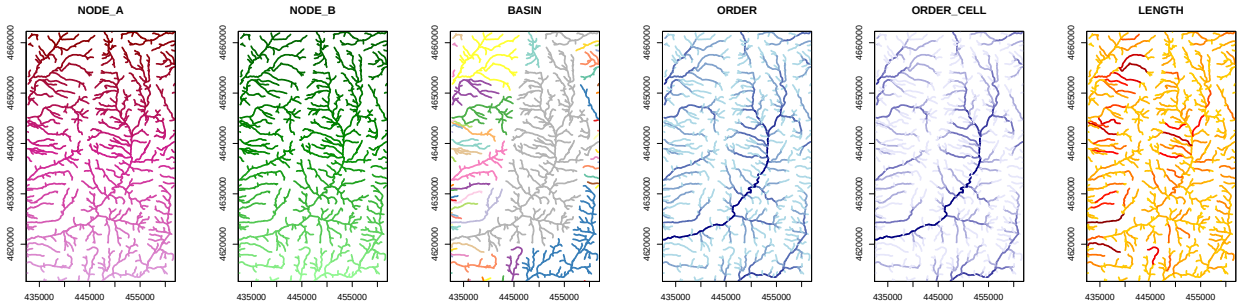


Le module renvoie une direction d'écoulement D8 codée comme suit.



Le fichier vectoriel renvoyé, et correspondant au réseau hydrographique théorique, présente plusieurs variables dans sa table attributaire, à savoir :

- NODE_A : XXXX ;
- NODE_B : XXXX ;
- BASIN : le code d'appartenance au bassin ;
- ORDER : XXXX ;
- ORDER_CELL : XXXX ;
- LENGTH : la longueur de chaque arc.



5.3.2.3 Calcul de l'accumulation d'écoulement

Pour calculer le raster d'accumulation de flux, il faut utiliser la bibliothèque **ta_hydrology**.

```
rsaga.get.modules("ta_hydrology", env = ENV_sag)
```

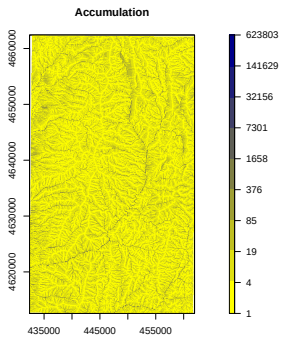
```
## $ta_hydrology
##    code                                  name interactive
## 1     0              Catchment Area (Parallel)       FALSE
## 2     1             Catchment Area (Recursive)       FALSE
## 3     2          Catchment Area (Flow Tracing)       FALSE
## 4     4                           Upslope Area       FALSE
## 5     6                        Flow Path Length       FALSE
```

```
## 6     7                          Slope Length       FALSE
## 7    10                          Cell Balance       FALSE
## 8    13                     Edge Contamination       FALSE
## 9    15                     SAGA Wetness Index       FALSE
## 10   16                            Lake Flood       FALSE
## 11   18      Catchment Area (Mass-Flux Method)       FALSE
## 12   19 Flow Width and Specific Catchment Area       FALSE
## 13   20        Topographic Wetness Index (TWI)       FALSE
## 14   21                     Stream Power Index       FALSE
## 15   22                              LS Factor       FALSE
## 16   23              Melton Ruggedness Number       FALSE
## 17   24                                TCI Low       FALSE
## 18   25                LS-Factor, Field Based       FALSE
## 19   26        Slope Limited Flow Accumulation       FALSE
```

On peut constater que plusieurs modules sont disponibles pour réaliser cette opération. Dans notre cas, nous utilisons le module `0` (**`Catchment Area (Parallel)`**). Attention, contrairement à ce qui est noté dans la documentation du module, il ne faut pas écrire le nom de l'argument **`Method`** entièrement en majuscules (seule la première lettre est une capitale). Si l'on ne respecte pas la casse de caractères, on obtiendra une erreur.

```
rsaga.geoprocessor(lib = "ta_hydrology", module = 0, env = ENV_sag,
                   param = list(ELEVATION   = "01_data/logan_FIL.sgrd",
                                ACCU_TOT    = "01_data/logan_ACC.sgrd",
                                Method      = 0))
```

```
##
##
## library path: C:\OSGEO4~1\apps\saga\modules\ta_hydrology.dll
## library name: Terrain Analysis - Hydrology
## tool name   : Catchment Area (Parallel)
## author      : O.Conrad (c) 2001-2010, T.Grabs portions (c) 2010
##
## Load grid: 01_data/logan_FIL.sgrd...
##
##
## Parameters
##
##
## Grid system: 30; 985x 1660y; 432419.019091x 4612607.446916y
## Elevation: logan_FIL
## Sink Routes: <not set>
## Weight: <not set>
## Material: <not set>
## Target: <not set>
## Catchment Area: Catchment Area
## Catchment Height: <not set>
## Catchment Slope: <not set>
## Total accumulated Material: Total accumulated Material
## Accumulated Material from _left_ side: <not set>
## Accumulated Material from _right_ side: <not set>
## Step: 1
## Catchment Aspect: <not set>
## Flow Path Length: <not set>
## Method: Deterministic 8
## Linear Flow: no
## Linear Flow Threshold: 500.000000
## Linear Flow Threshold Grid: <not set>
## Channel Direction: <not set>
## Convergence: 1.100000
##
## Create index: logan_FIL
## Save grid: 01_data/logan_ACC.sgrd...
```



5.3.2.4 Détermination du bassin versant recherché

Le module `20` (**`Snap Points to Grid`**) de la bibliothèque **`shapes_points`** permet le repositionnement automatique de l'exutoire sur le réseau hydrographique théorique.

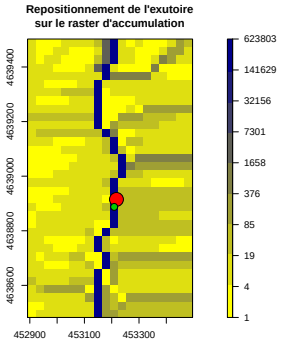
```
rsaga.geoprocessor(lib = "shapes_points", module = 20, env = ENV_sag,
                   param = list(INPUT    = "01_data/exu_POS.shp",
                                GRID     = "01_data/logan_ACC.sgrd",
                                OUTPUT   = "01_data/exu_POS_mov.shp",
                                DISTANCE = 50))
```

```
##
##
## library path: C:\OSGEO4~1\apps\saga\modules\shapes_points.dll
## library name: Shapes - Points
## tool name   : Snap Points to Grid
## author      : O.Conrad (c) 2012
##
## Load shapes: 01_data/exu_POS.shp...
## Load grid: 01_data/logan_ACC.sgrd...
##
##
## Parameters
##
##
## Grid system: 30; 985x 1660y; 432419.019091x 4612607.446916y
## Points: exu_POS
## Grid: logan_ACC
## Result: Result
## Moves: <not set>
## Search Distance (Map Units): 50.000000
## Search Shape: circle
## Extreme: maximum
##
## Save shapes: 01_data/exu_POS_mov.shp...
```

Après repositionnement sur le réseau, les nouvelles coordonnées sont les suivantes :

```
coordinates(EXU_pos_mov)
##      coords.x1 coords.x2
## [1,]    453209   4638887
```

Sur la carte suivante, on peut apprécier l'ampleur du déplacement de l'exutoire, depuis sa position d'origine (en rouge), vers sa nouvelle position sur le réseau hydrographique théorique (en vert).

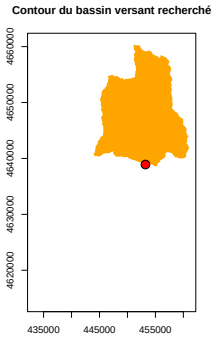


Le bassin versant en amont de l'exutoire est déterminé par le module **4** (**Upslope Area**) de la bibliothèque **ta_hydrology**.

```
rsaga.geoprocessor(lib = "ta_hydrology", module = 4, env = ENV_sag,
                   param = list(ELEVATION   = "01_data/logan_FIL.sgrd",
                                TARGET_PT_X = coordinates(EXU_pos_mov)[, 1],
                                TARGET_PT_Y = coordinates(EXU_pos_mov)[, 2],
                                AREA        = "01_data/logan_CBS.sgrd",
                                METHOD      = 0))
```

```
##
##
## library path: C:\OSGEO4~1\apps\saga\modules\ta_hydrology.dll
## library name: Terrain Analysis - Hydrology
## tool name   : Upslope Area
## author      : (c) 2001 by O.Conrad
##
## Load grid: 01_data/logan_FIL.sgrd...
##
##
## Parameters
##
##
## Grid system: 30; 985x 1660y; 432419.019091x 4612607.446916y
## Target Area: <not set>
## Target X coordinate: 453209.019091
## Target Y coordinate: 4638887.446916
## Elevation: logan_FIL
## Sink Routes: <not set>
## Upslope Area: Upslope Area
```

```
## Method: Deterministic 8
## Convergence: 1.100000
##
## Create index: logan_FIL
## Save grid: 01_data/logan_CBS.sgrd...
```



Le bassin versant ainsi obtenu présente une superficie d'environ 231.17 km^2.

5.3.3 Import des données SAGA GIS dans R

5.3.3.1 Données matricielles

Il n'est pas possible de lire, directement dans R, les fichiers de données matricielles au format SAGA GIS dans l'environnement de R, que ce soit avec le package **rgdal** ou avec avec le package **raster**. Pour importer des données matricielles au format SAGA GIS, il faut donc utiliser la fonction **rsaga.geoprocessor()** et appeler le module **2** (**GDAL: Export Raster to GeoTIFF**) de la bibliothèque **io_gdal**. Comme son nom l'indique, cette fonction permet de convertir un fichier au format ".sgrd" vers un format de données plus conventionnel (i.e. GeoTIFF). On peut alors importer ce dernier avec les fonctions habituelles telle que la fonction **raster()**.

```
rsaga.geoprocessor(lib = "io_gdal", module = 2, env = ENV_sag,
                   param = list(GRIDS = "01_data/logan_DIR.sgrd",
                                FILE  = "01_data/logan_DIR.tif"))
```

```
##
##
## library path: C:\OSGEO4~1\apps\saga\modules\io_gdal.dll
## library name: Import/Export - GDAL/OGR
## tool name   : GDAL: Export Raster to GeoTIFF
## author      : O.Conrad (c) 2007
##
## Load grid: 01_data/logan_DIR.sgrd...
##
##
## Parameters
##
##
## Grid system: 30; 985x 1660y; 432419.019091x 4612607.446916y
## Grid(s): 1 object (logan_DIR)
## File: 01_data/logan_DIR.tif
## Creation Options:
##
## Band 1
```

5.3.3.2 Données vectorielles

Les données vectorielles étant au format Shapefile, on les importe donc dans R avec les outils habituels, tels que ceux proposés par les packages **rgdal** et **maptools**.

5.4 Suite de logiciels ArcGIS

ArcGIS (ESRI, 2013) est une suite de logiciels d'information géographique propriétaires, développés par la société américaine ESRI (*Environmental Systems Research Institute, Inc.*) pour l'environnement Windows, depuis 1999. (Wikipedia, 2015a.)

C'est le package **RPyGeo** (Brenning, 2012) qui permet d'établir la communication entre R et ArcGIS. **RPyGeo** permet d'écrire des commandes Python qui appellent les modules ArcGIS. Rappelons ici que, tout comme le logiciel QGIS, ArcGIS propose une interface Python permettant de lancer des procédures *via* des scripts écrits dans ce langage de programmation.

```
library(RPyGeo)
```

5.4.1 Préparation des données

5.4.1.1 Définition de l'environnement de travail

Avant toute chose, il convient de définir l'environnement de travail permettant à R de se connecter à ArcGIS. Cette étape se fait grâce à la fonction **rpygeo.build.env()**. On lui fournit le chemin de l'espace de travail (**workspace**) et le chemin de l'interpréteur Python (**python.path**), où se situe la commande "python.exe". L'argument **overwriteoutput = 1** autorisera le réécriture de fichiers, si ces derniers sont déjà existants. Par ailleurs, il est possible de préciser la ou les extensions qui seront utilisées, mais la plupart du temps, il est inutile de le préciser.

```
ENV_arc <- rpygeo.build.env(python.path = "C:/Python27/ArcGIS10.2",
                            workspace = paste(getwd(), "01_data", sep = "/"),
                            overwriteoutput = 1,
                            extensions = "Spatial")
ENV_arc
## $modules
## [1] "arcgisscripting"
##
## $init
## [1] "gp = arcgisscripting.create()"
##
## $workspace
## [1] "E:/Data/boulot/misc/scientific_doc/R/formation/irstea/033-geomatique_LIV/01_manuscrit_KNITR/03_partie_analy
##
## $cellsize
## NULL
##
## $extent
## NULL
##
## $mask
## NULL
##
## $snapraster
## NULL
##
## $overwriteoutput
## [1] 1
##
## $extensions
## [1] "Spatial"
##
## $python.path
## [1] "C:/Python27/ArcGIS10.2"
##
## $python.command
## [1] "python.exe"
```

5.4.1.2 Fonction générique

Pour réaliser les traitements, le package **RPyGeo** propose la fonction générique **rpygeo.geoprocessor()**, qui permet d'appeler chacune des fonctionnalités d'ArcGIS. Pour cela, on fournit à cette fonction le nom du module ArcGIS que l'on souhaite utiliser à l'argument **fun**. Les différents paramètres du module sont, quant à eux, fournis à l'argument **args**, mais ces derniers ne sont pas explicites, il faut donc bien respecter l'ordre de la documentation. Par ailleurs, le package **RPyGeo** propose plusieurs fonctions correspondant à un implémentation de modules ArcGIS.

> Dans les exemples, pour plus de lisibilité lors de l'utilisation de la fonction générique **rpygeo.geoprocessor()**, les différents arguments des modules ArcGIS seront nommés avec la syntaxe de la documentation.

5.4.2 Traitement des données

Nous allons réaliser ici les traitements sur le modèle numérique de terrain, en vue de la détermination du contour d'un bassin versant.

5.4.2.1 Correction du fichier d'élévation

La première étape consiste à corriger le fichier d'élévation dans le but de combler les éventuelles cuvettes présentes. Pour cela, on utilise le module ArcGIS **Fill_sa**.

```
rpygeo.geoprocessor(fun  = "Fill_sa",
                    args = list(in_surface_raster  = "logan_MNT.tif",
                                out_surface_raster = "logan_FIL.tif"),
                    env  = ENV_arc)
## NULL
```

Comme dit précédemment, lors de l'utilisation de la fonction **rpygeo.geoprocessor()** les paramètres entrés dans l'argument **args** doivent respecter l'ordre mentionné dans la documentation.

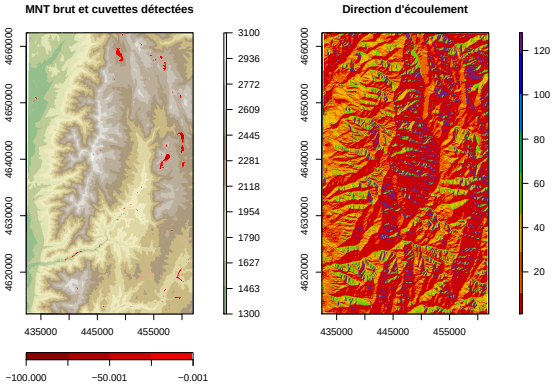
5.4.2.2 Détermination de la direction d'écoulement

Une fois que l'on a obtenu un modèle numérique de terrain topologiquement propre, on peut calculer la direction d'écoulement. Pour ce faire, on peut utiliser le module ArcGIS **FlowDirection_sa**, que l'on appelle avec la fonction générique **rpygeo.geoprocessor()**.

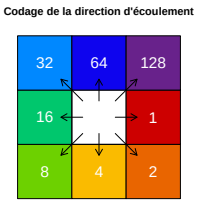
```
rpygeo.geoprocessor(fun  = "FlowDirection_sa",
                    args = list(in_surface_raster         = "logan_FIL.tif",
                                out_flow_direction_raster = "logan_DIR.tif",
                                force_flow                = "NORMAL"),
                    env  = ENV_arc)
## NULL
```

On peut également utiliser fonction **rpygeo.FlowDirection.sa()** du package **RPyGeo**, qui est une implémentation directe du module **FlowDirection_sa**.

```
rpygeo.FlowDirection.sa(in.surface.raster         = "logan_FIL.tif",
                        out.flow.direction.raster = "logan_DIR.tif",
                        force.flow = "NORMAL",
                        env = ENV_arc)
## NULL
```



Le module **FlowDirection_sa** calcule une direction d'écoulement D8 codée comme ci-dessous.



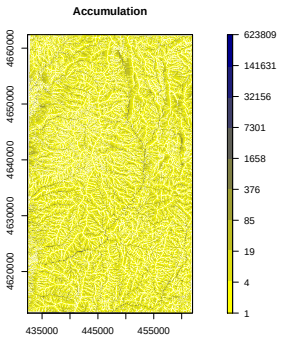
5.4.2.3 Calcul de l'accumulation d'écoulement

On peut, à présent, calculer l'accumulation de flux grâce au module **FlowAccumulation_sa**.

```
rpygeo.geoprocessor(fun  = "FlowAccumulation_sa",
                    args = list(in_flow_direction_raster = "logan_DIR.tif",
                                out_accumulation_raster  = "logan_ACC.tif"),
                    env  = ENV_arc)
## NULL
```

On peut également utiliser son implémentation directe dans le package **RPyGeo**, sous la forme de la fonction **FlowAccumulation.sa()**.

```
rpygeo.FlowAccumulation.sa(in.flow.direction.raster = "logan_DIR.tif",
                           out.accumulation.raster  = "logan_ACC.tif",
                           env = ENV_arc)
## NULL
```



5.4.2.4 Délimitation du bassin versant

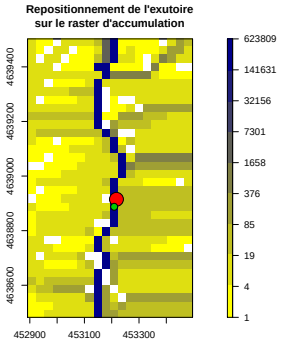
Il est possible de repositionner l'exutoire sur le réseau en utilisant le module **SnapPourPoint_sa**. Le module prend, en entrée, la position initiale de l'exutoire (aux formats vectoriel ou matriciel), ainsi que le raster d'accumulation d'écoulement. Il renvoie le résultat sous forme d'un fichier matriciel. Que ce soit en entrée ou en sortie, les pixels prennent une valeur "1" pour coder la position d'un l'exutoire et des valeurs vides ailleurs. Le module admet un argument facultatif, **snap_distance**, qui correspond à la distance maximale (en unité de carte) de recherche d'une cellule d'accumulation de flux plus élevée.

```
rpygeo.geoprocessor(fun  = "SnapPourPoint_sa",
                    args = list(in_pour_point_data     = "exu_POS.shp",
                                in_accumulation_raster = "logan_ACC.tif",
                                out_raster             = "exu_POS_mov",
                                snap_distance          = 50),
                    env  = ENV_arc)
## NULL
```

Après repositionnement sur le réseau, les nouvelles coordonnées sont les suivantes :

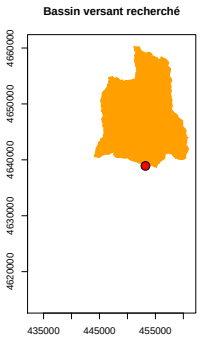
```
coordinates(EXU_pos_mov)
##           x       y
## [1,] 453209 4638887
```

Sur la carte suivante, on peut apprécier l'ampleur du déplacement de l'exutoire, depuis sa position d'origine (en rouge), vers sa nouvelle position sur le réseau hydrographique théorique (en vert).



Le bassin versant en amont de l'exutoire est déterminé par le module **Watershed_sa**. Le second argument du module, **in_pour_point_data**, correspond à la position de l'exutoire (il peut s'agir d'un fichier vectoriel ou d'un raster).

```
rpygeo.geoprocessor(fun  = "Watershed_sa",
                    args = list(in_flow_direction_raster = "logan_DIR.tif",
                                in_pour_point_data       = "exu_POS_mov",
                                out_raster               = "logan_CBS.tif"),
                    env  = ENV_arc)
## NULL
```



Le bassin versant ainsi obtenu présente une superficie d'environ 231.17 km^2.

5.5 Suite d'outils TauDEM

TauDEM (*Terrain Analysis Using Digital Elevation Models*) est une suite d'outils de traitement de modèle numérique de terrain pour extraire et analyser les informations hydrologiques d'après des données topographiques. TauDEM a été créé par David Tarboton (1997, 2013), de l'université d'État de l'Utah, à partir des années 1990, d'abord sous le nom de TARDEM, dont la première version date de 1997, puis sous son nom actuel, depuis 2001. Il est disponible sous la forme d'exécutables pour Windows [6]. Par ailleurs, des plugins pour ArcGIS et QGIS ont été développés, et le code source des modules est mis librement à disposition.

5.5.1 Installation

Pour installer et lancer TauDEM sous Windows, il est nécessaire de disposer des droits administrateurs et d'installer au préalable :

- Visual Studio 2010 C++ runtime libraries (x86 ou x64 selon la machine) ;
- Microsoft HPC Pack 2012 MS-MPI.

On peut ainsi disposer du programme "mpiexec.exe" qui permet à TauDEM de travailler en parallèle sur plusieurs processeurs.

5.5.2 Syntaxe des commandes

Malheureusement, il n'existe pas, pour le moment, de package permettant de communiquer entre R et TauDEM.

Pour lancer un exécutable TauDEM depuis R, il est nécessaire d'exécuter la commande avec la fonction `system()` [7].

Le premier argument que l'on définit est toujours le nombre de processeurs sur lequel on souhaite travailler (pour 8 processeurs : `mpiexec -n 8`). Le deuxième argument est le nom du module TauDEM que l'on souhaite exécuter. Il faut ensuite entrer les arguments du module choisi (l'ordre n'importe pas).

> Afin de raccourcir la taille des lignes de commande, pour qu'elles ne débordent pas en dehors des pages du présent document, le répertoire contenant les données ("01_data") a été défini comme nouveau répertoire de travail de R par défaut. Par conséquent, lorsque l'on effectuera une connexion à un fichier, seul sera écrit le nom de fichier, et non le chemin du répertoire le contenant.

```
setwd("01_data")
```

6. La description et l'aide des modules est disponible à l'adresse suivante : http://hydrology.usu.edu/taudem/taudem5/documentation.html.

7. Pour lancer TauDEM en lignes de commande (comme c'est le cas dans R), on peut se référer au manuel suivant : http://hydrology.usu.edu/taudem/taudem5/TauDEM51CommandLineGuide.pdf.

5.5.3 Traitement des données

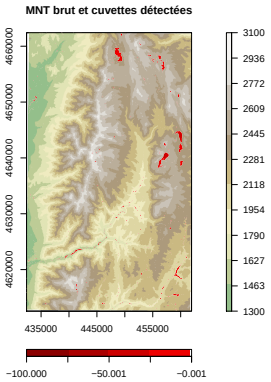
Nous allons réaliser ici les traitements sur le modèle numérique de terrain, en vue de la détermination du contour d'un bassin versant.

5.5.3.1 Correction du fichier d'élévation

La première étape du traitement consiste à combler les cuvettes du MNT. Pour cela, on utilise le module **`pitremove`**.

- entrées :
  - **`z`** : le raster du MNT brut.
- sorties :
  - **`fel`** : le raster du MNT comblé.

```
system("mpiexec -n 8 pitremove -z logan_MNT.tif -fel logan_FIL.tif")
```

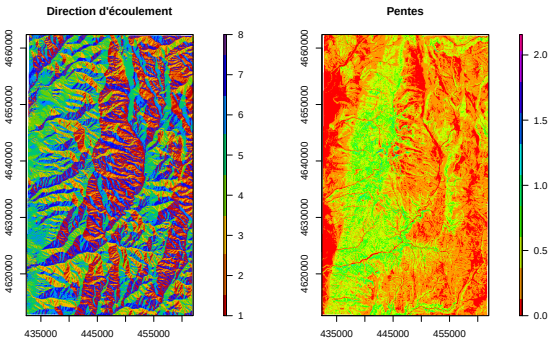


5.5.3.2 Détermination de la direction d'écoulement et des pentes

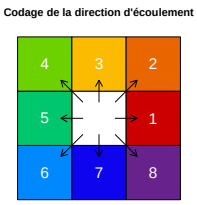
Le module **`D8Flowdir`** permet de calculer la direction d'écoulement D8 et les pentes à partir d'un MNT (ici le MNT comblé).

- entrées :
  - **`fel`** : le raster du MNT comblé.
- sorties :
  - **`p`** : le raster de la direction d'écoulement ;
  - **`sd8`** : le raster de pentes.

```
system("mpiexec -n 8 D8Flowdir -fel logan_FIL.tif -p logan_DIR.tif -sd8 logan_SLP.tif")
```



Le module **`D8Flowdir`** calcule une direction d'écoulement D8 codée comme suit.

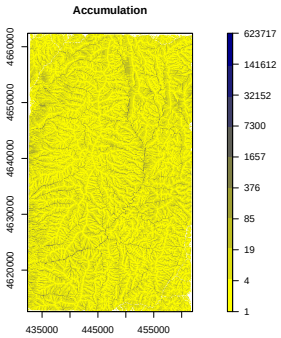


5.5.3.3 Calcul de l'accumulation d'écoulement

Le module **AreaD8** permet de calculer les aires de contribution à partir de la direction d'écoulement D8.

- entrées :
  - **p** : le raster de la direction d'écoulement D8.
- sorties :
  - **ad8** : le raster des aires de contribution.

```
system("mpiexec -n 8 AreaD8 -p logan_DIR.tif -ad8 logan_ACC.tif")
```

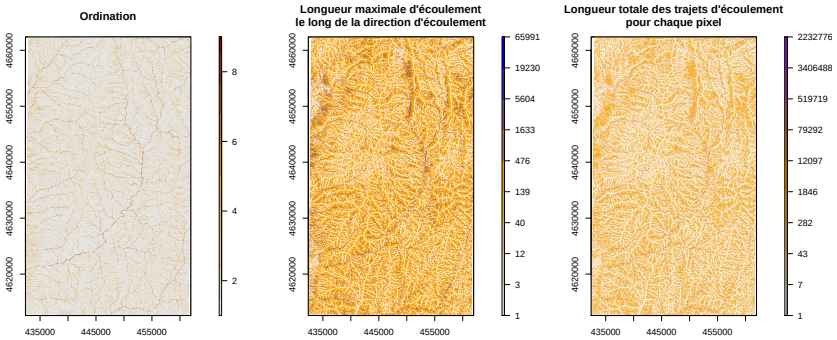


5.5.3.4 Détermination du réseau hydrographique théorique

Le module **Gridnet** permet de déterminer le raster du réseau à partir de la direction d'écoulement D8. XXXX

- entrées :
  - **p** : le raster de la direction d'écoulement D8.
- sorties :
  - **gord** : le raster du réseau hydrographique (grid network order) ;
  - **plen** : the longest flow path along D8 flow directions to each grid cell ;
  - **tlen** : the total length of all flow paths that end at each grid cell.

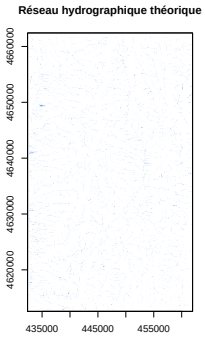
```
system("mpiexec -n 8 Gridnet -p logan_DIR.tif -gord logan_ORD.tif -plen logan_PLN.tif -tlen logan_TLN.tif")
```



Le module **Threshold** produit un raster du réseau hydrographique théorique à partir du raster des aires de contribution et d'un seuil.

- entrées :
  - **ssa** : le raster des aires de contribution ;
  - **thresh** : la valeur du seuil d'aire de contribution pour la détermination du réseau hydrographique.
- sorties :
  - **src** : le raster du réseau hydrographique théorique (un pixel vaut 1 pour le réseau et 0 ailleurs).

```
system("mpiexec -n 8 Threshold -ssa logan_ACC.tif -thresh 100 -src logan_NET.tif")
```



5.5.3.5 Délimitation du bassin versant

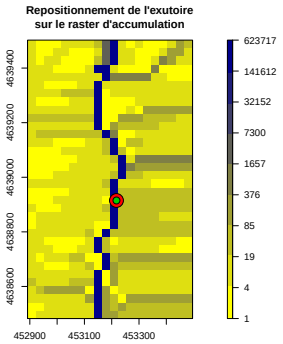
Le module **moveoutletstostreams** permet le repositionnement automatique de l'exutoire sur le réseau hydrographique théorique.

- entrées :
  - p : le raster de la direction d'écoulement ;
  - o : les coordonnées approximatives de l'exutoire au format vectoriel ;
  - src : le raster du réseau hydrographique théorique.
- sorties :
  - om : les coordonnées de l'exutoire replacé sur le réseau hydrographique théorique au format vectoriel.

```
system("mpiexec -n 8 moveoutletstostreams -p logan_DIR.tif -o exu_POS.shp -src logan_NET.tif -om exu_POS_mov.shp")
```

```
coordinates(EXU_pos_mov)
##       coords.x1 coords.x2
## [1,]  453217.4   4638915
```

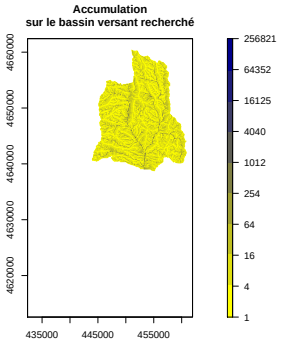
Sur la carte suivante, on peut apprécier l'ampleur du déplacement de l'exutoire, depuis sa position d'origine (en rouge), vers sa nouvelle position sur le réseau hydrographique théorique (en vert).



Comme vu plus haut, le module **AreaD8** permet de calculer les aires de contribution à partir de la direction d'écoulement D8. Ici, en plus du raster de direction d'écoulement, on définit également en entrée les coordonnées de l'exutoire du bassin versant que l'on souhaite définir.

- entrées :
  - **p** : le raster de la direction d'écoulement D8 ;
  - **o** : les coordonnées de l'exutoire positionné sur le réseau hydrographique au format vectoriel.
- sorties :
  - **ad8** : le raster de la surface du bassin en amont de l'exutoire.

```
system("mpiexec -n 8 Aread8 -p logan_DIR.tif -o exu_POS_mov.shp -ad8 logan_ACC_BS.tif")
```

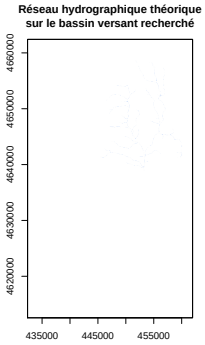


Le bassin versant ainsi obtenu présente une superficie d'environ 231.14 km².

Comme vu précédemment, le module **Threshold** produit un raster du réseau hydrographique théorique à partir du raster des aires de contribution et d'un seuil.

- entrées :
  - **ssa** : le raster des aires de contribution ;
  - **thresh** : la valeur du seuil d'aire de distribution pour la détermination du réseau hydrographique théorique.
- sorties :
  - **src** : le raster du réseau hydrographique théorique (un pixel vaut 1 pour le réseau et 0 ailleurs).

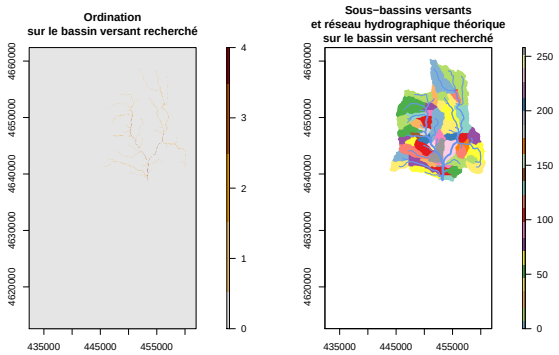
```
system("mpiexec -n 8 threshold -ssa logan_ACC_BS.tif -thresh 2000 -src logan_NET_BS.tif")
```



Le module **Streamnet** XXXX

- entrées :
  - **fel** : le raster du MNT comblé ;
  - **p** : le raster de la direction d'écoulement D8 ;
  - **ad8** : le raster de la surface du bassin en amont de l'exutoire ;
  - **src** : le raster du réseau hydrographique théorique ;
  - **o** : les coordonnées de l'exutoire positionné sur le réseau hydrographique théorique au format vectoriel.
- sorties :
  - **ord** : le raster du réseau suivant l'ordination de Strahler ;
  - **tree** : les relations entre les nœuds du réseau au format texte ;
  - **coord** : les coordonnées des nœuds du réseau au format texte ;
  - **net** : le réseau au format vectoriel ;
  - **w** : le raster des sous-bassins versants correspondants aux branches du réseau vectoriel.

```
system("mpiexec -n 8 Streamnet -fel logan_FIL.tif -p logan_DIR.tif -ad8 logan_ACC.tif -src logan_NET_BS.tif -o exu_
```



5.6 Logiciel QGIS

QGIS (QGIS Development Team, 2016), initialement appelé Quantum GIS, est un logiciel de système d'information géographique libre multiplate-forme distribué sous licence GPL. Son développement a débuté en mai 2002 et a été publiée en tant que projet sur SourceForge en juin de la même année. Le code a été développé en C++, mais QGIS peux utiliser des plugins développées dans ce même langage ou en Python. Il gère aussi bien les formats d'images matricielles, les données vectorielles, ainsi que les bases de données. QGIS fait partie des projets de la fondation *Open Source Geospatial*.

QGIS propose ses propres géotraitements, mais il permet également l'accès à certaines fonctionnalités de GDAL/OGR, GRASS GIS, SAGA GIS, Orfeo, et éventuellement TauDEM (si le *plugin* est préalablement installé).

Pour établir la connexion entre QGIS et R, il faut utiliser le package **RQGIS** (Muenchow & Schratz, 2016).

```
library(RQGIS)
```

5.6.1 Préparation des données

5.6.1.1 Définition de l'environnement de travail

On initialise la connexion à l'aide de la fonction **set_env()**. Cette dernière effectue une recherche automatique du chemin de l'application si l'argument **root** n'est pas spécifié (par défaut, **root = NULL**).

```
ENV_qgs <- set_env()
ENV_qgs
```

```
## $root
## [1] "C:\\OSGeo4W64"
##
## $qgis_prefix_path
## [1] "C:\\OSGeo4W64\\apps\\qgis"
##
## $python_plugins
## [1] "C:\\OSGeo4W64\\apps\\qgis\\python\\plugins"
```

5.6.1.2 Syntaxe des commandes

5.6.1.2.a Recherche de de module

On peut effectuer une recherche par mots clés sur les noms de bibliothèques ou de modules, grâce à la fonction **find_algorithms()**. L'argument **name_only**, permet de récupérer uniquement le nom du module ou ce dernier plus un court descriptif.

```
find_algorithms(search_term = "slope", name_only = FALSE, qgis_env = ENV_qgs)
```

```
##  [1] "Downslope distance gradient-------------------------->saga:downslopedistancegradient"
##  [2] "Dtm filter (slope-based)----------------------------->saga:dtmfilterslopebased"
##  [3] "Relative heights and slope positions----------------->saga:relativeheightsandslopepositions"
##  [4] "Slope length----------------------------------------->saga:slopelength"
##  [5] "Slope, aspect, curvature----------------------------->saga:slopeaspectcurvature"
##  [6] "Upslope Area----------------------------------------->saga:upslopearea"
##  [7] "Vegetation index (slope based)----------------------->saga:vegetationindexslopebased"
##  [8] "Slope------------------------------------------------>gdalogr:slope"
##  [9] "r.flow - Construction of slope curves (flowlines), flowpath lengths, and flowline densities (upslope areas
```

```
## [10] "r.slope - Generates raster maps of slope from a elevation raster map.--->grass:r.slope"
## [11] "r.slope.aspect - Generates raster layers of slope, aspect, curvatures and partial derivatives from a eleva
## [12] ""
```

5.6.1.2.b Consultation de la documentation

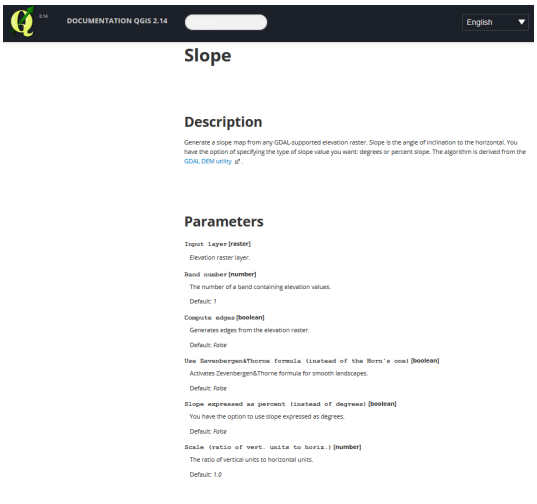
La fonction **get_usage** permet de récupérer la liste des paramètres d'un module, ainsi que le type d'objet qu'ils contiennent.

```
get_usage(alg = "gdalogr:slope", qgis_env = ENV_qgs)
```

```
##  [1] "ALGORITHM: Slope"                    "\tINPUT <ParameterRaster>"
##  [3] "\tBAND <ParameterNumber>"             "\tCOMPUTE_EDGES <ParameterBoolean>"
##  [5] "\tZEVENBERGEN <ParameterBoolean>"     "\tAS_PERCENT <ParameterBoolean>"
##  [7] "\tSCALE <ParameterNumber>"            "\tOUTPUT <OutputRaster>"
##  [9] ""                                    ""
## [11] ""
```

Pour obtenir davantage d'explications, on peut consulter la documentation des modules grâce à la fonction **open_help()**.

```
open_help(alg = "gdalogr:slope", qgis_env = ENV_qgs)
open_help(alg = "grass:r.slope.aspect", qgis_env = ENV_qgs)
```



5.6.1.2.c Définition des paramètres

La fonction **get_args_man()** récupère automatiquement, sous forme de liste, les paramètres du module (**alg**) et ses paramètres par défaut (**options**).

```
params <- get_args_man(alg = "gdalogr:slope",
                       options = TRUE,
                       qgis_env = ENV_qgs)
params
```

```
## $INPUT
## [1] "None"
##
## $BAND
## [1] "None"
##
## $COMPUTE_EDGES
## [1] "None"
##
## $ZEVENBERGEN
## [1] "None"
##
## $AS_PERCENT
## [1] "None"
##
## $SCALE
## [1] "None"
##
## $OUTPUT
## [1] "None"
```

Pour définir les paramètres non optionnels, il suffit de réaliser des assignations sur chacun des éléments de la liste concernés.

```
params$INPUT         <- paste(getwd(), "01_data/logan_MNT.tif", sep = "/")
params$BAND          <- 1
params$COMPUTE_EDGES <- 0
params$ZEVENBERGEN   <- 0
```

```
params$AS_PERCENT    <- 1
params$SCALE         <- 1
params$OUTPUT        <- paste(getwd(), "01_data/logan_SLP.tif", sep = "/")
```

Attention, les chemins doivent être complets depuis la racine et doivent comporter impérativement des slashs et non des anti-slash doublés.

5.6.1.2.d Fonction générique

Le package **RQGIS** propose une fonction générique permettant d'appeler n'importe quel module de n'importe quel logiciel, il s'agit **run_qgis()**. Cette fonction prend en premier argument le nom du module que l'on souhaite exécuter (**alg**) et les paramètres de celui-ci (**params**). Par ailleurs, l'argument **load_output** permet de charger automatiquement dans l'environnement de R, le résultat généré par le module.

```
RAS_slp <- run_qgis(alg = "gdalogr:slope",
                    params = params,
                    load_output = params$OUTPUT,
                    qgis_env = ENV_qgs)
RAS_slp
## class       : RasterLayer
## dimensions  : 1660, 985, 1635100  (nrow, ncol, ncell)
## resolution  : 30, 30  (x, y)
## extent      : 432404, 461954, 4612592, 4662392  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=12 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs
## data source : E:\Data\boulot\misc\scientific_doc\R\formation\irstea\033-geomatique_LIV\01_manuscrit_KNITR\03_par
## names       : logan_SLP
```

Mis à part pour Orfeo, les traitements de données au moyen des bibliothèques et des logiciels attaqués par QGIS ayant été abordés dans les parties précédentes, nous ne développerons pas davantage ces aspects.

5.7 Solution à adopter

Le question qui est maintenant soulevée est de savoir quelle est la solution à adopter pour le traitement des données géographiques utilisant un outil externe à R. Nous allons donc passer en revue les avantages et les inconvénients de chacune des solutions que nous venons de présenter.

GDAL/OGR – rgdalUtils. Cette solution est un peu à part des autres. En effet, la bibliothèque GDAL/OGR ne propose par vraiment les mêmes fonctionnalités que les autres outils que nous avons présentés. Ces dernières concernent la conversion de format de fichier, la conversion de système de coordonnées, etc., mais moins les traitements géomatiques à proprement parlé (e.g. détermination d'un réseau hydrographique théorique, délimitation d'un bassin versant, etc.), même si cette bibliothèque en présente toutefois quelques-unes (e.g. analyse de terrain). Pour ce qui est des avantages, on peut noter que GDAL/OGR est un logiciel libre et que la connexion entre ce dernier et R est très aisée. Un des points noirs du package **rgdalUtils** est que, pour le moment, seule une petite fraction des fonctionnalités de GDAL/OGR y sont implantées sous la forme de fonctions R. Pour les autres fonctionnalités, non implémentées, **rgdalUtils** propose une fonction générique d'assistance à l'écriture de commandes GDAL/OGR, ce qui est très utile. Cependant, sa mise en œuvre est assez lourde, car elle est extrêmement verbeuse. Un autre point négatif est la faiblesse de la taille de l'effectif des membres qui participent au développement et à la maintenance de GDAL/OGR (ce qui explique la persistance de certains bugs pourtant recensés depuis longtemps par les utilisateurs).

GRASS GIS – spgrass6/rgrass7. Cette solution présente de nombreux avantages. Tout d'abord, GRASS GIS est un logiciel libre. Par ailleurs, ce logiciel existe depuis longtemps, ce qui lui confère une certaine robustesse. L'utilisation des fonctionnalités de GRASS GIS dans R est aisée. En effet, les packages **spgrass6** et **rgrass7** proposent une fonction générique dont l'utilisation est très facile. Par ailleurs, la lecture dans R des données au format GRASS GIS est très aisé La documentation des différents modules de GRASS GIS est très explicite. Le principal point négatif reste la lourdeur de la création de la base de donnée, son poids, ainsi que la spécificité du format de données du logiciel. Par ailleurs, la connexion entre R et GRASS GIS est parfois un peu capricieuse. En effet, on rencontre parfois des problèmes de DLL manquantes ou de connexion avec une version de GRASS GIS qui fonctionne dans un emplacement, mais pas dans un autre (emplacement propre, dans OSGeo4W ou dans QGIS), sans que l'on comprenne pourquoi. Enfin, même s'il permet de réaliser de plus en plus de traitements sur les données vectorielles, ceux-ci restent encore minoritaires ; GRASS GIS est historiquement un logiciel tourné vers la manipulation de données matricielles. Par ailleurs, ce logiciel étant ancien, la parallélisation des calculs est rendue difficile, car le développement de GRASS GIS n'a pas été pensé en ce sens.

SAGA GIS – RSAGA. Comme pour les deux solutions précédentes, un avantage de celle-ci est qu'elle met en œuvre un logiciel libre. Par ailleurs, il est très facile d'établir la connexion entre le logiciel SAGA GIS et R. On ne rencontre, en effet, pas de souci particulier pour la mettre en place. L'existence d'une fonction générique proposée par le package `RSAGA` rend l'utilisation des modules de SAGA GIS très aisée sous R. Un des principaux points négatifs est la documentation de SAGA GIS qui n'est pas toujours très explicite. En effet, on ne comprend pas toujours très bien quelles sont les données à fournir aux modules et quels sont les résultats renvoyés par ce dernier. Il reste aussi la manipulation du format matriciel spécifique de SAGA GIS et la lourdeur de conversion pour une lecture depuis R, qui peuvent rendre l'utilisation de `RSAGA` un peu pénible. Tout comme GRASS GIS, SAGA GIS était initialement tourné vers le traitement de données matricielles, ce qui explique que la majorité des fonctionnalités soient orientées vers ce type de données.

ArcGIS – RPyGeo. Le logiciel ArcGIS étant sans doute un des plus utilisés en géomatique, cette solution présente un avantage certain. En outre ce logiciel traite tout aussi bien les données vectorielles que matricielles. Par ailleurs, l'aide du logiciel est assez bien réalisée et illustrée avec des exemples clairs. Même si son développement ne repose pas sur une communauté d'utilisateurs, la société ESRI, qui l'édite, dispose d'une grande force de frappe. Les mises à jour sont régulières et les problématiques de maintenance ne posent pas de difficultés particulières. Le principal inconvénient est qu'ArcGIS est un logiciel propriétaire disponible uniquement sous l'environnement Windows ou trois versions Linux utilisant des processeurs conformes à l'architecture x86 (Red Hat Enterprise Linux Server 5 et 6 ; SuSE Linux Enterprise Server 11). Par ailleurs, son coût est extrêmement élevé, ce qui est d'autant plus contraignant si l'on souhaite pouvoir disposer régulièrement des dernières versions mises à jour pour lesquelles de nouvelles fonctionnalités sont proposées et les bugs résolus. Concernant l'utilisation de la fonction générique du package `RPyGeo`, les arguments des fonctions ne sont pas nommés et les entrées ne sont pas distinguées des sorties, ce qui rend le code peu explicite.

TauDEM. Ce logiciel est gratuit et les sources sont mises à disposition des utilisateurs. Par ailleurs, cette solution présente un grand avantage en termes de parallélisation de calcul. Il est en effet très aisé de paralléliser les fonctionnalités et de choisir le nombre de processus que l'on souhaite lancer. D'un autre côté, un des points négatifs est l'orientation tournée uniquement vers les aspects hydrologiques. Par ailleurs, TauDEM ne propose que des fonctionnalité de traitement de données matricielles. La compilation a été réalisée pour Windows, et travailler avec ses outils sous Macintosh ou Linux demande un peu d'investissement de la part des utilisateurs. Sous R, il faut écrire les commandes système "à la main", car il n'existe pas de package permettant de communiquer avec TauDEM. Enfin, TauDEM n'est pas développé par une communauté ou une équipe, mais repose principalement sur l'investissement d'une seule et unique personne, ce qui peut poser question quant à l'évolution et à la pérennité de ce logiciel. Concernant la syntaxe des lignes de commandes, rien ne permet de distinguer les entrées des sorties, et les noms des arguments sont peu explicites, ce qui rend difficile la compréhension du code.

QGIS – RQGIS. Ce logiciel, libre, très populaire tend à supplanter ArcGIS. Le développement de sa communauté garantit un bon niveau de maintenance du logiciel et assure de trouver facilement de l'aide en ligne. Il propose ses propres fonctionnalités, mais également une partie de celles de GDAL/OGR, toutes celles GRASS GIS, Orfeo, SAGA GIS et TauDEM. La documentation des fonctionnalités, *via* le package `RQGIS`, est peu explicite, d'où un besoin de se référer à la documentation originelle de chacun des logiciels. L'initialisation de la connexion entre R et QGIS est très aisée, et la manipulation des fonctionnalités de chacun des packages ne pose pas de difficulté.

Quatrième partie

Représentation cartographique

Chapitre 6

Cartographie des objets spatiaux

De nombreux packages proposent des fonctionnalités permettant de réaliser des représentations cartographiques. Dans cette partie nous allons présenter certaines fonctions de différents packages (le but n'est nullement d'être exhaustif) que nous organiserons en trois groupes. Nous exposerons tout d'abord des fonctions qui utilisent une syntaxe conventionnelle, correspondant au *Painter's Model*, tel qu'implémenté dans R grâce au package **graphics** (R Core Team, 2016). Dans un deuxième temps, nous exposerons les fonctions qui utilisent une syntaxe de type *Trellis*, telle que celle proposée par le package **lattice** (Sarkar, 2008), et reposant sur le travail de Cleveland (1993). Enfin, il sera question des fonctions reposant sur une syntaxe appliquant la *Grammar of Graphics* de Wilkinson (2005), et popularisée sous R par le package **ggplot2** (Wickham, 2009).

Nous ne présenterons ici que les fonctions cartographiques travaillant avec des objets de classes **sp** et **raster**, exception fait du cas du package **ggplot2** (Wickham, 2009) (§ 6.3.1).

6.1 Syntaxe de type *Painter's Model*

La syntaxe de type *Painter's Model* repose sur les travaux menés aux Bell Labs sur le modèle GR-Z (Becker & Chambers, 1977). D'abord implémentée sous S (Becker & Chambers, 1984a,b, 1985) [1], elle a, par la suite, été transcrite sous R dans le package **graphics** (R Core Team, 2016). Cette manière de faire a prévalu dans la plupart des fonctionnalités graphiques de S, et elle est toujours très populaire en raison de sa simplicité, ainsi que de la disponibilité de sorties graphiques déjà implémentées. C'est la syntaxe des graphiques conventionnels de R. Les fonctionnalités cartographiques utilisant cette syntaxe sont donc compatibles avec les fonctions du package **graphics** (i.e. **points()**, **lines()**, **text()**, **legend()**, etc.).

Assigner une carte dans un objet. Quel que soit le package considéré, si ce dernier utilise la syntaxe de type *Painter's Model*, il n'est alors pas possible de sauver une carte dans un objet R, mais on peut toutefois utiliser la fonction **savePlot()** qui sauve la sortie existante dans la fenêtre graphique courante (voir aussi la fonction **dev.copy()**).

Assembler des cartes. Il n'y a pas de difficulté particulière pour assemble des cartes dans une même fenêtre graphique. On peut utiliser sans problème les fonctions habituelles : **par(mfrow)** ou **par(mfcol)**, **screen()** et ses fonctions associées, ou encore **layout()**.

6.1.1 Package sp

Comme nous l'avons vu précédemment (§1.1 et 1.2), le package **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a) définit les principales classes et méthodes de données spatiales de type vectoriel ou, dans une moindre mesure, de type matriciel. Par ailleurs, ce package fournit les outils nécessaires pour tracer des cartes à partir de ces objets.

```
library(sp)
```

6.1.1.1 Affichage

Que ce soit pour des objets vectoriels ou matriciels de classe **sp**, l'affichage se fait en appelant simplement la fonction **plot()**. Dans le cas des données matricielles uniquement, on peut également utiliser la fonction **image()**, mais il faut bien veiller à utiliser l'argument **asp = 1** pour fixer un ratio égal à 1 entre les unités des

1. Site web du langage S : http://ect.bell-labs.com/sl/S/.

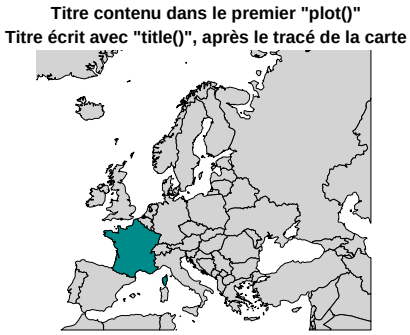
axes des abscisses et des ordonnées, sans quoi la carte est déformée. Pour connaître les arguments de la fonction **plot()** appliquée aux objets **sp**, il convient de consulter l'aide de la classe **sp** :

```
?'Spatial-class'
```

6.1.1.2 Éléments contextuels

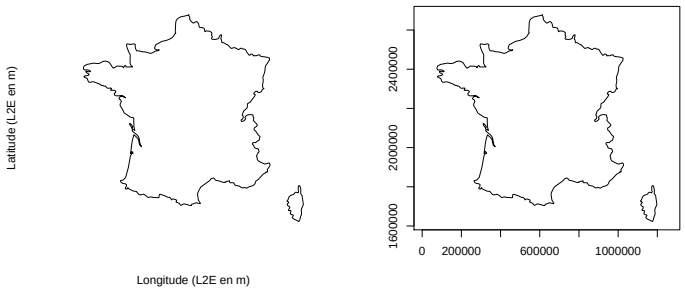
Titre. Le titre d'une carte se définit dans l'argument **main** du premier appel à la fonction **plot()**. Toute chaîne de caractères définie dans le **main** d'un appel secondaire à la fonction **plot()** n'aura aucun effet. *A posteriori*, une fois que l'ensemble des couches est dessiné, on peut tout de même ajouter un titre à la carte en utilisant la fonction **title()** du package **graphics** (R Core Team, 2016) (§ 6.1.3.2).

```
plot(EU_ctr1_LAEA, col = "lightgrey", main = 'Titre contenu dans le premier "plot()"')
plot(FR_ctr_LAEA, col = "cyan4"     , main = "Titre non utilisé", add = TRUE)
title(main = 'Titre écrit avec "title()", après le tracé de la carte', line = 0.4)
```



Axes. Pour afficher les étiquettes des axes, il suffit de définir les arguments **xlab** et **ylab** de la fonction **plot()**. On peut afficher les valeurs des axes en définissant l'argument **axes = TRUE**. Par défaut, **axes = FALSE** pour les fonctions **plot()** et **image()** appliquées à des objets de classe **sp**. Si l'on souhaite personnaliser l'affichage des axes, on peut utiliser la fonction **axis()** de package **graphics** (R Core Team, 2016) (§ 6.1.3.2).

```
plot(FR_ctr_L2E, xlab = "Longitude (L2E en m)", ylab = "Latitude (L2E en m)")
plot(FR_ctr_L2E, axes = TRUE)
```



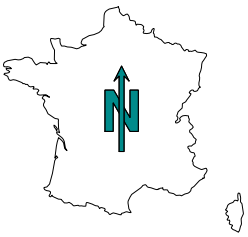
Orientation. Pour tracer la flèche du Nord, il faut fournir la fonction **layout.north.arrow()** en argument principal de la fonction **SpatialPolygonsRescale()**. Appliqués à la fonction **layout.north.arrow()**, les principaux arguments de la fonction **SpatialPolygonsRescale()** sont les suivants :

- **offset** : les coordonnées de la position de la flèche ;
- **scale** : la taille de la flèche [unité de la couche] ;
- **col** : la couleur de bordure ;
- **fill** : la couleur de fond ;
- **plot.grid** : gère la compatibilité avec les différentes syntaxes (doit être défini à la valeur **FALSE** pour que la fonction soit compatible avec la syntaxe conventionnelle de type *Painter's Model*).

L'argument **type**, proposé par **layout.north.arrow()**, admet 2 valeurs possibles :

- **1** : dessine la flèche avec la lettre "N" ;
- **2** : dessine la flèche sans la lettre "N".

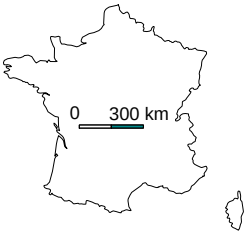
```
plot(FR_ctr_L2E)
SpatialPolygonsRescale(layout.north.arrow(type = 1), offset = c(550000, 2000000),
                       scale = 4e5, col = "black", fill = "cyan4",
                       plot.grid = FALSE)
```



Échelle. Pour tracer l'échelle, il faut fournir la fonction **layout.scale.bar()** en argument principal de la fonction **SpatialPolygonsRescale()**. Le texte est ajouté avec la fonction **text()** du package **graphics** (R Core Team, 2016) (§ 6.1.3.2). Appliqués à la fonction **layout.scale.bar()**, les principaux arguments de la fonction **SpatialPolygonsRescale()** sont les suivants :

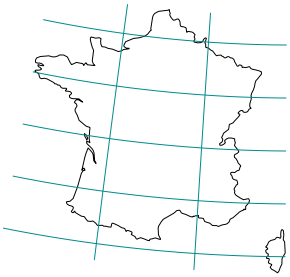
- **offset** : les coordonnées de la position de l'échelle ;
- **scale** : la taille de l'échelle [unité de la couche] ;
- **col** : la couleur de bordure ;
- **fill** : les deux couleurs de fond ;
- **plot.grid** : gère la compatibilité avec les différentes syntaxes (doit être défini à la valeur **FALSE** pour que la fonction soit compatible avec la syntaxe conventionnelle de type *Painter's Model*).

```
plot(FR_ctr_L2E)
SpatialPolygonsRescale(layout.scale.bar(), offset = c(430000, 2100000),
                       fill = c(NA, "cyan4"), scale = 3e5, plot.grid = FALSE)
text(410000 + c(0, 3e5), rep(2100000, 2) + 7e4, labels = c("0", "300 km"), cex = 1.4)
```



Graticules. On peut définir les graticules grâce à la fonction **gridlines()**. Cette fonction prend un objet **sp** en argument principal, et elle renvoie un objet de classe **SpatialLines** que l'on peut dessiner ensuite avec **plot()**. Dans l'exemple suivant, on souhaite réaliser la carte dans la projection LAEA (code EPSG 3035), mais représenter les graticules en WGS 84 (code EPSG 4326). Pour ce faire, il faut donc définir les graticules à partir d'une couche en WGS 84, puis la convertir ensuite en LAEA. On peut définir des étiquettes aux graticules avec la fonction **labels()** et les dessiner avec **text()**. Notez que le package **rgdal** (Bivand *et al.*, 2014) propose la fonction **llgridlines()**, plus simple d'utilisation (§ 6.1.4.1).

```
gl_W84  <- gridlines(FR_ctr_W84)
class(gl_W84)
## [1] "SpatialLines"
## attr(,"package")
## [1] "sp"
gl_LAEA <- spTransform(gl_W84, CRSobj = CRS("+init=epsg:3035"))
plot(FR_ctr_LAEA)
plot(gl_LAEA, col = "cyan4", add = TRUE)
```



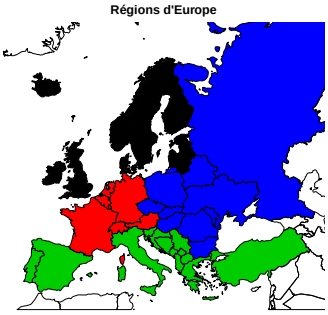
6.1.1.3 Carte typologique

De manière générale, pour colorer les entités d'un objet vectoriel de classe **sp**, il faut affecter une valeur a chacune des entités à colorer. La règle de recyclage de R s'applique au vecteur de couleurs, c'est-à-dire que si le vecteur de couleurs est trop court, il sera répété autant de fois que nécessaire. Par conséquent, si l'on attribue qu'une seule valeur, toutes les entités seront de la même couleur. La règle de recyclage habituelle s'applique donc à la fonction **plot()** du package **sp**.

Il est possible d'utiliser les données contenues dans la table attributaire pour gérer une palette de couleurs, ce qui permet de créer des cartes typologiques ou des cartes choroplèthes, comme nous allons le voir ci-après. Cette méthode s'applique à n'importe quel type d'objet vectoriel de classe **sp**, que ce soit **SpatialPoints(DataFrame)**, **SpatialMultiPoints(DataFrame)**, **SpatialLines(DataFrame)** ou **SpatialPolygons(DataFrame)**.

Il est ainsi possible de créer une carte typologique en fournissant simplement un facteur à l'argument **col** de la fonction **plot()**.

```
plot(EU_ctr1_LAEA, col = EU_ctr1_LAEA@data$part, main = "Régions d'Europe")
```

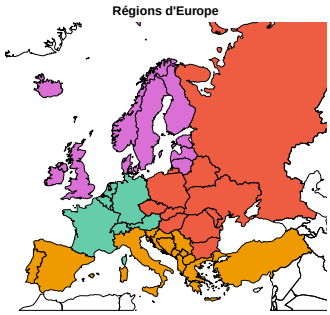


Les différentes modalités de la variable qualitative ainsi représentée prennent les couleurs par défaut renvoyées par la fonction **palette()** (autant de couleurs utilisées que nécessaires, et par ordre d'apparition).

```
palette()
## [1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"  "gray"
```

Si l'on souhaite choisir les couleurs, il y a deux possibilités, comme d'usage avec les graphiques conventionnels. La première consiste à modifier les valeurs de la palette. Pour cela, il suffit simplement de fournir un vecteur de couleurs à la fonction **palette()**. On pourra, par la suite, la réinitialiser avec la commande **palette("default")**.

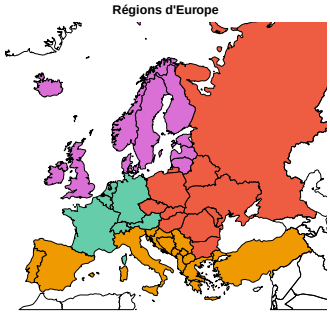
```
pal_col <- c("orchid", "aquamarine3", "orange2", "tomato2")
palette(pal_col)
palette()
## [1] "orchid"      "aquamarine3" "orange2"     "tomato2"
plot(EU_ctr1_LAEA, col = EU_ctr1_LAEA@data$part, main = "Régions d'Europe")
```




```
palette("default")
palette()
## [1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"  "gray"
```

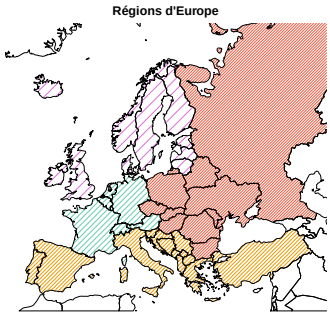
La seconde solution consiste à créer un vecteur de chaînes de caractères de noms couleurs correspondant aux différentes modalités du facteur initial. Pour cela, il suffit de redéfinir les niveaux du facteur avec l'argument **labels** de la fonction **factor()**. Le nouveau facteur ainsi créé doit ensuite être converti en caractères, sans quoi la palette par défaut sera utilisée en lieu et place des couleurs choisies par l'utilisateur.

```
col_vec <- as.character(factor(EU_ctr1_LAEA@data$part, labels = pal_col))
plot(EU_ctr1_LAEA, col = col_vec, main = "Régions d'Europe")
```



Si l'on désire hachurer l'intérieur de polygones plutôt que de les colorer, c'est possible. Pour cela, il suffit d'utiliser les arguments **angle**, **density** et **col** de la fonction **plot()**, qui permettent de gérer respectivement l'angle d'inclinaison des hachures, leur densité et leur couleur. Attention, certains formats de sortie graphique (e.g. le format PDF) n'autorisent pas la colorisation des hachures ; elles apparaissent alors en noir. Il est bien sûr possible de combiner la coloration de polygones et les hachures en superposant des couches successives (§ 6.1.1.6).

```
col_vec <- as.character(factor(EU_ctr1_LAEA@data$part, labels = pal_col))
plot(EU_ctr1_LAEA, angle = 45, density = as.numeric(EU_ctr1_LAEA@data$part)*8, col = col_vec,
     main = "Régions d'Europe")
```

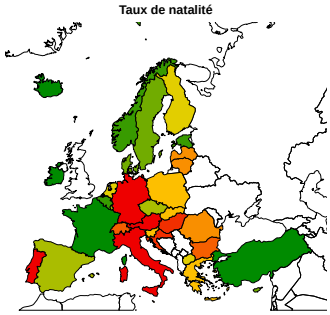


6.1.1.4 Carte choroplèthe

De cette manière, on peut également réaliser une carte choroplèthe en colorant des polygones en fonction des valeurs d'une variable quantitative. Pour ce faire, on peut discrétiser la variable à l'aide de la fonction **cut()**, et l'on fournit un vecteur de noms de couleurs à l'argument **labels**. Comme la fonction **cut()** renvoie un facteur, il faut convertir de dernier en chaîne de caractères avant de le fournir à l'argument **col** de la fonction **plot()**, sans quoi le facteur sera automatiquement converti au format numérique et ce sera alors la palette par défaut qui sera appliquée.

```
pal_nb <- 10
pal_col <- colorRampPalette(c("red2", "gold", "green4"))(pal_nb)
birth_rate <- EU_ctr1_LAEA@data$birth_rate
col_vec <- as.character(cut(birth_rate, include.lowest = TRUE,
                            breaks = quantile(birth_rate, probs = seq(0, 1, length.out = pal_nb+1),
```

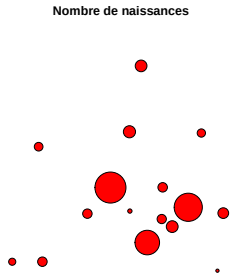
```
                                        na.rm = TRUE),
                          labels =  pal_col))
plot(EU_ctr1_LAEA, col = col_vec, main = "Taux de natalité")
```



6.1.1.5 Carte en symboles proportionnels

On peut tracer une carte en symboles proportionnels en jouant simplement avec les arguments **pch** et **cex** de la fonction **plot()**.

```
plot(EU_labpt_LAEA, pch = 21, bg = "red", cex = sqrt(EU_ctr1_LAEA@data$birth/2e4),
     main = "Nombre de naissances")
```

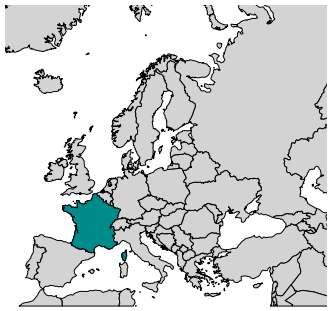


6.1.1.6 Superposer des couches

Avec cette syntaxe conventionnelle, il n'est pas possible de réaliser de "projection à la volée", c'est-à-dire que, pour pouvoir être superposées, toutes les couches doivent être dans le même système de coordonnées.

Contrairement à ce qui est fait de manière courante avec les graphiques conventionnels, dans le cas de l'utilisation des données spatiales, il est tout à fait possible de superposer des couches en faisant plusieurs appels successifs aux fonctions **plot()** ou **image()**. En effet, d'habitude, on effectue un seul appel à **plot()** (ou **image()**), puis les appels suivants se font avec des fonctions telles que **points()** ou **lines()**. Ici, une fois le premier appel réalisé, il suffit simplement d'utiliser l'argument **add = TRUE** de la fonction **plot()** pour autoriser les couches à se superposer. On peut toutefois se servir de fonctions telles que **points()** ou **lines()** pour les superposer avec des objets spatiaux, mais elles devront, quant à elles, être utilisées avec des objets conventionnels (i.e. **vector**, **matrix**, **data.frame**). Notez qu'il n'y a aucune difficulté particulière pour superposer des données vectorielles avec des données matricielles.

```
plot(EU_ctr1_LAEA, col = "lightgrey")
plot(FR_ctr_LAEA, col = "cyan4", add = TRUE)
```



6.1.2 Package raster

Comme nous l'avons vu précédemment, le package **raster** (Hijmans, 2015) définit les classes et méthodes de données spatiales de type matriciel les plus utilisées dans R. Outre les outils de traitement de données, il fournit

les fonctionnalités, utilisant la syntaxe de type *Painter's Model*, permettant de tracer des cartes à partir des objets de classe **raster**.

```
library(raster)
```

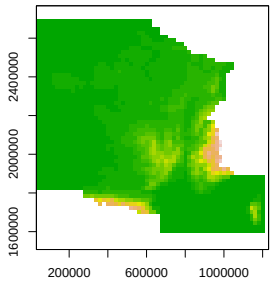
6.1.2.1 Affichage

Comme pour des données matricielles de classe **sp**, l'affichage des données matricielles **raster** se fait en appelant les fonctions **plot()** ou **image()**. Si l'on utilise la dernière fonction mentionnée, il faut bien veiller à utiliser l'argument **asp = 1** pour que la carte ne soit pas déformée.

La fonction **plot()** du package **raster** présente de nombreux arguments. En voici quelques-uns :

- **maxpixels** : le nombre maximal de pixels dessinés (**500000**, par défaut) ;
- **breaks** : les limites des classes ;
- **col** : la palette de couleurs des valeurs ou des classes définies par **breaks** (1 couleur par valeur possible du raster et non une valeur par pixel) ;
- **alpha** : le facteur de transparence ;
- **colNA** : la couleur des valeurs manquantes ;
- **ext** : l'emprise géographique de la carte ;
- **interpolate** : l'interpolation des valeurs du raster ;
- **main** : le titre de la carte ;
- **legend** : trace ou non la légende (mais la marge reste présente) ;
- **axes** : trace ou non les axes.

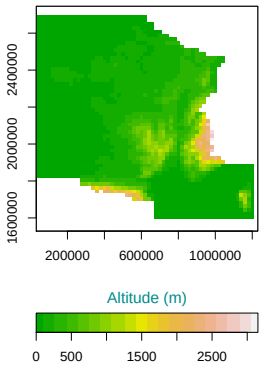
```
plot(FR_MNT20000_L2E, col = terrain.colors(20), legend = FALSE)
```



Pour connaître l'ensemble des paramètres, il convient de consulter l'aide de la fonction **image.plot()** du package **fields** (Nychka *et al.*, 2014). On peut tout de même noter que cette fonction propose, entre autres, les arguments suivants :

- **horizontal** : la position horizontale ou non de la légende ;
- **legend.only** : ajoute la légende à un graphique existant ;
- **legend.args** : la liste d'arguments relatifs à la mise en forme du titre de la légende ;
- **legend.mar** : la marge de la légende ;
- **legend.lab** : le titre de la légende ;
- **legend.shrink** : la longueur relative de la légende par rapport au graphique ;
- **legend.width** : la largeur de la légende.

```
plot(FR_MNT20000_L2E, col = terrain.colors(20),
     horizontal = TRUE,
     legend.shrink = 1.0, legend.width = 1.2,
     legend.args = list(text = "Altitude (m)", col = "cyan4", cex = 1.2, side = 3, line = 0.4))
```

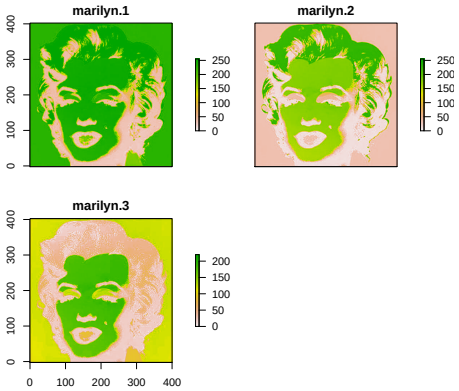


On peut, bien entendu, utiliser directement la fonction **`image.plot()`**. Notez que tous les arguments de la fonction **`image.plot()`** ne sont pas disponibles dans **`plot()`**. C'est notamment le cas de :

- **`legend.line`** : la distance à laquelle est dessinée la légende ;
- **`nlevel`** : le nombre de couleurs utilisées dans la légende ;
- **`border`** : la couleur de bordure autour des pixels ;
- **`lwd`** : la largeur de bordure autour des pixels.

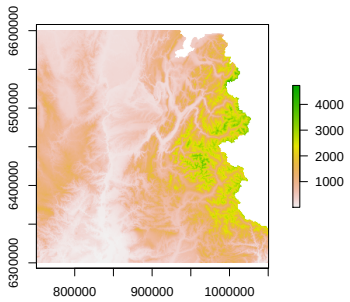
Cette fonction permet également de gérer les rasters multicouches de classes **`RasterBrick`** ou **`RasterStack`**. Elle trace automatiquement les divers couches matricielles et leur associe une échelle commune.

```
class(marylin)
## [1] "RasterBrick"
## attr(,"package")
## [1] "raster"
plot(marylin)
```



On peut zoomer sur une emprise, sans pour autant découper le raster, grâce à la fonction **`zoom()`**. L'argument **`ext`** définit l'emprise, en prenant un objet de classe **`extent`**. Il faut préférer cette solution à l'utilisation des arguments **`xlim`** et **`ylim`** des fonctions **`plot()`** ou **`image()`** car, dans ce cas, l'ensemble des pixels sera inutilement dessiné, même ceux en dehors des limites du graphique (et alors invisibles), ce qui peut poser problème pour les rasters de grande taille.

```
EXT_sub
## class       : Extent
## xmin        : 750000
## xmax        : 1050000
## ymin        : 6300000
## ymax        : 6600000
zoom(FR_MNT1000_L93, ext = EXT_sub, new = FALSE)
```

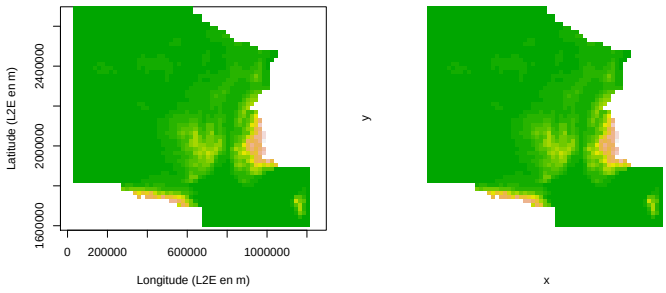


6.1.2.2 Éléments contextuels

Titre. Il se définit dans l'argument `main` des fonctions `plot()` et `image()`, lors de leur premier appel. On peut également utiliser la fonction `title()` du package `graphics` (R Core Team, 2016) (§ 6.1.3.2).

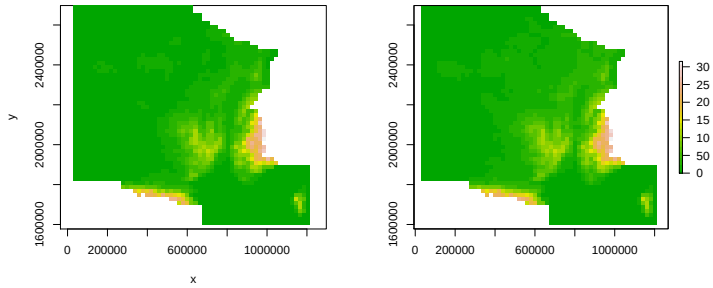
Axes. Pour définir les étiquettes des axes, il faut utiliser les arguments `xlab` et `ylab`. Par défaut, les axes sont affichés (`axes = TRUE`) pour les fonctions `plot()` et `image()` appliquées à des données de classe `raster`. On peut personnaliser l'affichage des axes en utilisant la fonction `axis()` du package `graphics` (R Core Team, 2016) (§ 6.1.3.2).

```
par(mfrow = c(1, 2))
image(FR_MNT20000_L2E, col = terrain.colors(20), asp = 1,
      xlab = "Longitude (L2E en m)", ylab = "Latitude (L2E en m)")
image(FR_MNT20000_L2E, col = terrain.colors(20), asp = 1,
      axes = FALSE)
```



Légende. Par défaut, la fonction `plot()` trace la légende et la fonction `image()` ne le fait pas. On peut empêcher l'affichage avec la fonction `plot()` en utilisant l'argument `legend = FALSE`.

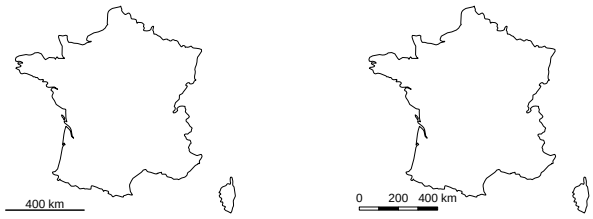
```
par(mfrow = c(1, 2))
image(FR_MNT20000_L2E, col = terrain.colors(20), asp = 1)
plot(FR_MNT20000_L2E, col = terrain.colors(20))
```



Échelle. Le package `raster` propose la fonction `scalebar()` pour tracer l'échelle. Cette fonction est très facile d'utilisation et est à préférer à celle proposée dans le package `sp` (§ 6.2.1.3). Deux types de représentations sont disponibles : un simple trait horizontal ou une barre dont le nombre de subdivisions peut être choisi.

```
par(mfrow = c(1, 2))
plot(FR_ctr_L2E)
scalebar(400e3, type = "line", label = "400 km")
```

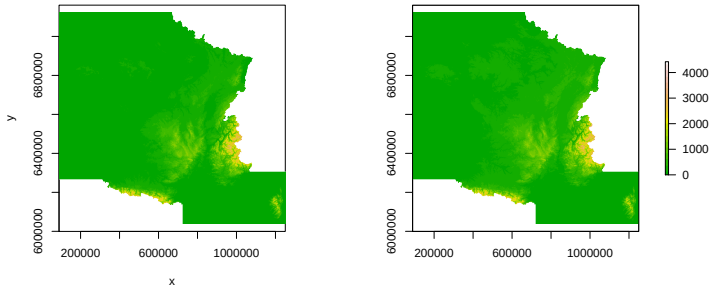
```
plot(FR_ctr_L2E)
scalebar(400e3, type = "bar", divs = 4, label = c("0", "200", "400 km"))
```



6.1.2.3 Vitesse d'affichage de la carte

Comme nous venons de le voir, pour dessiner un raster, on peut utiliser les fonctions **image()** ou **plot()**. Cependant, les vitesses d'exécution de ces fonctions ne sont pas du tout les mêmes ; la fonction **image()** est, en effet, beaucoup plus rapide pour dessiner une carte avec des objets de classe **raster**.

```
par(mfrow = c(1, 2))
system.time(image(FR_MNT1000_L93, col = terrain.colors(20), asp = 1))
##    user  system elapsed
##   0.343   0.007   0.378
system.time( plot(FR_MNT1000_L93, col = terrain.colors(20), asp = 1))
```



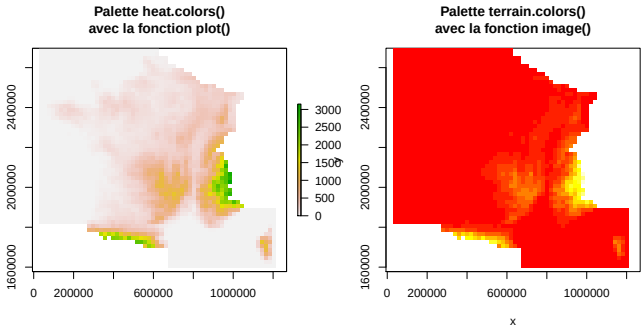

```
##    user  system elapsed
##   1.525   0.101   1.664
```

Par ailleurs, afin de ne pas prendre trop de temps, quand un raster est tracé, le nombre de pixels dessinés est, par défaut, limité à 500 000. Il est toutefois possible d'augmenter cette valeur en modifiant l'argument **maxpixels**, afin d'améliorer la qualité du rendu de la carte ou, au contraire la réduire afin d'accélérer la production de la carte.

6.1.2.4 Colorer les mailles d'un raster

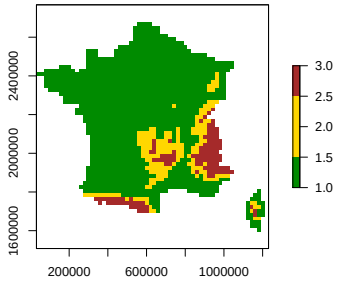
Par défaut, la fonction **plot()** utilise la palette **terrain.colors()**. La fonction **image()**, quant à elle, utilise la palette **heat.colors()**.

```
par(mfrow = c(1, 2))
plot(FR_MNT20000_L2E , main = "Palette heat.colors()\navec la fonction plot()")
image(FR_MNT20000_L2E, main = "Palette terrain.colors()\navec la fonction image()", asp = 1)
```



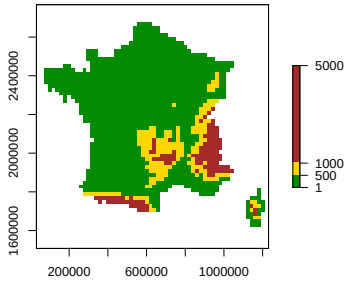
Pour les données **raster**, le vecteur de couleurs doit avoir une longueur égale au nombre de valeurs possibles des pixels, et non pas une longueur égale au nombre de pixels lui-même. Dans l'exemple suivant on classifie un raster d'altitude en 3 catégories *via* la fonction **cut()** (par défaut la valeur la plus basse est exclue : **include.lowest = FALSE**) ; on affecte donc à la fonction **plot()** un vecteur de 3 couleurs. Par défaut, ce sont les numéros des classes qui apparaissent dans la légende (ici, 1, 2 ou 3).

```
FR_MNT20000_L2E_cut4 <- cut(FR_MNT20000_L2E, breaks = c(0, 500, 1000, 5000))
plot(FR_MNT20000_L2E_cut4, col = c("green4", "gold", "brown"))
```



On peut aussi directement utiliser l'argument **breaks** de la fonction **plot()** (par défaut la valeur la plus basse est incluse), et fournir alors une couleur par classe. Cette méthode présente l'avantage de conserver les valeurs brutes dans la légende.

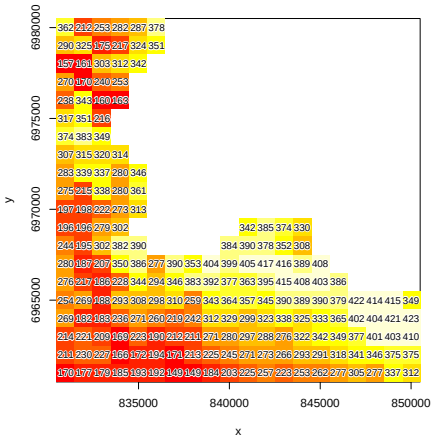
```
plot(FR_MNT20000_L2E, breaks = c(1, 500, 1000, 5000), col = c("green4", "gold", "brown"))
```



6.1.2.5 Écrire des valeurs dans les mailles d'un raster

On peut écrire les valeurs des mailles d'un raster à l'aide de la fonction **text()**. Le texte est alors automatiquement tracé au centre de chacun des pixels. Pour tenter d'améliorer la lisibilité, on peut entourer le texte d'un halo (**halo = TRUE**).

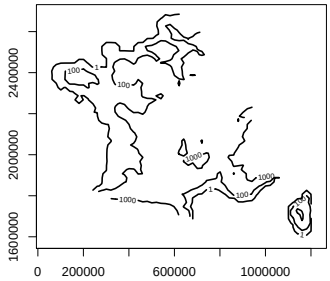
```
FR_MNT1000_L93_clip_EXT <- crop(FR_MNT1000_L93, extent(8.3e5, 8.5e5, 6.96e6, 6.98e6))
image(FR_MNT1000_L93_clip_EXT, asp = 1)
text(FR_MNT1000_L93_clip_EXT, cex = 0.8, halo = TRUE)
```



6.1.2.6 Dessiner des isolignes

On peut tracer les isolignes à partir des valeurs d'un raster à l'aide de la fonction **contour()**. On peut définir manuellement les courbes de niveau que l'on souhaite tracer à l'aide de l'argument **levels**. Pour connaître l'ensemble des arguments disponibles, il faut se référer à fonction **contour()** du package **graphics** (R Core Team, 2016). Dans l'exemple suivant, on dessine les isoclines définies à partir d'un MNT.

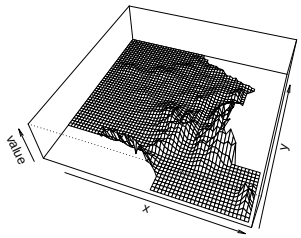
```
contour(FR_MNT20000_L2E, levels = c(0, 1, 100, 1000, 10000), lwd = 2)
```



6.1.2.7 Dessiner un raster en 3D

La fonction **persp()** du package **raster** permet de dessiner facilement un raster en perspective. On choisit l'angle d'affichage en définissant la direction d'observation ; l'argument **theta** correspond à l'azimute et **phi** correspond à la colatitude.

```
persp(FR_MNT20000_L2E, expand = 0.25, theta = 20, phi = 45)
```



Notez que la sortie graphique ainsi générée n'est pas interactive. En effet, l'utilisateur ne peut pas faire pivoter la figure avec le pointeur de sa souris. S'il souhaite l'afficher sous un autre point de vue, il devra faire un nouvel appel à la fonction en définissant des angles différents.

6.1.2.8 Dessiner un raster de couleurs RGB

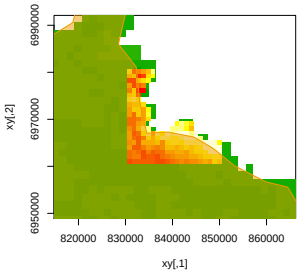
Si l'on souhaite tracer le raster de mélange des 3 bandes (rouge verte et bleue), il faut utiliser la fonction **plotRGB()**.

```
marylin
## class       : RasterBrick
## dimensions  : 400, 400, 160000, 3  (nrow, ncol, ncell, nlayers)
## resolution  : 1, 1  (x, y)
## extent      : 0, 400, 0, 400  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## data source : /home/irstea/Samba/smbe/Data/boulot/misc/scientific_doc/R/formation/irstea/033-geomatique_LIV/01_r
## names       : marilyn.1, marilyn.2, marilyn.3
## min values  :         0,         0,         0
## max values  :       255,       255,       255
plotRGB(marylin)
```

6.1.2.9 Superposer des couches

Avec cette syntaxe conventionnelle, il n'est pas possible de réaliser de "projection à la volée". Pour superposer les couches, il suffit de réaliser des appels successifs aux fonction **plot()**, **image()** et/ou **contour()** en utilisant l'argument **add = TRUE**. Par ailleurs, superposer des données matricielles **raster** et des données vectorielles **sp** ne pose aucun problème.

```
plot(FR_MNT1000_L93_clip_EXT@extent+2e4, col = NA, asp = 1)
image(FR_MNT1000_L93, col = terrain.colors(20), asp = 1, add = TRUE)
image(FR_MNT1000_L93_clip_EXT, asp = 1, add = TRUE)
plot(FR_ctr_L93, border = "orange2",
     col = adjustcolor("orange2", alpha.f = 0.5), add = TRUE)
```



6.1.3 Package graphics

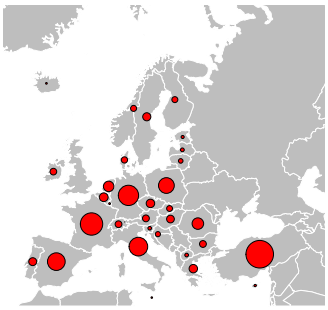
Le package **graphics** (R Core Team, 2016) est la transcription dans R de la syntaxe de type *Painter's Model* pour permettre la production de sorties graphiques. De nombreuses fonctions de ce package peuvent être utilisées en complément de celles, spécifiques aux données spatiales, fournies par les packages **sp** et **raster**.

6.1.3.1 Carte en symboles proportionnels

Il est possible d'ajouter des symboles sur une carte à l'aide la fonction **symbols()**. Cette fonction propose plusieurs types de représentations :

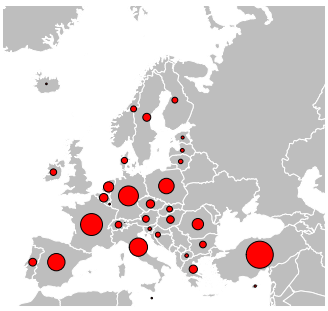
- **circles** : des cercles ;
- **squares** : des carrés ;
- **rectangles** : des rectangles ;
- **stars** : des étoiles ;
- **thermometers** : des thermomètres ;
- **boxplots** : des boîtes à moustaches.

```
pts_size <- sqrt(EU_ctr1_LAEA@data$birth) * 200
plot(EU_ctr1_LAEA, col = "grey", border = "white")
symbols(coordinates(EU_ctr1_LAEA), circles = pts_size, bg = "red", inches = FALSE, add = TRUE)
```



Une autre solution consiste à utiliser la fonction **points()** et jouer avec l'argument **cex** pour gérer la taille des symboles. Ici on a choisi de représenter des points, mais on peut utiliser n'importe quel type de symboles autorisé par la fonction **points()** *via* l'argument **pch**.

```
pts_size <- sqrt(EU_ctr1_LAEA@data$birth) * 200
plot(EU_ctr1_LAEA, col = "grey", border = "white")
points(coordinates(EU_ctr1_LAEA), pch = 21, cex = pts_size/4e4, bg = "red")
```



6.1.3.2 Éléments contextuels

Titre. On peut utiliser la fonction `title()` pour ajouter un titre à une carte.

Axes. (R Core Team, 2016) Pour définir les noms des axes, on utilise les arguments `xlab` et `ylab`. On peut masquer les axes avec les arguments `xaxt = "none"` et `yaxt = "none"` de la fonction `par()`. Pour personnaliser l'affichage des axes on peut utiliser la fonction `axis()`.

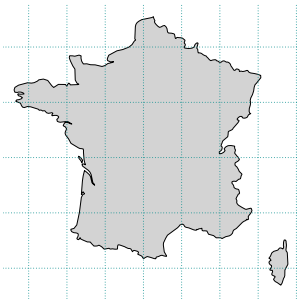
Légende. L'affichage de la légende se fait par un simple appel à la fonction `legend()`, comme avec un graphique conventionnel.

Positionner des étiquettes. On peut ajouter du texte sur la carte à l'aide de la fonction `text()`. Dans l'exemple suivant, on commence par récupérer les centroïdes des polygones grâce à la fonction `coordinates()`, et l'on fournit leurs coordonnées à `text()`. Dans cette fonction, l'argument `pos` permet de choisir la position à partir de coordonnées spécifiées (`1` : dessous, `2` : à gauche, `3` : en haut, `4` : à droite) et l'argument `offset` permet de gérer le décalage de l'étiquette (exprimé en taille de caractère ; nécessite au préalable la définition de `pos`). Avec cette fonction, on ne peut pas empêcher les étiquettes de se chevaucher, ce qui nuit à la lisibilité de la carte. D'autres packages proposent des fonctions qui permettent de réduire ou d'éviter les chevauchements (§ 6.1.5.2).

```
EU_ctr1_LAEA_labpt <- coordinates(EU_ctr1_LAEA)
plot(EU_ctr1_LAEA, col = "lightgray", border = "white")
points(EU_ctr1_LAEA_labpt, pch = 21, bg = "cyan4", cex = 0.4)
text(EU_ctr1_LAEA_labpt, labels = EU_ctr1_LAEA@data$name, pos = 3, offset = 0.3, cex = 0.6)
```

Graticules. On peut ajouter des graticules en appelant la fonction `grid()`. Mais comme la grille produite présente un maillage régulier, elle ne correspond véritablement aux graticules (i.e. l'ensemble des méridiens et parallèles portés par une carte) que pour une représentation dans système non projeté (e.g. le WGS 84), peu propice à la représentation cartographique (le package `rgdal` propose une fonction tenant compte du système de coordonnées utilisé).

```
plot(FR_ctr_W84, col = "lightgray")
grid(col = "cyan4")
```



6.1.3.3 Gérer la couleur du fond de carte

Pour définir la couleur de fond, il faut utiliser l'argument **bg** de la fonction **par()**, qui régit les paramètres graphiques.

```
par(bg = "lightblue")
plot(FR_ctr_L2E, border = "orange2",
     col = adjustcolor("orange2", alpha.f = 0.5))
```

De manière plus générale, sauf exception, il n'y a pas de problème pour utiliser les différentes possibilités de la fonction **par()**, afin de gérer les paramètres graphiques comme à l'accoutumé.

6.1.4 Package rgdal

Comme nous l'avons déjà vu, le package **rgdal** (Bivand *et al.*, 2014) propose une implémentation des outils GDAL/OGR (GDAL Development Team, 2012) et PROJ.4 (Evenden *et al.*, 2015) dans R. En cela, il permet de lire et d'écrire de nombreux formats de fichiers spatiaux, qu'ils soient vectoriels ou matriciels, et par ailleurs, il permet la gestion des systèmes de coordonnées spatiaux. Outre ces fonctionnalités, il propose néanmoins une fonction cartographique très pratique permettant de dessiner des graticules.

```
library(rgdal)
```

6.1.4.1 Éléments contextuels

Graticules. On peut ajouter les graticules grâce à la fonction **llgridlines()**. L'unité des étiquettes peut être définie par l'utilisateur, mais le système de coordonnées choisi pour les étiquettes ne doit pas être un système projeté (e.g. WGS 84, qui est la système défini par défaut). L'argument **side**, quant à lui, permet de choisir la position des étiquettes (on peut aussi choisir de les masquer).

```
plot(EU_ctr1_LAEA)
llgridlines(EU_ctr1_LAEA, side = "WS", col = "cyan4", lty = 3)
```

6.1.5 Package maptools

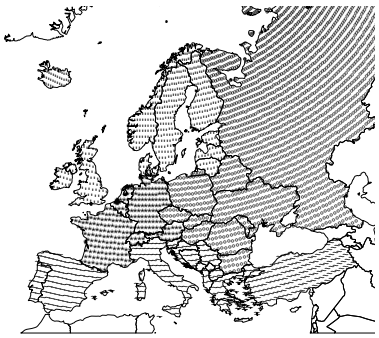
Le package **maptools** (Bivand & Lewin-Koh, 2015) est spécialisé dans la lecture et l'écriture de fichiers au format Shapefile, ainsi que dans la conversion des objets R au format **sp** depuis et vers d'autres formats définis par d'autres packages tels que **PBSmapping**, **spatstat**, **maps**, **RArcInfo**, **Mondrian**, etc. Par ailleurs, il propose également quelques fonctions intéressantes pour la représentation cartographique. Nous allons en donner un aperçu ici.

```
library(maptools)
```

6.1.5.1 Dessiner des symboles dans un polygone

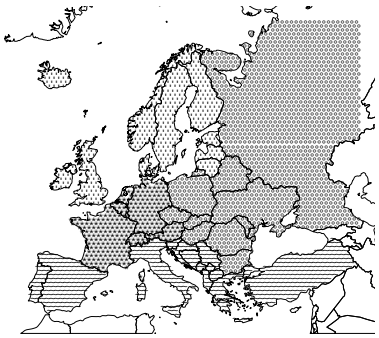
Pour dessiner des symboles dans des polygones, on peut utiliser la fonction **symbolsInPolys()**. Elle ne fonctionne que si la couche, sur laquelle la fonction est appliquée, est en WGS 84 (code EPSG 4326). Dans l'exemple ci-dessous, on dispose d'une couche de l'Europe exprimée en LAEA (code EPSG 3035). On la convertit donc en WGS 84, on applique ensuite la fonction **symbolsInPolys()**, puis on transforme la couche de symboles ainsi crée en LAEA, afin de pouvoir réaliser une représentation cartographique dans le système de coordonnées initial. Cela a pour effet néfaste de distordre la régularité des symboles. Indépendamment ce problème, un des défauts du résultat obtenu est que les symboles peuvent dépasser certains contours de polygones, ce qui nuit fortement à la lisibilité de la carte.

```
EU_ctr1_W84 <- spTransform(EU_ctr1_LAEA, CRS("+init=epsg:4326"))
symb_type <- c("'", "*", "~", "°")
np <- sapply(slot(EU_ctr1_W84, "polygons"), function(x) length(slot(x, "Polygons")))
EU_ctr1_W84_symb <- symbolsInPolys(EU_ctr1_W84, 2,
                                   symb = as.character(factor(EU_ctr1_W84$part, labels = symb_type)))
proj4string(EU_ctr1_W84_symb) <- "+init=epsg:4326"
EU_ctr1_LAEA_symb <- spTransform(EU_ctr1_W84_symb, CRS("+init=epsg:3035"))
plot(EU_ctr1_LAEA)
plot(EU_ctr1_LAEA_symb, pch = as.character(EU_ctr1_LAEA_symb$symb), add = TRUE)
```



Notez que pour obtenir un résultat similaire, mais sans distorsion, on peut appliquer la fonction **spsample()**. On travaille ici directement dans le système de coordonnées dans lequel on souhaite réaliser la carte, cela évite donc les distorsions dans la répartition des symboles. En revanche, cela ne règle pas le problème du débordement des symboles en dehors des frontières des polygones.

```
EU_ctr1_LAEA_symb <- spsample(EU_ctr1_LAEA, 5e3, type = "hexagonal")
symb_tab <- data.frame(part = c("Eastern Europe", "Northern Europe", "Southern Europe", "Western Europe"),
                       pch  = c(             "°",               "'",               "~",              "*"),
                       stringsAsFactors = FALSE)
plot(EU_ctr1_LAEA)
invisible(sapply(seq_len(nrow(symb_tab)), function(x) {
  tmp <- EU_ctr1_LAEA_symb[EU_ctr1_LAEA[EU_ctr1_LAEA@data$part %in% symb_tab$part[x], ], ]
  plot(tmp, pch = symb_tab$pch[x], add = TRUE)
}))
```



6.1.5.2 Éléments contextuels

Légende. Le package **maptools** propose la fonction **leglabs()** qui facilite la mise en forme du texte de la légende. Dans l'exemple ci-dessous, on commence par préparer les limites de classes et la palette de couleurs souhaitées, et l'on attribue une couleur à chacun des polygones.

```
pal_ncl <- 5
pal_brk <- quantile(EU_ctr1_LAEA@data$birth_rate, probs = seq(0, 1, len = pal_ncl+1), na.rm = TRUE)
pal_brk <- round(pal_brk, dig = 1)
pal_col <- colorRampPalette(c("red2", "gold", "green4"))(pal_ncl)
pop_col <- pal_col[findInterval(EU_ctr1_LAEA@data$birth_rate, pal_brk, all.inside = TRUE)]
```

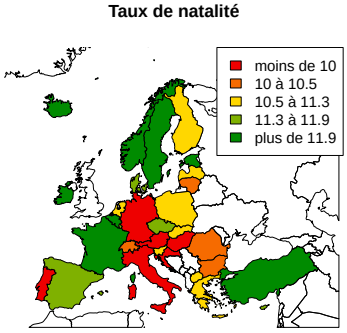
On peut, à présent, construire les étiquettes de la légende en définissant plusieurs arguments :

- **under** : la chaîne de caractères désignant ce qui est sous la première limite ;
- **over** : la chaîne de caractères désignant ce qui est au-dessus de la dernière limite ;
- **between** : la chaîne de caractères qui séparent les limites.

```
pop_leg <- leglabs(pal_brk, under = "moins de", over = "plus de", between = "à")
```

On obtient alors la carte suivante :

```
plot(EU_ctr1_LAEA, col = pop_col, main = "Taux de natalité")
legend("topright", legend = pop_leg, fill = pal_col, bg = "white")
```



Positionner des étiquettes. On peut ajouter du texte sur la carte avec une fonction qui utilise une routine d'optimisation permettant de trouver de bons emplacements aux étiquettes et ce, afin d'éviter les chevauchements. Dans l'exemple suivant, on commence par récupérer les centroïdes des polygones grâce à la fonction **coordinates()**, et l'on fournit leurs coordonnées à la fonction **pointLabel()**, dans laquelle l'argument **offset** permet de gérer le décalage de l'étiquette par rapport aux coordonnées spécifiées.

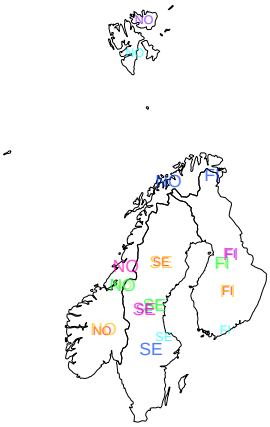
```
EU_ctr1_LAEA_labpt <- coordinates(EU_ctr1_LAEA)
plot(EU_ctr1_LAEA, col = "lightgray", border = "white")
points(EU_ctr1_LAEA_labpt, pch = 21, bg = "cyan4", cex = 0.4)
pointLabel(x = EU_ctr1_LAEA_labpt[, 1], y = EU_ctr1_LAEA_labpt[, 2],
           labels = EU_ctr1_LAEA@data$name, offset = 0, cex = 0.6)
```

Sur la carte, on peut constater que les étiquettes ne sont pas toujours positionnées au même emplacement par rapport au point auquel elles correspondent.

Il est également possible de positionner d'étiquettes le longs de lignes avec la fonction **`lineLabel()`**.

Par ailleurs, concernant le positionnement d'étiquettes à l'intérieur de polygones, on peut utiliser la fonction **`polygonsLabel()`** du package **`rgeos`** (Bivand & Rundel, 2014). Cette dernière propose plusieurs algorithmes de positionnement des étiquettes (voir les arguments **`method`** et **`polypart`**).

```
SC_ctr1_LAEA <- EU_ctr1_LAEA[EU_ctr1_LAEA@data$iso_a2 %in% c("FI", "NO", "SE"),  ]

lab_meth <- c("buffer", "centroid", "random", "inpolygon")
lab_poly <- c("all", "largest")

plot(SC_ctr1_LAEA)
k <- 1
invisible(sapply(seq_along(lab_meth), function(x) {
  sapply(seq_along(lab_poly), function(y) {
    polygonsLabel(SC_ctr1_LAEA, labels = SC_ctr1_LAEA@data$iso_a2,
                  method = lab_meth[x], col = adjustcolor(rainbow(8)[k], alpha = 0.7),
                  polypart = lab_poly[y], cex = sqrt(y)/1.2)
    k <<- k+1
  })
}))
```



6.1.6 Package `cartography`

Le package **`cartography`** (Giraud & Lambert, 2016) est spécialisé dans la production de sorties cartographiques. Les fonctions qu'il propose sont compatibles avec les fonctions graphiques conventionnelles des packages **`sp`** ou **`raster`**. Les fonctions de ce package sont toutes relatives à la représentation d'objets vectoriels (essentiellement des polygones), aucune ne concerne les objets matriciels ; on peut toutefois réaliser sans problème des superpositions de couches avec de tels objets.

Avant de créer ce package, Giraud avait développé le package **`Rcarto`** (2013). Ce dernier manquant cruellement de souplesse, il a été abandonné et remplacé par le package **`cartography`**. Même si **`Rcarto`** est toujours disponible sur le site du CRAN, il est déconseillé de l'utiliser ; nous ne le présenterons donc pas ici.

```
library(cartography)
```

De manière générale, les fonctions de ce package, qui réalisent principalement des tracés de polygones, présentent toutes les mêmes arguments principaux :
- **spdf** : un objet **SpatialPolygonsDataFrame** ;
- **df** : le **data.frame** contenant la table attributaire ;
- **var** : le nom de la colonne de la table attributaire correspondant à la variable d'intérêt.

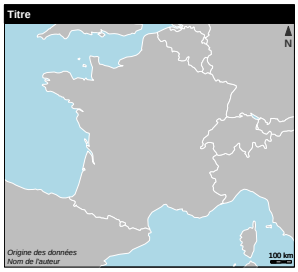
6.1.6.1 Éléments contextuels

Légende. Par défaut, la plupart des fonctions du package **cartography** (e.g. **choroLayer()**, **propSymbolsLayer()**, **discLayer()**, etc.) ajoutent une légende sur la carte produite ; pour ne pas la dessiner, il faut utiliser l'argument **legend.pos = "n"**. Les principaux arguments, permettant de gérer l'affichage de la légende au sein des fonctions cartographiques, sont les suivants :
- **legend.frame = TRUE** : la légende est tracée dans une boîte ;
- **legend.pos** : la position de la légende ;
- **legend.title.txt** : le titre ;
- **legend.title.cex** : la taille du titre ;
- **legend.values.cex** : la taille du texte des valeurs ;
- **legend.values.rnd** : le nombre de décimales des valeurs ;
- **legend.nodata** : l'appellation des valeurs manquantes (absent pour **discLayer()**) ;
- **legend.values.order** : ordre des valeurs (uniquement pour la fonction **typoLayer()**)
- **legend.style** : le style de la légende (**"c"** : compacte ou **"e"** : étendue) (uniquement pour les fonctions **propSymbolsLayer()** et **propTrianglesLayer()**).

Par ailleurs, le package propose plusieurs fonctions permettant d'ajouter une légende adaptée au type de représentation d'une carte déjà existante :
- **legendChoro()** : pour les cartes choroplèthes ;
- **legendGradLines()** : pour une taille des lignes en fonction d'une variable discrétisée ;
- **legendPropLines()** : pour une taille des lignes proportionnelle à une variable ;
- **legendBarsSymbols()** : pour les symboles en barres ;
- **legendCirclesSymbols()** : pour les symboles en cercles ;
- **legendPropTriangles()** : pour les symboles en triangles ;
- **legendSquaresSymbols()** : pour les symboles en carrés.

Orientation, échelle, titre, source des données, etc. La fonction **layoutLayer()**, appelée avant ou après le dessin de la carte, permet de gérer le style de nombreux éléments contextuels de cette dernière, tels que le titre (**title**), le nom de l'auteur (**author**), le nom de la source des données (**sources**), l'échelle (**scale**), l'orientation (**north** ou **south**) ou l'emprise géographique de la carte (**extent** ; définie par un autre objet **sp**). Dans l'exemple suivant, on fait deux appels à la fonction **layoutLayer()**, un avant le dessin de la couche et un autre après. Le premier permet de définir l'emprise de la carte, la couleur de fond, les couleurs du bandeau de titre. Le second appel vient surajouter le titre, le nom de l'auteur, l'origine des données, l'échelle et la flèche du Nord. Ces éléments seraient masqués par la couche de polygones, s'ils étaient tracés lors de la première utilisation de la fonction.

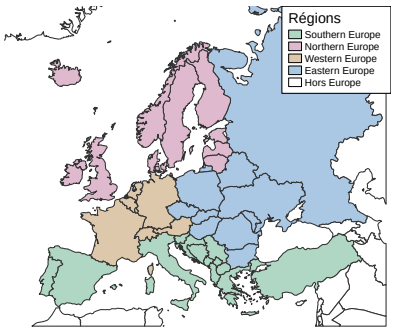
```
layoutLayer(author = "",
            sources = "",
            col = "black", coltitle = "white", bg = "lightblue",
            extent = EU_ctr1_LAEA[EU_ctr1_LAEA@data$name == "France", ])
plot(EU_ctr1_LAEA, col = "grey", border = "white", add = TRUE)
layoutLayer(title = "Titre",
            author = "Nom de l'auteur",
            sources = "Origine des données", scale = 0, north = TRUE)
```



6.1.6.2 Carte typologique

Il est possible de réaliser une carte typologique à l'aide de la fonction **typolayer()**. L'argument **var** correspond au nom d'une la colonne de la table attributaire contenant la variable qualitative que l'on souhaite représenter.

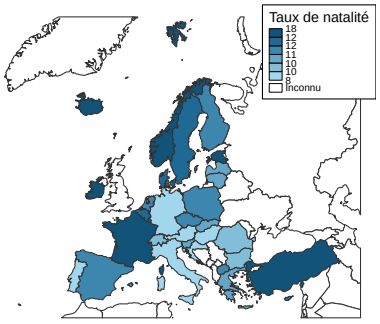
```
typoLayer(spdf = EU_ctr1_LAEA, df = EU_ctr1_LAEA@data, var = "part",
          legend.pos = "topright", legend.frame = TRUE,
          legend.nodata = "Hors Europe", legend.title.txt = "Régions")
```



6.1.6.3 Carte choroplèthe

La fonction **choroLayer()** permet de réaliser une carte choroplèthe. Plusieurs arguments permettent à l'utilisateur de gérer la méthode de discrétisation (manuelle : **breaks** ou automatique : **method** = **"sd"**, **"equal"**, **"quantile"**, **"fisher-jenks"**, **"q6"** ou **"geom"**), les couleurs (**col** et **colNA**) ou le style de la légende.

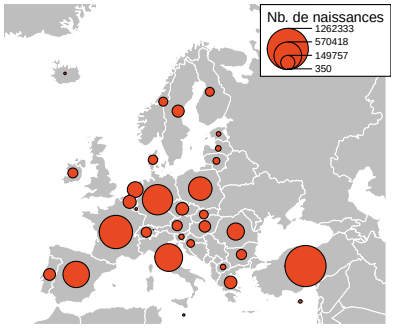
```
choroLayer(spdf = EU_ctr1_LAEA, df = EU_ctr1_LAEA@data, var = "birth_rate",
           legend.pos = "topright", legend.frame = TRUE, legend.nodata = "Inconnu",
           legend.title.txt = "Taux de natalité")
```



6.1.6.4 Carte en symboles proportionnels

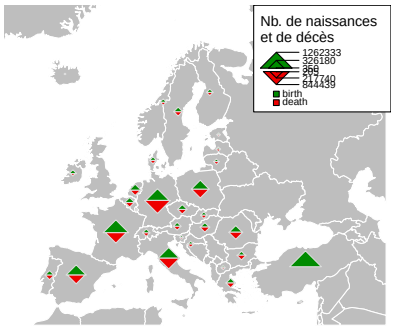
Pour une carte en symboles proportionnels, il faut utiliser la fonction **propSymbolsLayer()**. La couche ainsi produite se superpose à une carte déjà existante. Les premiers arguments sont les mêmes que ceux de la fonction **choroLayer()**. L'argument **symbols** permet de choisir le type des symboles (**"circle"**, **"square"** ou **"bar"**). On peut également gérer le style de la légende (**legend.style** = **"c"** : compacte ou **"e"** : étendue). De nombreux arguments permettent à l'utilisateur de gérer les bornes des classes, la taille des symboles ou le style de la légende. Dans l'exemple suivant, on fournit directement un objet de classe **SpatialPolygonsDataFrame** à l'argument **spdf**, et les symboles sont directement tracés à l'emplacement des centroïdes. On ne peut pas travailler avec un objet de classe **SpatialPointsDataFrame**, même si c'est bien des entités ponctuelles que l'on représente.

```
plot(EU_ctr1_LAEA, col = "grey", border = "white")
propSymbolsLayer(spdf = EU_ctr1_LAEA, df = EU_ctr1_LAEA@data, var = "birth",
                 symbols = "circle", inches = 0.2,
                 legend.style = "c", legend.frame = TRUE,
                 legend.pos = "topright", legend.title.txt = "Nb. de naissances")
```



Pour une carte dessinant des doubles triangles proportionnels, le package **cartography** propose la fonction **propTrianglesLayer()**. Les triangles représentant chacun une variable, on définira les deux colonnes d'intérêt de la table attributaire dans les arguments **var1** et **var2**.

```
plot(EU_ctr1_LAEA, col = "grey", border = "white")
propTrianglesLayer(spdf = EU_ctr1_LAEA, df = EU_ctr1_LAEA@data,
                   var1 = "birth", var2 = "death",
                   col1 = "green4", col2 = "red2",
                   legend.pos = "topright", legend.frame = TRUE,
                   legend.title.txt = "Nb. de naissances\net de décès")
```



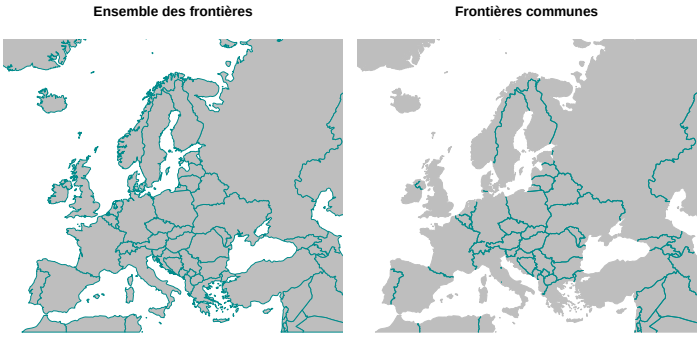
6.1.6.5 Dessiner des frontières

La fonction **getBorders()** permet de récupérer les frontières communes entre des polygones. On lui fournit un objet de classe **SpatialPolygonsDataFrame**, et l'on récupère un objet **SpatialLinesDataFrame**.

```
EU_ctr1_LAEA_bd <- getBorders(EU_ctr1_LAEA)
class(EU_ctr1_LAEA_bd)
## [1] "SpatialLinesDataFrame"
## attr(,"package")
## [1] "sp"
```

Graphiquement, cela donne le résultat suivant :

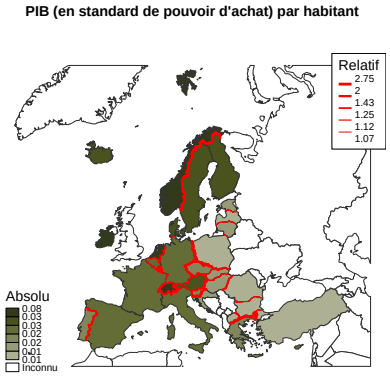
```
par(mfrow = c(1, 2))
plot(EU_ctr1_LAEA, border = "cyan4", col = "grey", main = "Ensemble des frontières")
plot(EU_ctr1_LAEA, border = "grey", col = "grey", main = "Frontières communes")
plot(EU_ctr1_LAEA_bd, col = "cyan4", lwd = 1, add = TRUE)
```



La fonction **discLayer()** permet tracer des discontinuités à partir de l'objet renvoyé par la fonction **getBorders()**. Dans l'exemple suivant, on représente les discontinuités entre pays voisins en termes de différence de PIB (en standard de pouvoir d'achat) par habitant. Plus la différence de PIB est élevée et plus le trait représentant la frontière entre deux pays est large.

De nombreux paramètres permettent de gérer la représentation des frontières. On a notamment :
- **col** : les couleurs des bordures ;
- **nclass** : le nombre de classes souhaitées ;
- **method** : la méthode de discrétisation des classes ;
- **threshold** : la proportion de frontières que l'on souhaite représenter ;
- **sizemin** : l'empattement de la valeur minimale des bordures ;
- **sizemax** : l'empattement de la valeur maximale des bordures ;
- **type** : le type de discontinuité (**"rel"** : relatif ou **"abs"** : absolu).

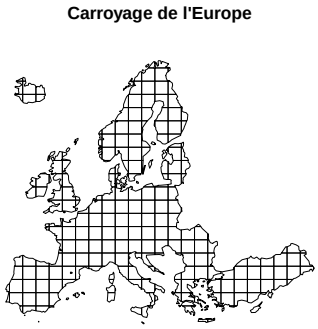
```
choroLayer(EU_ctr1_LAEA, df = EU_ctr1_LAEA@data, var = "gdpps_pop",
           method = "quantile", col = carto.pal(pal1 = "kaki.pal", n1 = 6), nclass = 6,
           legend.title.txt = "Absolu", legend.values.rnd = 2, legend.nodata = "Inconnu")
discLayer(spdf = EU_ctr1_LAEA_bd, df = EU_ctr1_LAEA@data, var = "gdpps_pop",
          col = "red", nclass = 5, method = "quantile",
          threshold = 0.9, sizemin = 0.6, sizemax = 3.2, type = "rel",
          legend.frame = TRUE, legend.pos = "topright", legend.title.txt = "Relatif",
          add = TRUE)
title("PIB (en standard de pouvoir d'achat) par habitant", cex.main = 0.8)
```



6.1.6.6 Carte en carroyage

Il est possible de définir un carroyage, c'est-à-dire de créer une grille régulière intersectant le contour général d'une couche de polygones à l'aide de la fonction **getGridLayer()**. Ceci permet de passer d'un maillage hétérogène à un maillage homogène. La fonction renvoie une liste contenant un **SpatialPolygonsDataFrame** de la grille régulière (dont la table attributaire contient *id* : un identifiant et *cell_area* : les aires des mailles), ainsi qu'un **data.frame** des surfaces intersectées (contenant *id_cell* : l'identifiant des mailles de la grille ; *id_geo* : les identifiant de la couche de polygones et l*area_pct* : la part de la zone du polygone intersectant la maille).

```
EU_grid_LAEA <- getGridLayer(spdf = EU_reg_LAEA, cellsize = 2e5)
names(EU_grid_LAEA)
## [1] "spdf" "df"
plot(EU_grid_LAEA$spdf, main = "Carroyage de l'Europe")
```

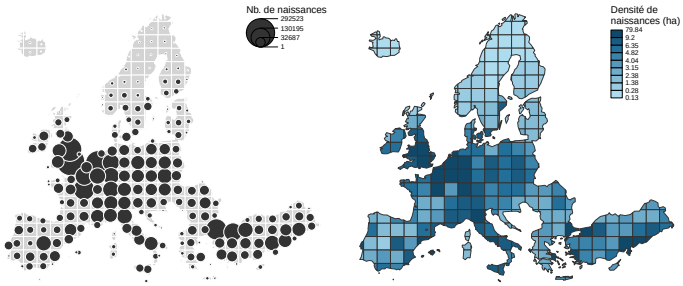


Pour réaliser une carte choroplèthe à partir du carroyage, il faut tout d'abord recalculer les données attributaires sur les mailles de la grille avec la fonction **getGridData()**.

```
EU_grid_LAEA_df <- getGridData(x = EU_grid_LAEA, df = EU_reg_LAEA@data, var = "birth", dfid = NULL)
head(EU_grid_LAEA_df)
##   id_cell    birth birth_density
## 1     102 33363.32  1.205795e-06
## 2     103 18750.98  4.687652e-07
## 3     104 15307.56  3.826814e-07
## 4     105 21767.14  5.441676e-07
## 5     106 59510.13  1.505535e-06
## 6     107 25043.61  1.615528e-06
EU_grid_LAEA_df$birth_density <- EU_grid_LAEA_df$birth_density * 2e5^2 * 1e-04
```

On peut alors réaliser des représentations cartographiques sur la grille calculée avec des symboles proportionnels ou bien par une carte choroplèthe.

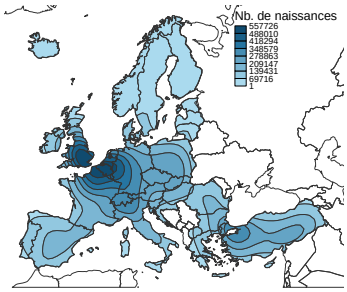
```
par(mfrow = c(1, 2))
plot(EU_grid_LAEA$spdf, col = "lightgrey", border = "white")
propSymbolsLayer(spdf = EU_grid_LAEA$spdf, df = EU_grid_LAEA_df, var = "birth",
                 inches = 0.2, col = "grey20", border = "white",
                 legend.style = "c", legend.pos = "topright", legend.title.txt = "Nb. de naissances")
choroLayer(spdf = EU_grid_LAEA$spdf, df = EU_grid_LAEA_df , var = "birth_density",
           legend.pos = "topright", legend.title.txt = "Densité de\nnaissances (ha)",
           legend.values.rnd = 2)
```



6.1.6.7 Carte lissée

On peut réaliser des cartes lissées à l'aide de la fonction **smoothLayer()** en fonction d'une ou deux variables (**var** et **var2**). Plusieurs arguments permettent de paramétrer le lissage (**typefct**, **span**, **beta** et **resolution**). On peut également ajouter un polygone de masque (**mask**).

```
smoothLayer(spdf = EU_reg_LAEA, EU_reg_LAEA@data,
            var = "birth", span = 2e5, beta = 2,
            mask = EU_reg_LAEA,
            legend.pos = "topright", legend.title.txt = "Nb. de naissances")
plot(EU_ctr1_LAEA, border = "grey20", add = TRUE)
```



6.1.6.8 Superposer des couches

Il n'y a pas ici de recommandation particulière à faire. La superposition de couches se fait de manière conventionnelle par des appels successifs aux différentes fonctions, en utilisant l'argument **`add = TRUE`**, comme avec les fonctions des packages **`sp`** et **`raster`**. Les fonctions du package **`cartography`** sont compatibles avec les fonctions **`plot()`** et **`image()`**.

Afin de combiner des symboles avec une carte typologique ou une carte choroplèthe, on peut appeler les fonctions de manière successive ou bien utiliser les fonctions **`propSymbolsTypoLayer()`** et **`propSymbolsChoroLayer()`**, qui réalisent directement la superposition.

6.1.7 Package `GISTools`

Le package **`GISTools`** (Brunsdon & Chen, 2014), propose quelques outils pour la manipulation de données spatiales et pour la production cartographique.

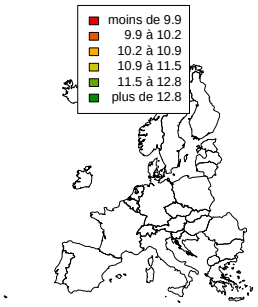
```
library(GISTools)
```

6.1.7.1 Éléments contextuels

Légende. On peut dessiner la légende d'une carte choroplèthe avec la fonction **`choro.legend()`**. Les arguments **`px`** et **`py`**, définissent la position, **`sh`** prend un objet de classe **`shading`**, **`under`**, **`over`** et **`between`** permettent de gérer l'affichage du texte, et **`fmt`** permet de choisir le format d'affichage des valeurs des bornes de classes. Pour créer un objet de classe **`shading`**, il faut utiliser les fonctions **`shading()`** ou **`auto.shading()`**. Un objet de classe **`shading`** correspond à un objet définissant des bornes de classes et des couleurs de classes associées. Attention, contrairement à **`shading()`**, la fonction **`auto.shading()`** n'admet pas de valeur manquante.

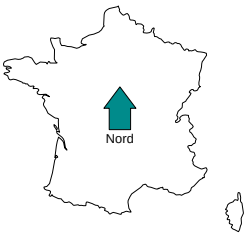
```
pal_col <- colorRampPalette(c("red2", "gold", "green4"))(6)
shades_br1 <- shading(breaks = quantile(EU_ctr2_LAEA@data$birth_rate,
                                        seq(0.1, 0.9, by = 0.2), include.lowest = TRUE),
                      cols = pal_col)
shades_br1
## $breaks
##       10%       30%       50%       70%       90%
##  9.851025 10.171411 10.852776 11.518014 12.770702
##
## $cols
## [1] "#EE0000" "#F45600" "#FBAC00" "#CBC700" "#65A900" "#008B00"
##
## attr(,"class")
## [1] "shading"
shades_br2 <- auto.shading(EU_ctr2_LAEA@data$birth_rate,
                           cutter = quantileCuts, n = 6,
                           cols = pal_col)
shades_br2
## $breaks
## 16.66667% 33.33333%       50% 83.33333%
##       9.9      10.0      11.0      12.0
##
## $cols
## [1] "#EE0000" "#F45600" "#FBAC00" "#CBC700" "#65A900" "#008B00"
##
## attr(,"class")
## [1] "shading"

plot(EU_ctr2_LAEA)
choro.legend(px = 3024544, py = 6402818, sh = shades_br1, bg = "white",
             under = "moins de", over = "plus de", between = "à",
```

```
            fmt = "%.1f")
```



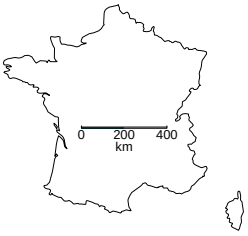
Orientation. On peut orienter la carte en dessinant une flèche du Nord avec la fonction **`north.arrow()`**. Les arguments **`x`** et **`y`** permettent de définir la position du symbole, **`len`** sa longueur, et **`lab`** le texte associé.

```
plot(FR_ctr_L2E)
north.arrow(x = 620000, y = 2100000, len = 50000, lab = "Nord",
            cex.lab = 1, tcol = "black", col = "cyan4")
```



Échelle. On peut dessiner une échelle grâce à la fonction **`map.scale()`**. Les arguments **`x`** et **`y`** permettent de définir la position du symbole, **`len`** sa longueur, **`ndivs`** le nombre de divisions de la barre et **`subdiv`** les valeurs intermédiaires.

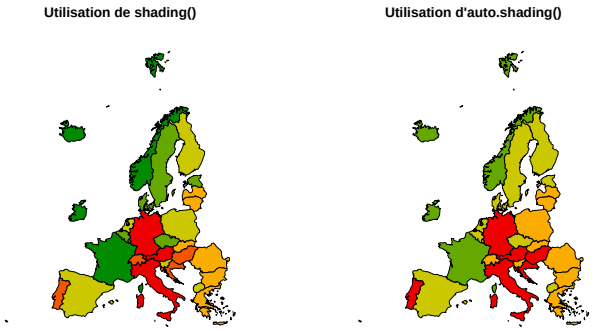
```
plot(FR_ctr_L2E)
map.scale(x = 640000, y = 2100000, len = 400000, units = "km",
          ndivs = 2, subdiv = 200, tcol = "black", sfcol = "cyan4")
```



6.1.7.2 Carte choroplèthe

On peut dessiner une carte choroplèthe avec la fonction **`choropleth()`**. Cette fonction prend un objet de classe **`spatialPolygonsDataFrame`** en argument principal (**`sp`**), le nom de la variable d'intérêt (**`v`**) et un objet de classe **`shading`** définissant la discrétisation de la variable d'intérêt. D'autres arguments, que l'on retrouve dans la fonction **`plot()`**, du package **`sp`** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a), sont aussi disponibles (§ 6.1.1.1).

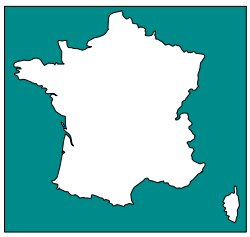
```
par(mfrow = c(1, 2))
choropleth(EU_ctr2_LAEA, v = EU_ctr2_LAEA@data$birth_rate, shading = shades_br1,
           main = "Utilisation de shading()")
choropleth(EU_ctr2_LAEA, v = EU_ctr2_LAEA@data$birth_rate, shading = shades_br2,
           main = "Utilisation d'auto.shading()")
```



6.1.7.3 Dessiner un polygone de masque

On peut définir un polygone de masque à l'aide de la fonction **poly.outer()**, à laquelle on fournit une couche (vectorielle ou matricielle) définissant l'emprise à découper (**exo.object**) par le contour d'un polygone (**input.poly**), en tenant éventuellement compte d'une zone "tampon" (**extend**), élargissant ainsi l'emprise du masque.

```
FR_MNT20000_L2E_mask <- poly.outer(exo.object = FR_MNT20000_L2E, input.poly = FR_ctr_L2E, extend = 1000)
plot(FR_MNT20000_L2E_mask, col = "cyan4")
```



6.1.8 Package getcartr

Le package **getcartr** (Brunsdon & Charlton, 2014) est spécialisé dans la production de cartogrammes, autrement appelés cartes en anamorphoses. Dans ce type de représentation, la géométrie de l'espace (le plus souvent la distance ou la surface) est déformée en fonction des valeurs d'une variable d'intérêt.

L'installation du package est un peu plus complexe que d'ordinaire, et se déroule en 3 temps.

Étape 1. Tout d'abord, il faut installer une bibliothèque C nommée FFTW3 [2]. Sous Windows [3], il faut créer une variable d'environnement nommée *FFTW3_DIR* pointant vers le répertoire contenant les fichiers *fftw3.h* et *libfftw3-3.dll* (attention, le chemin doit comporter des slashs comme dans R et non des antislashs comme c'est le cas normalement sous Windows). Ce répertoire doit, par ailleurs, être inscrit dans le *Path* de la machine. Enfin, la bibliothèque *libwinpthread-1.dll* doit être présente sur la machine. Sous Linux, il n'y a pas de difficulté particulière.

Étape 2. Le package **getcartr** dépend du package **Rcartogram** (Lang, 2011), qui fait l'interface avec la bibliothèque FFTW3. **Rcartogram** est disponible uniquement sous GitHub [4]. On peut l'installer en utilisant la fonction **install_github()** du package **devtools**.

```
devtools::install_github("omegahat/Rcartogram")
```

2. Site web de la bibliothèque FFTW : http://www.fftw.org/
3. Notes d'installation de la bibliothèque FFTW3 sous Windows : http://www.fftw.org/install/windows.html
4. Sources du package **Rcartogram** : https://github.com/omegahat/Rcartogram/

Étape 3. On peut enfin installer **getcartr**, depuis sont dépôt GitHub[5].

```
devtools::install_github("chrisbrunsdon/getcartr", subdir = "getcartr")
```

On peut maintenant charger le package **getcartr**.

```
library(getcartr)
```

6.1.8.1 Dessiner un cartogramme

La fonction **quick.carto()** permet de réaliser très facilement un cartogramme à partir d'un objet de classe **SpatialPolygonsDataFrame** (**spdf**), en fonction d'une variable d'intérêt (**v**). Attention, les données manquantes ne sont pas autorisées au sein de cette dernière. Notez que plusieurs arguments permettent de paramétrer l'algorithme calculant l'anamorphose :

- **res** : la résolution de la grille de distorsion ;
- **thresh** : la plus basse valeur de densité autorisée ;
- **blur** : le degré de flou gaussien à appliquer à la grille de densité ;
- **prec** : la précision lors de l'exécution d'interpolation ;
- **gapdens** : la densité à attribuer aux intervalles entre les polygones.

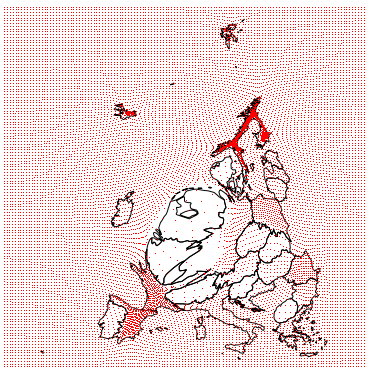
```
EU_ctr2_LAEA_ctg <- quick.carto(spdf = EU_ctr2_LAEA, v = EU_ctr2_LAEA@data$pop_dens)
EU_ctr2_LAEA_ctg
## class       : SpatialPolygonsDataFrame
## features    : 31
## extent      : 12.93515, 114.8144, 2.713177, 123.7076  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## variables   : 15
## names       : id, iso_a2, iso_a3,        name, continent,           part,   area,      pop,    pop_dens,
## min values  :  1,     AT,    AUT,     Austria,    Europe, Eastern Europe, 100250, 10045401,  100.942761, 1.18020
## max values  :  9,     SK,    SWE, Switzerland,    Europe, Western Europe,  91590,  9182927,   99.814747, 9.80370
```

L'objet renvoyé étant de classe **SpatialPolygonsDataFrame**, on peut le dessiner *via* un simple appel à la fonction **plot()**.

```
plot(EU_ctr2_LAEA_ctg, main = "Anamorphose en fonction\nde la densité de population")
```

On peut apprécier le degré de déformation de la carte due à l'anamorphose en dessinant un semi de points avec la fonction **dispersion()**.

```
plot(EU_ctr2_LAEA_ctg)
dispersion(EU_ctr2_LAEA_ctg, add = TRUE)
```



5. Sources du package **getcartr** : https://github.com/chrisbrunsdon/getcartr/

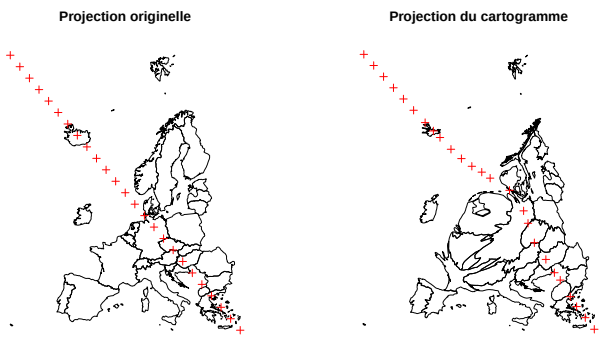
6.1.8.2 Superposer des couches en fonction d'une distorsion

On peut également déformer des couches de points, de lignes ou de polygones à partir d'un cartogramme déjà réalisé, et ce grâce aux fonctions **warp.points()**, **warp.lines()** ou **warp.polys()**. Ces fonctions prennent un objet **sp** en premier argument et, en second argument, l'objet **SpatialPolygonsDataFrame** renvoyé par la fonction **quick.carto()**.

```
x1 <- seq(EU_ctr2_LAEA@bbox["x", "max"], EU_ctr2_LAEA@bbox["x", "min"], length.out = 25)
y1 <- seq(EU_ctr2_LAEA@bbox["y", "min"], EU_ctr2_LAEA@bbox["y", "max"], length.out = 25)
pbox <- SpatialPoints(coords = cbind(x1, y1))
pbox_ctg <- warp.points(pts = pbox, warper = EU_ctr2_LAEA_ctg)
str(pbox_ctg)
```

```
## Formal class 'SpatialPoints' [package "sp"] with 3 slots
##   ..@ coords     : num [1:25, 1:2] 114.7 110.7 106.9 104.1 99.7 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "newx" "newy"
##   ..@ bbox       : num [1:2, 1:2] 13.3 3.13 114.71 124.09
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "newx" "newy"
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr NA
```

```
par(mfrow = c(1, 2))
plot(EU_ctr2_LAEA, main = "Projection originelle")
plot(pbox, col = "red", add = TRUE)
plot(EU_ctr2_LAEA_ctg, main = "Projection du cartogramme")
plot(pbox_ctg, col = "red", add = TRUE)
```



Il est également possible d'utiliser la fonction **quick.carto.transform()**, qui renvoie, à partir d'un objet **patialPolygonsDataFrame** issu de la fonction **quick.carto()**, une fonction permettant de distordre un autre objet de classe **sp**, quel qu'il soit (points, polylignes ou polygones) selon une anamorphose définie.

```
FUN_ctg <- quick.carto.transform(poly = EU_ctr2_LAEA_ctg)
plot(EU_ctr2_LAEA_ctg, main = "Centroïdes des polygones exprimés\ndans la projection du cartogramme")
plot(FUN_ctg(EU_labpt_LAEA), col = "red", add = TRUE)
```

6.1.9 Package cartogram

Le package **cartogram** (Jeworutzki, 2016) est lui aussi spécialisé dans la production de cartogrammes. Il est cependant moins performant que le package **getcartr** (Brunsdon & Charlton, 2014), car il ne permet pas de paramétrer l'algorithme de distorsion (il ne comporte qu'une seule fonction qui réalise le cartogramme), et parce qu'il est beaucoup moins rapide (car entièrement programmé en R).

```
library(cartogram)
```

6.1.9.1 Dessiner un cartogramme

La fonction **cartogram()** permet de réaliser un cartogramme à partir d'un objet de classe **SpatialPolygonsDataFrame** (**shp**), en fonction d'une variable d'intérêt (**weight**). Attention, cette dernière ne doit pas comporter de données manquantes. On peut gérer ici le nombre d'itérations à effectuer (**itermax**).

```
EU_ctr2_LAEA_ctg <- cartogram(shp = EU_ctr2_LAEA, weight = "pop_dens",
                              itermax = 15, maxSizeError = 1.0001)
EU_ctr2_LAEA_ctg
## class       : SpatialPolygonsDataFrame
## features    : 31
## extent      : 1849145, 5946459, 1206340, 6211403  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +units=m +no_defs
## variables   : 15
## names       : id, iso_a2, iso_a3,        name, continent,           part,   area,      pop,    pop_dens,
## min values  :  1,     AT,    AUT,     Austria,    Europe, Eastern Europe, 100250, 10045401,  100.942761, 1.1802
## max values  :  9,     SK,    SWE, Switzerland,    Europe, Western Europe,  91590,  9182927,   99.814747, 9.8037
```

La fonction renvoie un objet de classe **SpatialPolygonsDataFrame** ; on peut le dessiner par un simple appel à la fonction **plot()**.

```
plot(EU_ctr2_LAEA_ctg, main = "Anamorphose en fonction\nde la densité de population")
```

6.1.10 Package threejs

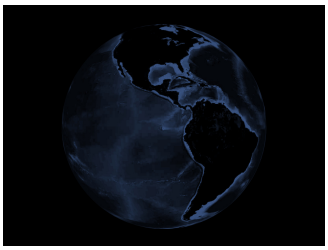
Le package **threejs** (Lewis, 2016) permet de créer quelques types de graphiques interactifs 3D, à savoir des nuages de points, des réseaux, et des globes (terrestres ou extraterrestres). Il utilise la bibliothèque JavaScript Three.js [6].

```
library(threejs)
```

6.1.10.1 Dessiner un globe interactif

Pour dessiner un globe terrestre, il suffit d'utiliser la fonction **globejs()**.

```
globejs()
```



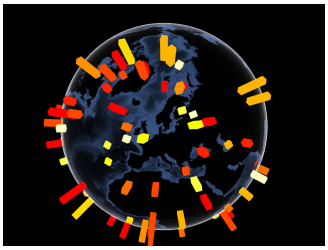
Les arguments **rotationlat** et **rotationlong** permettent de choisir le point de vue initial de la scène (latitude et longitude exprimées en radian). On peut afficher une atmosphère, sous la forme d'un halo blanc, avec l'argument **atmosphere = TRUE**. On peut choisir la couleur de fond avec l'argument **bg** (**"black"**, par défaut). Pour personnaliser la taille de l'image dans la fenêtre graphique, il faut jouer avec les arguments **height** et **width** (par défaut, la taille s'adapte automatiquement) et **fov** pour le niveau de zoom (**35**, par défaut ; plus la valeur est petite plus on est proche).

```
globejs(rotationlat = 1.0, rotationlong = 4.5, bg = "navy", atmosphere = TRUE, fov = 20)
```

6. Site web de la bibliothèque JavaScript Three.js : https://threejs.org/.

On peut ajouter des barres sur le globe, si l'on en spécifie les coordonnées dans les arguments **lat** et **long**. Il est possible définir les longueurs (**value**), la largeur (**pointsize**, une seule valeur possible) et les couleurs de ses barres (**col**). Notez qu'il faut au minimum deux couples de coordonnées pour que le tracé soit effectué.

```
globejs(lat = sample(-90:90, 100), long = sample(-90:90, 100),
        value = sample(0:100, 100), pointsize = 10,
        col = heat.colors(10)[sample(1:10, 100, replace = TRUE)],
        rotationlat = 1.0, rotationlong = 4.5, atmosphere = TRUE)
```



Il est possible de dessiner des trajectoires. Pour ce faire, il faut passer un objet de classe **matrix** ou **data.frame** à l'argument **arcs**, et le définir comme suit :

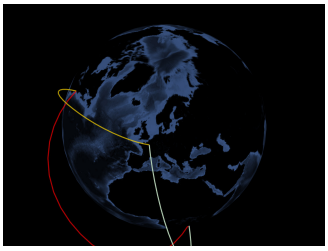
- 1re colonne : la latitude d'origine ;
- 2e colonne : la longitude d'origine ;
- 3e colonne : la latitude de destination ;
- 4e colonne : la longitude de destination.

```
xy_PA <- c(+ 2.35, +48.85)
xy_NY <- c(-73.78, +40.63)
xy_JB <- c(+28.04, -26.20)
globe_traj <- cbind(rbind(xy_PA, xy_PA, xy_NY, xy_JB),
                    rbind(xy_NY, xy_JB, xy_JB, xy_PA))
colnames(globe_traj) <- c("lon_orig", "lat_orig", "lon_des", "lat_dest")
rownames(globe_traj) <- NULL
head(globe_traj)
##      lon_orig lat_orig lon_des lat_dest
## [1,]     2.35    48.85  -73.78    40.63
## [2,]     2.35    48.85   28.04   -26.20
## [3,]   -73.78    40.63   28.04   -26.20
## [4,]    28.04   -26.20    2.35    48.85
```

Plusieurs arguments permettent de gérer le représentation des trajectoires :

- **arcsColor** : les noms de couleurs HTML[7] des traits (1 ou plusieurs valeurs) ;
- **arcsOpacity** : l'indice d'opacité des couleurs (1 seule valeur entre 0 et 1) ;
- **arcsHeight** : la hauteur de la trajectoire au dessus du globe (1 seule valeur entre 0 et 1) ;
- **arcsLwd** : l'empattement des traits (1 seule valeur entre 0 et 1).

```
globe_traj <- globe_traj[, c("lat_orig", "lon_orig", "lat_dest", "lon_des")]
globejs(arcs = globe_traj,
        arcsColor = c("gold", "green", "red"), arcsOpacity = 0.7,
        arcsHeight = 0.5, arcsLwd = 4,
        rotationlat = 1.0, rotationlong = 4.5)
```



7. Couleur du Web : https://fr.wikipedia.org/wiki/Couleur_du_Web.

Le package **threejs** ne permet pas d'afficher des couches vectorielles à proprement parler. Il est cependant possible de passer par l'intermédiaire d'un fichier image pour représenter de type de données. Pour ce faire, il faut tout d'abord créer une image (aux format JPEG ou PNG, par exemple) à partir d'une carte classique. Pour que la représentation cartographique soit correcte, l'image ne doit pas comporter de marge et les proportions du fichier produites doivent respecter celle de l'image dessinée. Par ailleurs, il faut veiller à gérer les limites des axes. Il faut, bien évidemment, prendre soin de ne pas dessiner les axes, les étiquettes de ces derniers ou encore un cadre autour de la carte.

```
globe_file <- tempfile(fileext = ".png")
```

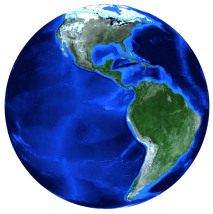
```
png(globe_file, width = 1000, height = 500, bg = "grey20", antialias = "default")
  par(mar = c(0, 0, 0, 0), pin = c(4, 2), pty = "m", xpd = FALSE,
      xaxs = "i",  yaxs = "i", xaxt = "n", yaxt = "n", bty = "n")
  plot(WD_ctr_W84_po, col = "black", bg = "grey20", border = "grey20", ann = FALSE,
       xlim = c(-180, 180), ylim = c(-90, 90))
  plot(EU_ctr1_W84, col = "cyan4", border = "grey20", add = TRUE)
dev.off()
```

L'image ainsi générée peut être alors fournie à l'argument **img** de la fonction **globejs()**. Notez que de cette manière, il est possible de réaliser toutes les représentations graphiques classiques (cartes typologiques, choroplèthes ou en symboles proportionnels).

```
globejs(img = globe_file, rotationlat = 1.0, rotationlong = 4.5)
```

Ainsi, il est tout à fait possible d'avoir une représentation d'un objet céleste. Pour cela on passe une image satellite de cete objet à l'argument **img**. Dans notre exemple, on souhaite disposer d'une représentation à partir d'une image satelleite de la Terre.

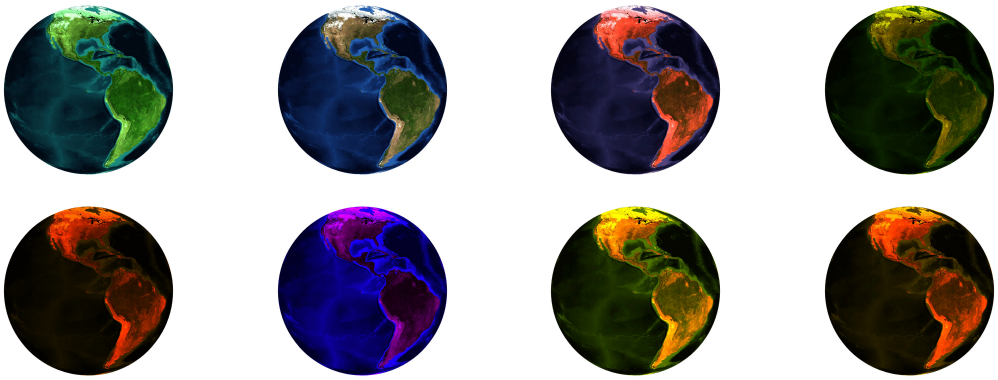
```
earth <- "http://eoimages.gsfc.nasa.gov/images/imagerecords/73000/73909/world.topo.bathy.200412.3x5400x2700.jpg"
globejs(img = earth, bg = "white")
```



Plusieurs arguments permettent de gérer les couleurs :

- **bodycolor** : la couleur de réflectivité ;
- **emissive** : la couleur d'émissivité ;
- **lightcolor** : la couleur d'ambiance de la scène.

```
globejs(img = earth, bg = "white", emissive = "white", lightcolor = "green")
globejs(img = earth, bg = "white", emissive = "white", lightcolor = "#333333")
globejs(img = earth, bg = "white", emissive = "white", lightcolor = "red")
globejs(img = earth, bg = "white", emissive = "red"  , lightcolor = "red")
globejs(img = earth, bg = "white", emissive = "red"  , lightcolor = "green")
globejs(img = earth, bg = "white", emissive = "green", lightcolor = "red")
globejs(img = earth, bg = "white", emissive = "red"  , lightcolor = "blue")
globejs(img = earth, bg = "white", emissive = "red"  , lightcolor = "gold")
globejs(img = earth, bg = "white", emissive = "gold" , lightcolor = "red")
```



6.2 Syntaxe de type *Trellis*

La syntaxe de type *Trellis* repose sur les travaux que William Cleveland a menés aux Bell Labs avec Richard Becker [8], et qu'il a exposés en 1993 dans son livre intitulé *Visualizing Data*. Cette syntaxe a été popularisée sous R par le package **lattice** (Sarkar, 2008), inspiré de la suite Trellis originellement développée pour S et S-PLUS aux Bell Labs. Elle repose sur le *Grid graphics engine* et dépend du package **grid**, développé par Paul Murrel, et qui est une partie intégrante de R (R Core Team, 2016). Une des conséquences de ce fait est que cette syntaxe n'est pas compatible avec celle des outils graphiques habituels de R. Les paramètres définis par la fonction **par()** n'ont, par exemple, aucun effet sur les fonctions du package **lattice**, car ces dernières disposent de leurs propres jeux de paramètres graphiques. Une autre conséquence est que l'on peut stocker les graphiques comme n'importe quel objet R, et que l'on peut les réutiliser par la suite.

Les fonctions cartographiques, que nous allons présenter dans cette partie, ne sont donc pas compatibles avec les fonctions graphiques conventionnelles du package **graphics**, ni avec aucune autre fonction utilisant les méthodes défnies par la classe **sp** (§ 6.1).

6.2.1 Package sp

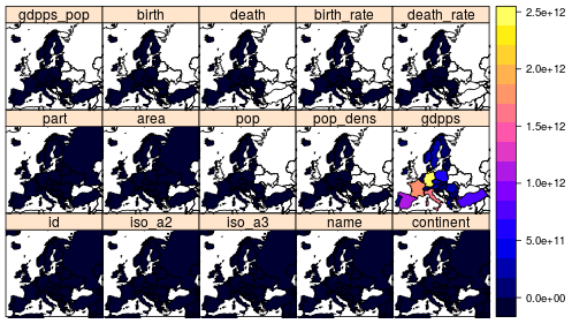
Outre les sorties cartographiques conventionnelles de type *Painter's Model*, le package **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a) propose des fonctions utilisant une syntaxe de type *Trellis*.

```
library(sp)
```

6.2.1.1 Affichage

C'est la fonction **spplot()** qui permet de tracer des cartes de type *Trellis* avec le package **sp**. En argument principal, elle prend un objet de classe **sp** (vectoriel ou matriciel). L'argument **col.regions = NA** permet de ne pas dessiner le contenu de l'entité considérée).

```
spplot(EU_ctr1_LAEA)
```

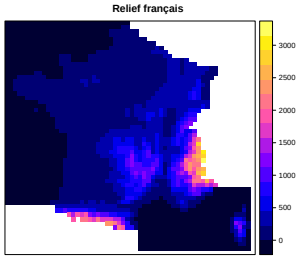


Comme on peut le constater, par défaut, l'ensemble des variables de la table attributaire est représentée en treillis avec une échelle commune. Dans cet exemple, toutes les variables n'étant pas exprimées dans une seule et même unité, la sortie produite n'a pas de sens.

8. Le site web du projet Trellis : http://ect.bell-labs.com/sl/project/trellis/display.writing.html/.

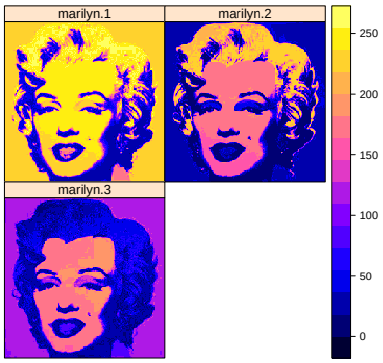
La fonction **spplot()** permet de réaliser des cartes matricielles à partir d'objets de classes **sp** ou **raster**.

```
spplot(FR_MNT20000_L2E, main = "Relief français")
```



Elle gère les rasters multicouches de classes **RasterBrick** ou **RasterStack** ; elle trace automatiquement les divers couches matricielles et leur associe une échelle commune.

```
class(marylin)
## [1] "RasterBrick"
## attr(,"package")
## [1] "raster"
spplot(marylin)
```



6.2.1.2 Assigner une carte dans un objet

Comme signalé plus haut, on peut stocker les graphiques dans des objet R, ce qui permet de pouvoir les réutiliser par la suite. Ces objet sont de classe **trellis**.

```
map_sp <- spplot(EU_ctr1_LAEA)
class(map_sp)
## [1] "trellis"
names(map_sp)
##  [1] "formula"           "as.table"          "aspect.fill"       "legend"
##  [5] "panel"             "page"              "layout"            "skip"
##  [9] "strip"             "strip.left"        "xscale.components" "yscale.components"
## [13] "axis"              "xlab"              "ylab"              "xlab.default"
## [17] "ylab.default"      "xlab.top"          "ylab.right"        "main"
## [21] "sub"               "x.between"         "y.between"         "par.settings"
## [25] "plot.args"         "lattice.options"   "par.strip.text"    "index.cond"
## [29] "perm.cond"         "condlevels"        "call"              "x.scales"
## [33] "y.scales"          "panel.args.common" "panel.args"        "packet.sizes"
## [37] "x.limits"          "y.limits"          "x.used.at"         "y.used.at"
## [41] "x.num.limit"       "y.num.limit"       "aspect.ratio"      "prepanel.default"
## [45] "prepanel"
```

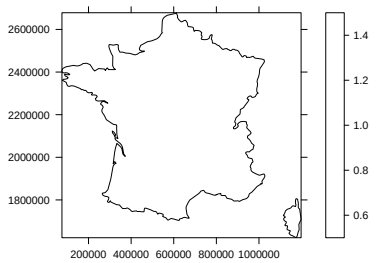
On peut modifier bon nombre de paramètres *a posteriori* à l'aide de la fonction **update.trellis()** du package **lattice** (Sarkar, 2008).

6.2.1.3 Éléments contextuels

Titre. On peut afficher un titre en utilisant l'argument **main** de la fonction **spplot()**.

Axes. On peut afficher les axes à l'aide de l'argument **scales**.

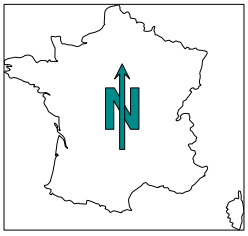
```
spplot(FR_ctr_L2E, col.regions = NA, scales = list(draw = TRUE))
```



Légende. Lors de l'utilisation de la fonction **`spplot()`**, il est possible de masquer la légende en utilisant l'argument **`colorkey = FALSE`**.

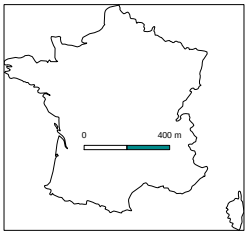
Orientation. Pour tracer la flèche du Nord, il faut utiliser la fonction **`layout.north.arrow()`** en argument principal de la fonction **`SpatialPolygonsRescale()`**. Les arguments sont les mêmes que ceux déjà décrits pour la syntaxe de type *Painter's Model* (§ 6.1.1.2). Notez que l'argument **`plot.grid`**, qui gère la compatibilité avec les différentes syntaxes, doit être défini à la valeur **`FALSE`** (valeur par défaut) pour que la fonction soit compatible avec la syntaxe de type *Trellis*).

```
lna <- list("SpatialPolygonsRescale", layout.north.arrow(), offset = c(550000, 2000000),
            scale = 4e5, fill = "cyan4")
spplot(FR_ctr_L2E, sp.layout = list(lna), col.regions = NA, colorkey = FALSE)
```



Échelle. Pour tracer l'échelle, il faut utiliser la fonction **`layout.scale.bar()`** en argument principal de la fonction **`SpatialPolygonsRescale()`**. Les arguments sont les mêmes que ceux déjà décrits pour la syntaxe de type *Painter's Model* (§ 6.1.1.2). Ici aussi, l'argument **`plot.grid`** gère la compatibilité avec les différentes syntaxes graphiques. Le texte, quant à lui, est ajouté grâce à la fonction **`sp.text()`**.

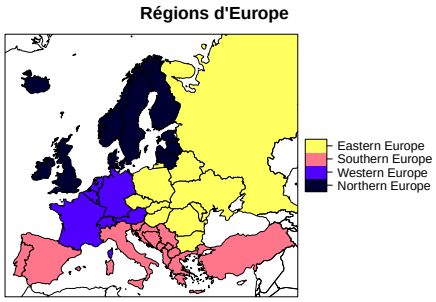
```
lsb <- list("SpatialPolygonsRescale", layout.scale.bar(), offset = c(450000, 2000000),
            scale = 4e5, fill = c("transparent", "cyan4"))
lt1 <- list("sp.text", c(450000, 2070000), "0")
lt2 <- list("sp.text", c(850000, 2070000), "400 m")
spplot(FR_ctr_L2E, sp.layout = list(lsb, lt1, lt2), col.regions = NA, colorkey = FALSE)
```



6.2.1.4 Carte typologique

La fonction **`spplot()`** permet de dessiner des cartes typologiques sur une variable qualitative. Pour cela, il suffit simplement de donner un objet de la classe **`SpatialPolygonsDataFrame`** en premier argument, et le nom d'une colonne d'intérêt de la table attributaire à l'argument **`zcol`** (la variable doit être un facteur et non un vecteur de chaînes de caractères). La présence de données manquantes dans la variable d'intérêt est gérée automatiquement par la fonction. On peut fournir plusieurs facteurs à l'argument **`zcol`**, mais ces derniers devront présenter les mêmes modalités possibles, sans quoi une erreur sera renvoyée et la carte ne sera pas dessinée.

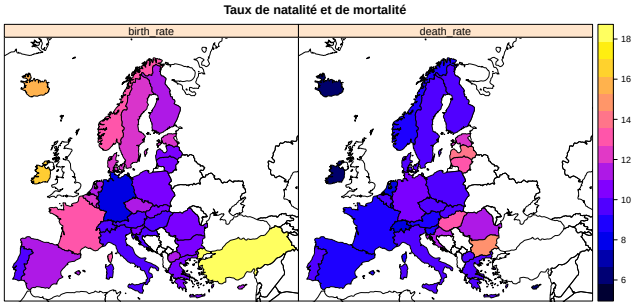
```
map_sp_typ <- spplot(EU_ctr1_LAEA, zcol = "part", main = "Régions d'Europe")
map_sp_typ
```



6.2.1.5 Carte choroplèthe

Pour une carte choroplèthe sur variable quantitative, c'est le même procédé, **zcol** correspondant à la colonne de la table attributaire contenant la variable numérique que l'on souhaite représenter. Par ailleurs, si **zcol** contient plusieurs noms de variables, la fonction renvoie automatiquement un graphe en treillis et associe une échelle commune aux différentes variables. Pour que la représentation ait un sens, ces dernières doivent donc être exprimées dans la même unité.

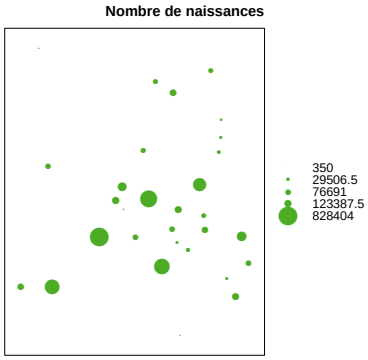
```
map_sp_cho <- spplot(EU_ctr1_LAEA, zcol = c("birth_rate", "death_rate"),
                     main = "Taux de natalité et de mortalité")
map_sp_cho
```



6.2.1.6 Carte en symboles proportionnels

On peut également réaliser des cartes en symboles proportionnels, de types cercles ou disques, à l'aide de la fonction **bubble()**. Ici l'argument **zcol**, a qui l'on passe le nom de la variable d'intérêt, n'admet qu'une seule valeur

```
map_sp_bub <- bubble(EU_labpt_LAEA, zcol = "birth", fill = TRUE, main = "Nombre de naissances")
map_sp_bub
```



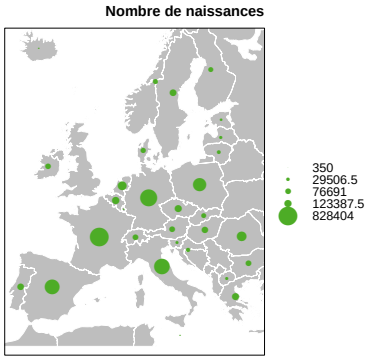
6.2.1.7 Faceting

Il n'y a pas de *faceting* autre que celui automatiquement réalisé par la fonction **spplot()** lorsque plusieurs variables sont considérées. En revanche, le package **maptools** (Bivand & Lewin-Koh, 2015), propose, quant à lui, une fonction permettant de faire du *faceting* en fonction d'une variable conditionnelle (§ 6.2.4.1).

6.2.1.8 Superposer des couches

Pour superposer des couches, il faut utiliser la syntaxe **lattice** habituelle.

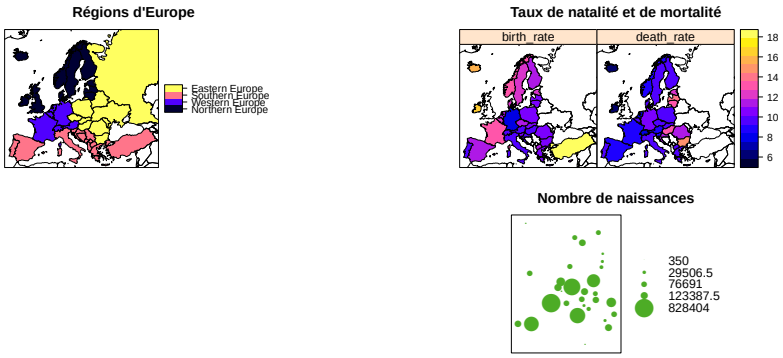
```
bubble(EU_labpt_LAEA, zcol = "birth", main = "Nombre de naissances",
       panel = function(...) {
         sp.polygons(EU_ctr1_LAEA, fill = "grey", col = "white")
         sp:::panel.bubble(...)
       })
```



6.2.1.9 Assembler des cartes

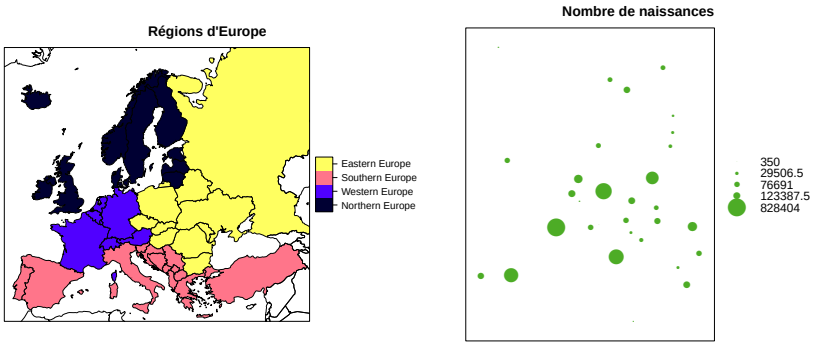
Solution 1 Une première solution consiste à faire appel à la fonction **print()** (ou **plot()**, les deux fonctions présentent le même comportement). L'argument **split** sert à définir l'emplacement de la carte dans la fenêtre graphique, et l'argument **more** permet de définir si une carte est encore attendue ou non.

```
print(map_sp_typ, split = c(1, 1, 2, 2), more = TRUE)
print(map_sp_cho, split = c(2, 1, 2, 2), more = TRUE)
print(map_sp_bub, split = c(2, 2, 2, 2), more = FALSE)
```



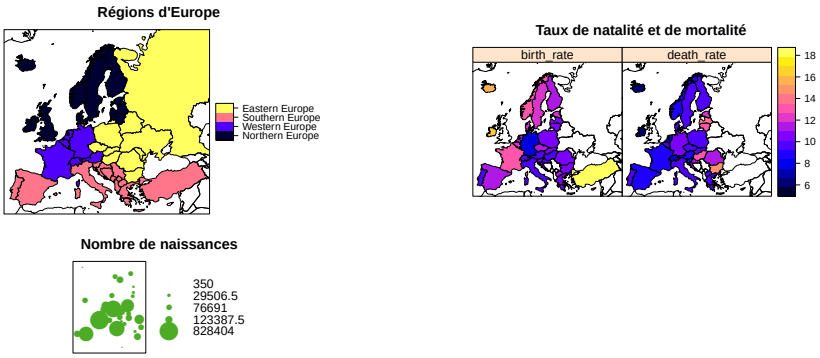
Solution 2 Comme c'est le cas habituellement, les objets de classe **trellis** peuvent êtres assemblés à l'aide de la fonction **grid.arrange()** du package **gridExtra** (Auguie, 2016). Pour cela, il suffit de fournir à la fonction les différents objets **trellis** correspondant aux cartes, et de définir le nombre de lignes (**nrow**) et/ou de colonnes (**ncol**) souhaitées.

```
gridExtra::grid.arrange(map_sp_typ, map_sp_bub, ncol = 2)
```



Pour pouvoir gérer plus finement les paramètres graphiques, on peut passer par l'intermédiaire de la fonction **arrangeGrob()**, toujours proposée par **gridExtra**. Elle requiert une liste d'objets de classe **trellis** dans l'argument **grobs**, et l'on peut, par exemple, gérer les tailles relatives des lignes et des colonnes avec les arguments **heights** et **widths**.

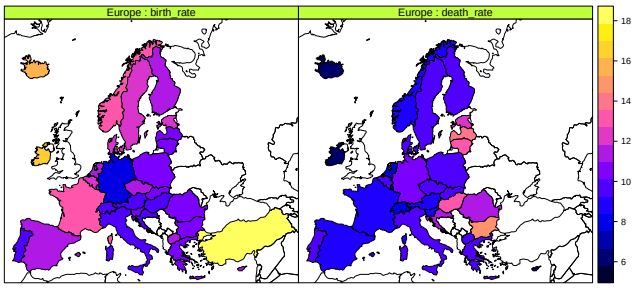
```
grob <- gridExtra::arrangeGrob(grobs = list(map_sp_typ, map_sp_cho, map_sp_bub),
                               heights = c(2, 1.2), widths = c(2, 1.2),
                               nrow = 2, ncol = 2)
class(grob)
## [1] "gtable" "gTree"  "grob"   "gDesc"
gridExtra::grid.arrange(grob)
```



6.2.1.10 Thèmes graphiques

Comme à l'accoutumé sous **lattice** (Sarkar, 2008), on peut ici personnaliser la sortie graphique avec son propre style grâce à la fonction **strip.custom()**. On peut également utiliser des styles prédéfinis grâce à la fonction **strip.default()**.

```
strip_sp <- lattice::strip.custom(bg = "olivedrab1", fg = "white",
                                  strip.names = TRUE, var.name = "Europe")
spplot(EU_ctr1_LAEA, zcol = c("birth_rate", "death_rate"), strip = strip_sp)
```



Pour davantage de solutions de paramétrage de thèmes, on peut utiliser le package **latticeExtra** (Sarkar & Andrews, 2016), et notamment la fonction **custom.theme()**.

6.2.2 Package raster

Le package **raster** (Hijmans, 2015) définit des classes et méthodes de données matricielles. Il propose également des fonctionnalités pour tracer des cartes à partir des objets de classe **raster**, en utilisant la syntaxe de type *Painter's Model*, comme nous l'avons déjà vu (§ 6.1.2), mais aussi de type *Trellis*, comme nous allons le voir ici.

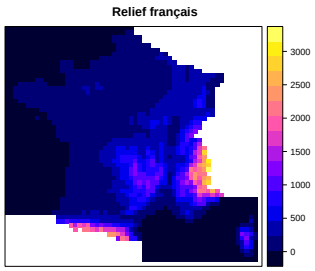
```
library(raster)
```

6.2.2.1 Affichage

Tout comme le package **sp**, le package **raster** propose lui aussi une fonction **spplot()**, mais elle permet quant à elle, de dessiner des cartes à partir d'objets de classe **raster**.

Cette fonction permet de dessiner des cartes à partir de rasters simples de classe **RasterLayer**.

```
class(FR_MNT20000_L2E)
## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"
spplot(FR_MNT20000_L2E, main = "Relief français")
```



6.2.2.2 Assigner une carte dans un objet

Comme la fonction **spplot()** renvoie un objet de classe **trellis**, on peut l'assigner dans un objet, comme nous l'avons vu précédemment (§ 6.2.1.2).

6.2.2.3 Éléments contextuels

On peut utiliser les fonctionnalités proposées par le package **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a) (§ 6.2.1.3).

6.2.3 Package rasterVis

(§ **??**)

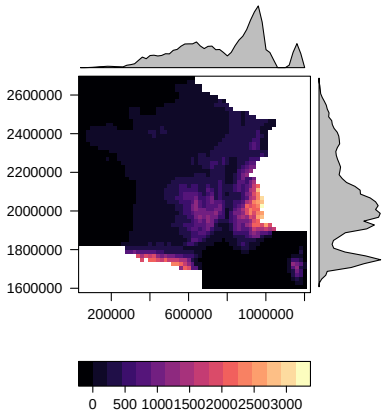
Le package **rasterVis** (Perpiñán & Hijmans, 2014) [9] est spécialisé dans l'amélioration des méthodes de visualisation et d'interaction avec les données matricielles. Il met en œuvre des méthodes de visualisation pour des données quantitatives et catégorielles, que ce soit pour des rasters simples ou multiples. Par ailleurs, il fournit des outils pour afficher des rasters spatio-temporels et des champs de vecteurs.

```
library(rasterVis)
```

6.2.3.1 Affichage

Pour dessiner des rasters, le package **rasterVis** propose la fonction **levelplot()** comme une alternative aux fonctions **plot()** et **image()** du package **raster**. Ce package gère les objets matriciels de classe **sp**, mais pas ceux de classe **raster**.

```
levelplot(FR_MNT20000_L2E)
```



Notez que l'on peut masquer les distributions marginales avec l'argument **margin = FALSE**.

9. Site web du package **rasterVis** : https://oscarperpinan.github.io/rastervis/.

6.2.3.2 Assigner une carte dans un objet

Le package **rasterVis** dépendant du package **lattice** (Sarkar, 2008), on peut, comme pour ce dernier, assigner une carte dans un objet R afin de la réutiliser. L'objet renvoyé est de classe **trellis**.

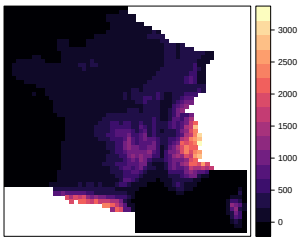
```
map_rv_lp <- levelplot(FR_MNT20000_L2E)
class(map_rv_lp)
## [1] "trellis"
names(map_rv_lp)
##  [1] "formula"           "as.table"          "aspect.fill"       "legend"
##  [5] "panel"             "page"              "layout"            "skip"
##  [9] "strip"             "strip.left"        "xscale.components" "yscale.components"
## [13] "axis"              "xlab"              "ylab"              "xlab.default"
## [17] "ylab.default"      "xlab.top"          "ylab.right"        "main"
## [21] "sub"               "x.between"         "y.between"         "par.settings"
## [25] "plot.args"         "lattice.options"   "par.strip.text"    "index.cond"
## [29] "perm.cond"         "condlevels"        "call"              "x.scales"
## [33] "y.scales"          "panel.args.common" "panel.args"        "packet.sizes"
## [37] "x.limits"          "y.limits"          "x.used.at"         "y.used.at"
## [41] "x.num.limit"       "y.num.limit"       "aspect.ratio"      "prepanel.default"
## [45] "prepanel"
```

6.2.3.3 Éléments contextuels

Titre. On peut afficher un titre en utilisant l'argument **main** de la fonction **levelplot()**.

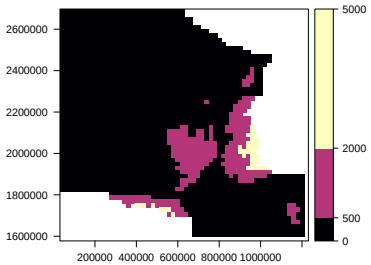
Axes. On peut afficher (comportement par défaut) ou masquer les axes à l'aide de l'argument **scales** (comme dans l'exemple ci-après). Pour masquer les étiquettes, il faut utiliser les arguments **xlab = NULL** et/ou **ylab = NULL**.

```
levelplot(FR_MNT20000_L2E, margin = FALSE, scales = list(draw = FALSE))
```



Légende. On peut masquer la légende en utilisant l'argument **colorkey = FALSE** lors de l'utilisation de la fonction **levelplot()**. Pour décider des valeurs de discrétisation, il faut associer l'utilisation de l'argument **at**, qui va gérer l'affichage du raster, avec celle de **colorkey**, qui va gérer l'affichage de la légende.

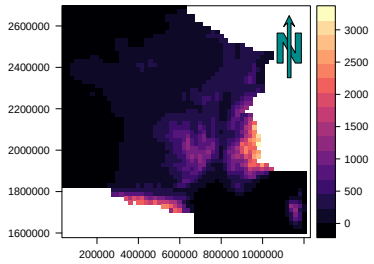
```
lvp_key <- list(at = c(0, 500, 2000, 5000),
                labels = list(at = c(0, 500, 2000, 5000)))
levelplot(FR_MNT20000_L2E, margin = FALSE, at = lvp_key$at, colorkey = lvp_key)
```



Orientation. Pour tracer la flèche du Nord, il faut faute utiliser les fonctions proposées par le package **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a), comme nous l'avions vu plus haut (§ 6.1.1.2), à savoir **layout.north.arrow()** et **SpatialPolygonsRescale()**. Pour pouvoir superposer le symbole au **levelplot()**, il faut utiliser la fonction **layer()** du package **latticeExtra** (Sarkar & Andrews, 2016).

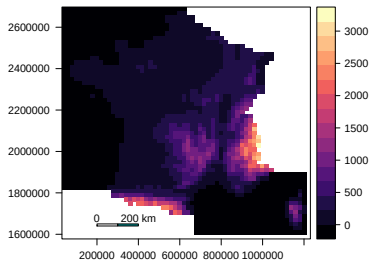
```
levelplot(FR_MNT20000_L2E, margin = FALSE) +
      latticeExtra::layer({
          SpatialPolygonsRescale(layout.north.arrow(), offset = c(1070000, 2350000),
```

```
                                scale = 3e5, fill = "cyan4")
      })
```



Échelle. Pour tracer l'échelle, il faut utiliser les fonctions **`layout.scale.bar()`** et **`SpatialPolygonsRescale()`** du package **`sp`** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a) (§ 6.1.1.2). L'affichage sur la carte est également réalisé avec la fonction **`layer()`** du package **`latticeExtra`** (Sarkar & Andrews, 2016). Pour ajouter le texte, on utilise la fonction **`grid.text()`** du package **`grid`** (R Core Team, 2016).

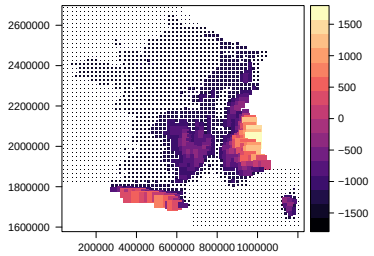
```
levelplot(FR_MNT20000_L2E, margin = FALSE) +
      latticeExtra::layer({
          SpatialPolygonsRescale(layout.scale.bar(), offset = c(200000, 1640000),
                                 scale = 2e5, fill = c("transparent", "cyan4"))
        grid.text(x = c(2e5, 3e5, 4e5), y = 1680000, label = c("0", "", "200 km"),
                  gp = gpar(cex = 0.8), default.units = "native")
      })
```



6.2.3.4 Dessiner des pixels de différentes tailles

La fonction **`levelplot()`** propose l'argument **`shrink`** qui permet de gérer la taille d'un pixel dessiné en fonction de sa valeur.

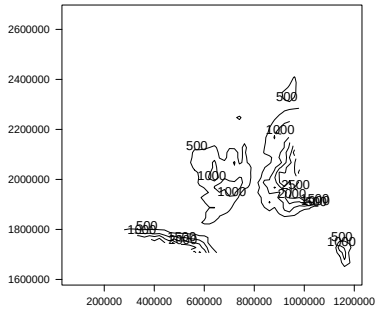
```
rCenter <- (maxValue(FR_MNT20000_L2E) + minValue(FR_MNT20000_L2E)) / 2
levelplot(FR_MNT20000_L2E - rCenter, margin = FALSE, shrink = c(0.2, 4.0))
```



6.2.3.5 Dessiner des isolignes

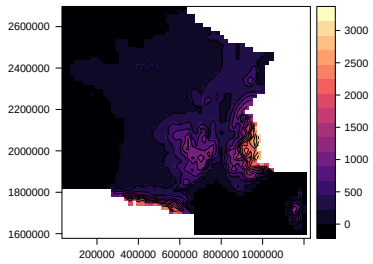
Il est possible de tracer des isolignes, très facilement, grâce à la fonction **`contourplot()`**. On définit les bornes des classes à l'aide de l'argument **`at`**, et la valeur des étiquettes avec **`labels`** (**`= NULL`** pour l'absence d'étiquette).

```
contourplot(FR_MNT20000_L2E, contour = TRUE, margin = FALSE)
```



On peut également utiliser la fonction **levelplot()** et l'argument **contour = TRUE**. Dans ce cas, par défaut, les pixels sont dessinés ; pour les masquer, on peut utiliser l'argument **region = FALSE**. De manière similaire à **contourplot()**, on définit les bornes des classes à l'aide de l'argument **at**, et de valeur des étiquettes avec **labels** (**= ""** répété autant de fois que de bornes pour l'absence d'étiquette).

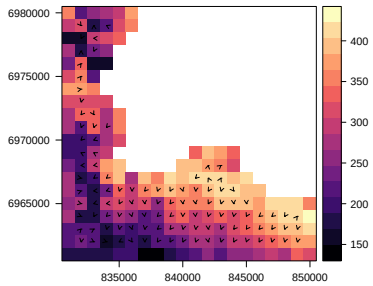
```
levelplot(FR_MNT20000_L2E, contour = TRUE, margin = FALSE)
```



6.2.3.6 Dessiner un champ de vecteurs

On peut tracer des vecteurs dans les pixels à l'aide de la fonction **vectorplot()**.

```
vectorplot(FR_MNT1000_L93_clip_EXT)
```

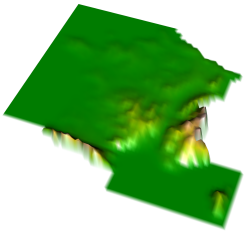


Pour un autre type de représentation, on peut également utiliser la fonction **streamplot()**.

6.2.3.7 Dessiner un raster en 3D

Pour réaliser des représentations 3D interactives, on peut utiliser la fonction **plot3D()**, qui requiert des fonctionnalités fournies par le package **rgl** (Adler *et al.*, 2014). L'argument **zfac** permet de gérer le degré d'exagération que l'on souhaite appliquer à la variable représentée en Z.

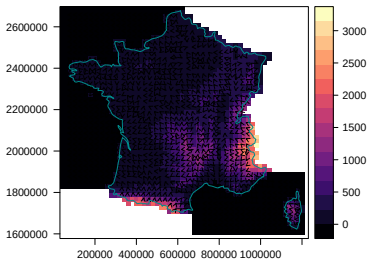
```
library(rgl)
plot3D(FR_MNT20000_L2E, zfac = 0.5)
```



6.2.3.8 Superposer des couches

Pour superposer des données vectorielles avec un objet dessiné au moyen d'une fonction de **rasterVis**, on peut utiliser la fonction **layer()** du package **latticeExtra** (Sarkar & Andrews, 2016). Dans notre exemple, on superpose des polygones au dessus d'un raster ; pour cela, dans **layer()**, on utilise la fonction **sp.polygons()** du package **sp** (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a). Pour d'autres types d'entités, on utilisera **sp.points()** ou **sp.lines()**.

```
vectorplot(FR_MNT20000_L2E) +
  latticeExtra::layer(sp.polygons(FR_ctr_L2E, col = "cyan4", fill = "transparent"))
```



6.2.3.9 Assembler des cartes

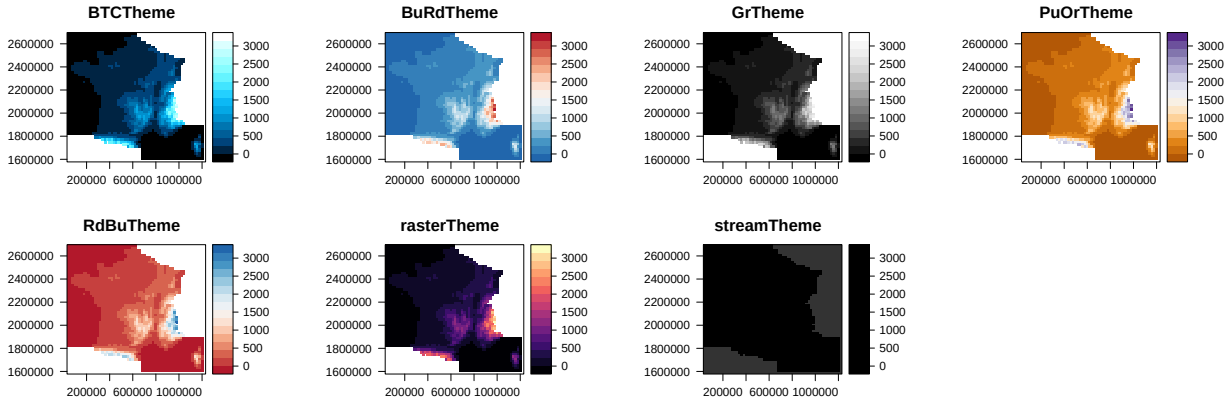
Les objets créés étant de classe **trellis**, il faut procéder de la même manière que pour les cartes créées avec la fonction **spplot()** du package (Pebesma & Bivand, 2005 ; Bivand *et al.*, 2013a) (§ 6.2.1.9).

6.2.3.10 Thèmes graphiques

Plusieurs thèmes de représentations cartographiques sont proposés par le package **rasterVis**. Les palettes utilisées proviennent des packages **RColorBrewer** (Neuwirth, 2014) et **hexbin** (Carr *et al.*, 2015). Pour pouvoir utiliser les styles, il faut fournir une fonction de thème à l'argument **par.settings** des différentes fonctions de cartographie. Les fonctions qui sont proposées sont les suivantes :

```
apropos("Theme$", ignore.case = FALSE)
##  [1] "BTCTheme"     "BuRdTheme"    "GrTheme"      "infernoTheme" "magmaTheme"   "plasmaTheme"
##  [7] "PuOrTheme"    "rasterTheme"  "RdBuTheme"    "simpleTheme"  "streamTheme"  "viridisTheme"
## [13] "YlOrRdTheme"
```

Elles correspondent aux styles suivants :



6.2.4 Package maptools

Comme mentionné plus haut, le package **maptools** (Bivand & Lewin-Koh, 2015) est spécialisé dans la lecture et l'écriture de fichiers au format Shapefile (§ 2.2.1), ainsi que dans la conversion de formats d'objets spatiaux (§ 1.3.1, 1.3.2 et 1.3.3). Outre les fonctions cartographiques de syntaxe conventionnelle précédemment décrites (§ 6.1.5), il propose également des fonctions utilisant la syntaxe de type *Trellis*.

```
library(maptools)
```

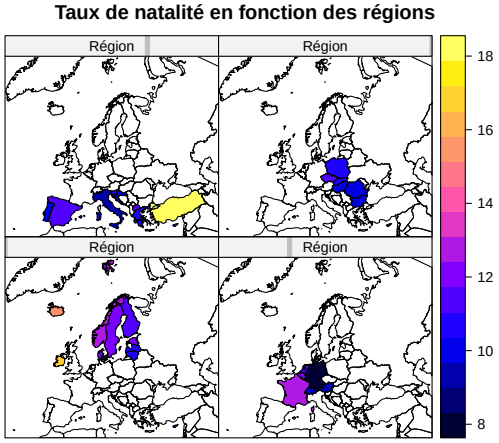
6.2.4.1 Faceting

La fonction **CCmaps()** permet de faire du *faceting* en réalisant une carte choroplèthe conditionnelle, c'est-à-dire qu'elle est capable de fournir une carte choroplète pour chaque modalité d'une variable qualitative. Cette variable qualitative doit être présentée sous la forme d'un objet de classe **shingle**. Il faut utiliser la fonction **as.shingle()** pour transformer un **factor** en **shingle**.

```
EU_ctr1_LAEA_shg <- as.shingle(EU_ctr1_LAEA@data$part)
str(EU_ctr1_LAEA_shg)
## Class 'shingle'  atomic [1:68] 3 1 3 NA 2 NA 2 4 3 4 ...
##   ..- attr(*, "levels")=List of 4
##   .. ..$ : int [1:2] 1 1
##   .. ..$ : int [1:2] 2 2
##   .. ..$ : int [1:2] 3 3
##   .. ..$ : int [1:2] 4 4
##   .. ..- attr(*, "class")= chr "shingleLevel"
```

Ici, on représente la carte choroplèthe conditionnelle. La variable conditionnelle doit être entrée sous forme de liste dans l'argument **cvar** de la fonction **CCmaps()**. Cette dernière ne renvoie pas un objet **trellis**, mais un objet de classe **SpatialPolygonsDataFrame**, dont la table attributaire contient la variable de départ, mais aussi les valeurs de cette dernière répartie en autant de colonnes que de modalités présentes dans la variable conditionnelle. Même si l'on stocke le résultat dans une variable, la carte est automatiquement dessinée.

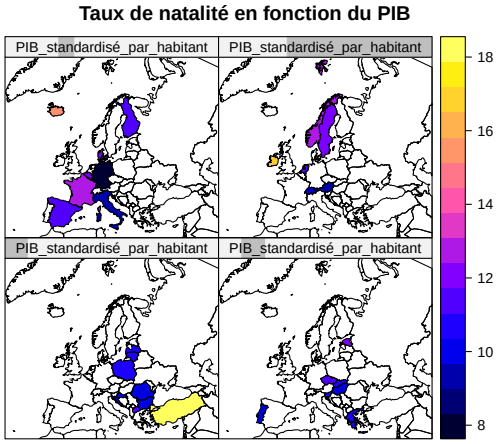
```
map_mt_ccm <- CCmaps(obj = EU_ctr1_LAEA, zcol = "birth_rate", cvar = list(Région = EU_ctr1_LAEA_shg),
                     main = "Taux de natalité en fonction des régions")
```




```
summary(map_mt_ccm)
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##        min     max
## x 1558231 7650000
## y 1300000 6450000
## Is projected: TRUE
## proj4string :
## [+proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +units=m
## +no_defs]
## Data attributes:
##    birth_rate        Région_1        Région_2        Région_3        Région_4
##  Min.   : 8.301   Min.   :10.42   Min.   : 8.301   Min.   : 9.672   Min.   : 9.87
##  1st Qu.:10.100   1st Qu.:11.34   1st Qu.: 9.899   1st Qu.: 9.911   1st Qu.:10.24
##  Median :10.875   Median :11.89   Median :11.254   Median :10.701   Median :10.62
##  Mean   :11.352   Mean   :12.45   Mean   :10.779   Mean   :11.309   Mean   :10.72
##  3rd Qu.:11.877   3rd Qu.:12.57   3rd Qu.:11.663   3rd Qu.:11.414   3rd Qu.:11.20
##  Max.   :17.884   Max.   :16.81   Max.   :12.952   Max.   :17.884   Max.   :11.66
##  NA's   :35       NA's   :58      NA's   :59       NA's   :58       NA's   :61
```

La variable conditionnelle peut être un facteur, comme dans l'exemple précédent, ou bien elle peut être une variable quantitative discrétisée. Pour passer d'une variable numérique directement en un objet de classe **shingle**, il faut utiliser la fonction **equal.count()** du package **lattice** (Sarkar, 2008). Attention, le nom de la liste de l'argument **cvar**, ne doit comporter ni espace, ni caractère spécial.

```
EU_ctr2_LAEA_shg <- lattice::equal.count(EU_ctr1_LAEA@data$gdpps_pop, number = 4, overlap = 0)
str(EU_ctr2_LAEA_shg)
## Class 'shingle'  atomic [1:68] NA NA NA NA 0.0312 ...
##   ..- attr(*, "levels")=List of 4
##   .. ..$ : num [1:2] 0.00854 0.01575
##   .. ..$ : num [1:2] 0.0161 0.0235
##   .. ..$ : num [1:2] 0.0261 0.0309
##   .. ..$ : num [1:2] 0.0309 0.078
##   .. ..- attr(*, "class")= chr "shingleLevel"
CCmaps(obj = EU_ctr1_LAEA, zcol = "birth_rate",
       cvar = list("PIB_standardisé_par_habitant" = EU_ctr2_LAEA_shg),
       main = "Taux de natalité en fonction du PIB")
```



6.3 Syntaxe de type *Grammar of Graphics*

Le *Grammar of Graphics* est issue des travaux réalisés par Leeland Wilkinson sur la visualisation graphique. Ce dernier a décrit cette syntaxe, en 2005, dans son livre *The Grammar of Graphics*. Sous R, cette syntaxe, tout comme celle de type *Trellis*, repose sur le *Grid graphics engine* dépendant du package **grid** de Paul Murrel (R Core Team, 2016). Cette syntaxe a été popularisée sous R *via* le package **ggplot2** (Wickham, 2009). Comme expliqué pour la syntaxe de type *Trellis* (§ 6.2), les paramètres définis par la fonction **par()**, n'ont aucun effet sur les fonctions utilisant ce type de syntaxe. Ici aussi, on peut stocker les graphiques comme n'importe quel objet, et l'on peut les réutiliser *a posteriori*. Les fonctions cartographiques reposant sur la *Grammar of Graphics* ne sont pas compatibles avec les fonctions graphiques conventionnelles du package **graphics**.

6.3.1 Package ggplot2

Le package **ggplot2** (Wickham, 2009) [10] est un package spécialisé dans la production de sorties graphiques. Il n'est pas cantonné à la visualisation de données conventionnelle, et permet en effet de réaliser des représentations cartographiques. Cependant, il ne gère pas (ou de manière peu satisfaisante) les objets de classes **sp** ou **raster** ; ces derniers doivent donc être préalablement transformés en **data.frame** avant d'être manipulés.

Comme n'importe quel objet de classe **ggplot**, les cartes ainsi produites pourront être assignées dans des objets R.

```
library(ggplot2)
```

6.3.1.1 Préparation des données et affichage

Quel que soit le type de carte que l'on souhaite dessiner, la syntaxe est toujours la même : il faut réaliser un premier appel avec la fonction **ggplot()**, puis on utilise une fonction plus spécifique selon le type d'entité que l'on souhaite dessiner. Les données doivent impérativement être fournies à l'argument **data** sous le format **data.frame**. Cependant, elles peuvent êtres fournies à l'argument **data** de la fonction **ggplot()** ou à ceux des fonctions secondaires propres aux types d'entités dessinées. Attention, pour conserver la même unité sur les axes des abscisses et des ordonnées, il faut veiller à bien utiliser la fonction **coord_equal()** (dont le ratio, par défaut, vaut 1).

10. Site web du package **ggplot2** : http://ggplot2.org/.

Pour représenter n'importe quel type de données spatiales, une autre possibilité est d'utiliser la fonction **qplot()** (pour *quick plot*). Il s'agit d'une fonction conçue pour les utilisateurs plus familiers de la syntaxe de type *Painter's Model*. Plus facile d'utilisation, elle offre cependant moins de souplesse que les fonctions reposant sur la *Grammar of Graphics* ; son utilisation ne sera donc pas détaillée ici.

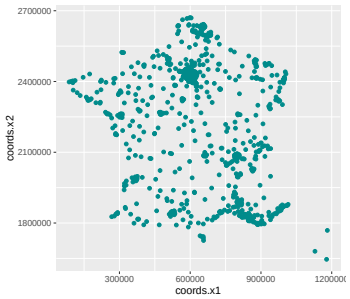
Avant toute chose, il convient donc de transformer les objets spatiaux en **data.frame**. Nous allons donc voir comment procéder avec les différents types d'entités vectorielles ou avec les objets matriciels.

Classe SpatialPoints(DataFrame). Pour tracer des entités ponctuelles, les objets de classe **SpatialPoints(DataFrame)** doivent êtres convertis en **data.frame** à l'aide de la fonction **as.data.frame()**.

```
class(FR_com_L2E)
## [1] "SpatialPointsDataFrame"
## attr(,"package")
## [1] "sp"
FR_com_L2E_df <- as.data.frame(FR_com_L2E)
head(FR_com_L2E_df)
##   insee_com    nom_comm z_moyen superficie population coords.x1 coords.x2
## 0     40088         DAX      14       1970       21.0  325858.0   1861357
## 1     26198  MONTELIMAR      93       4640       35.5  791705.1   1953288
## 2     64445         PAU     211       3152       82.8  381660.7   1816810
## 3     19275       USSEL     658       5102       10.2  597442.9   2061089
## 4     58194      NEVERS     190       1735       37.5  662348.4   2221236
## 5     33056 BLANQUEFORT      13       3398       14.6  367080.3   1995752
```

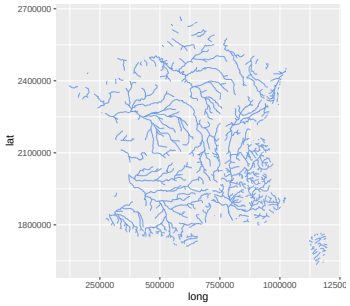
Pour dessiner une carte à partir du **data.frame**, il suffit alors d'utiliser la syntaxe habituelle du package **ggplot2** ; on associe ici les fonctions **ggplot()** et **geom_point()**.

```
ggplot(data = FR_com_L2E_df, mapping = aes(x = coords.x1, y = coords.x2)) +
  geom_point(color = "cyan4") +
  coord_equal()
```



Classe SpatialLinesDataFrame. Pour dessiner des entités linéaires, il est possible de dessiner directement les objets de classe **SpatialLinesDataFrame**. Attention, pour dessiner correctement les lignes, il ne faut pas utiliser la fonction **geom_line()**, mais la fonction **geom_path()**.

```
ggplot(data = FR_riv_L2E, mapping = aes(x = long, y = lat, group = group)) +
  geom_path(col = "cornflowerblue") +
  coord_equal()
```



Cependant, avec cette manière de faire, il ne sera par exemple pas possible de colorer les arcs en fonction d'une variable. Il est donc recommandé de convertir les objets de classe **SpatialLinesDataFrame** en **data.frame**. Cette conversion est rendue possible par l'utilisation de la fonction **fortify()**. En premier argument de cette fonction, on doit fournir l'objet **SpatialLinesDataFrame**, et l'on donne à l'argument *region*, un identifiant unique des entités géographiques à représenter.

```
class(FR_riv_L2E)
```

```
## [1] "SpatialLinesDataFrame"
## attr(,"package")
## [1] "sp"
FR_riv_L2E@data$id <- seq_along(FR_riv_L2E)
FR_riv_L2E_df <- fortify(FR_riv_L2E, region = "id")
```

L'objet renvoyé par la fonction **fortify()** est un **data.frame** contenant des longitudes (*long*), des latitudes (*lat*), un identifiant de point unique (nom de la colonne précédemment fournie à l'argument **region** de la fonction **fortify()**), un identifient d'entité (*group*).

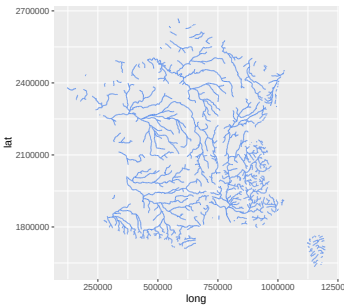
```
class(FR_riv_L2E_df)
## [1] "data.frame"
head(FR_riv_L2E_df)
##       long     lat order piece id group
## 1 627387.8 1743969     1     1  0   0.1
## 2 628383.2 1744071     2     1  0   0.1
## 3 625823.8 1743506     1     1  1   1.1
## 4 627387.8 1743969     2     1  1   1.1
## 5 628383.2 1744071     1     1  2   2.1
## 6 630450.1 1743475     2     1  2   2.1
```

Les données de la table attributaire ayant été perdues en cours de manipulation, il faut les ajouter en joignant le **data.frame** renvoyé par **fortify()** à la table attributaire de l'objet **sp** de départ.

```
FR_riv_L2E_df <- merge(FR_riv_L2E_df, FR_riv_L2E@data, by = "id")
head(FR_riv_L2E_df)
##    id     long     lat order piece group   length strahler
## 1   1 625823.8 1743506     1     1   1.1 1165.685        5
## 2   1 627387.8 1743969     2     1   1.1 1165.685        5
## 3  10 640337.8 1743075     1     1  10.1 1531.371        5
## 4  10 642361.4 1743988     2     1  10.1 1531.371        5
## 5 100 641841.5 1725156     1     1 100.1  100.000        4
## 6 100 643149.5 1726198     2     1 100.1  100.000        4
```

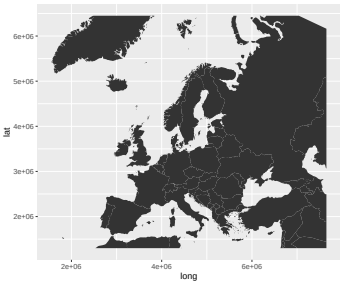
On peut à présent réaliser la carte à l'aide des fonctions graphiques du package **ggplot2** ; il faut simplement bien définir l'argument **group** de la fonction **aes()** avec la colonne *group* du **data.frame** précédemment préparé.

```
ggplot(data = FR_riv_L2E_df, mapping = aes(x = long, y = lat, group = group)) +
  geom_path(col = "cornflowerblue") +
  coord_equal()
```



Classe SpatialPolygonsDataFrame. La situation est la même que celle présentée pour les **SpatialLinesDataFrame**. Il est possible de dessiner directement les objets de classe **SpatialPolygonsDataFrame**, mais dans ce cas, il ne sera pas possible de colorer les polygones en fonction d'une variable.

```
ggplot(data = EU_ctr1_LAEA, mapping = aes(x = long, y = lat, group = group)) +
  geom_polygon() +
  coord_equal()
```



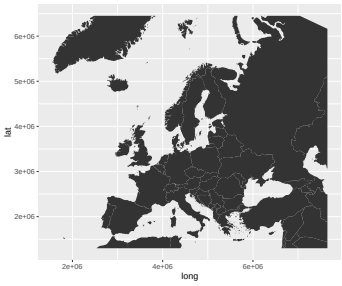
Pour dessiner, de manière satisfaisante, des polygones à partir des données de la classe **SpatialPolygonsDataFrame**, les étapes du reformatage sont identiques à celles réalisées pour un objet de type **SpatialLinesDataFrame**. On

transforme les données en **data.frame** à l'aide de la fonction **fortify()**, puis on adjoint les données attributaires perdues.

```
class(EU_ctr1_LAEA)
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
EU_ctr1_LAEA_df <- fortify(EU_ctr1_LAEA, region = "id")
EU_ctr1_LAEA_df <- merge(EU_ctr1_LAEA_df, EU_ctr1_LAEA@data, by = "id")
head(EU_ctr1_LAEA_df)
##   id    long     lat order  hole piece group iso_a2 iso_a3    name continent           part
## 1  1 5147498 2215522     1 FALSE     1   1.1     AL    ALB Albania    Europe Southern Europe
## 2  1 5151037 2213471     2 FALSE     1   1.1     AL    ALB Albania    Europe Southern Europe
## 3  1 5159123 2203518     3 FALSE     1   1.1     AL    ALB Albania    Europe Southern Europe
## 4  1 5164817 2194573     4 FALSE     1   1.1     AL    ALB Albania    Europe Southern Europe
## 5  1 5174057 2192470     5 FALSE     1   1.1     AL    ALB Albania    Europe Southern Europe
## 6  1 5179453 2189433     6 FALSE     1   1.1     AL    ALB Albania    Europe Southern Europe
##    area pop pop_dens gdpps gdpps_pop birth death birth_rate death_rate
## 1 27400  NA       NA    NA        NA    NA    NA         NA         NA
## 2 27400  NA       NA    NA        NA    NA    NA         NA         NA
## 3 27400  NA       NA    NA        NA    NA    NA         NA         NA
## 4 27400  NA       NA    NA        NA    NA    NA         NA         NA
## 5 27400  NA       NA    NA        NA    NA    NA         NA         NA
## 6 27400  NA       NA    NA        NA    NA    NA         NA         NA
```

On peut alors réaliser la carte à l'aide des fonctions graphiques de **ggplot2**. Pour dessiner les polygones, on utilise la fonction **geom_polygon()**.

```
ggplot(data = EU_ctr1_LAEA_df, mapping = aes(x = long, y = lat, group = group)) +
  geom_polygon() +
  coord_equal()
```



Classe raster. Le package **ggplot2** ne permet pas de tracer directement des rasters. Il faut donc les convertir en points à l'aide de la fonction **rasterToPoints()** du package **raster** (Hijmans, 2015). On transforme alors la **matrix** renvoyée en un **data.frame()**. Notez qu'il existe une solution plus simple avec le package **rasterVis** (Perpiñán & Hijmans, 2014), qui propose une fonction permettant de dessiner directement des objets de classe **raster** (§ 6.3.2.1).

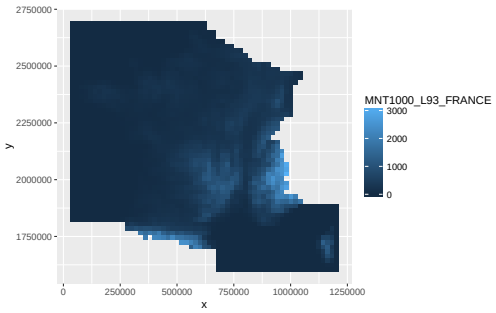
```
class(FR_MNT20000_L2E)
## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"
FR_MNT20000_L2E_pt <- raster::rasterToPoints(FR_MNT20000_L2E)
FR_MNT20000_L2E_pt <- as.data.frame(FR_MNT20000_L2E_pt)
```

Par défaut, les trois noms de colonnes sont x, y et le nom du raster de d'origine pour la variable donnée en Z. Dans l'exemple suivant, le raster *FR_MNT20000_L2E* provenait de la transformation du raster *FR_MNT1000_L93*, c'est pourquoi il porte le nom de ce dernier et que c'est ce qui apparaît en nom de troisième colonne de l'objet **FR_MNT20000_L2E_pt**.

```
head(FR_MNT20000_L2E_pt)
##           x       y MNT1000_L93_FRANCE
## 1  40783.02 2687461                  0
## 2  60783.02 2687461                  0
## 3  80783.02 2687461                  0
## 4 100783.02 2687461                  0
## 5 120783.02 2687461                  0
## 6 140783.02 2687461                  0
FR_MNT20000_L2E@data@names
## [1] "MNT1000_L93_FRANCE"
```

On peut alors dessiner la carte en utilisant la fonction **geom_raster()**.

```
ggplot(data = FR_MNT20000_L2E_pt, mapping = aes(x = x, y = y)) +
  geom_raster(mapping = aes(fill = MNT1000_L93_FRANCE)) +
  coord_equal()
```



6.3.1.2 Assigner une carte dans un objet

Le package **ggplot2** dépend du package **grid** (R Core Team, 2016), il permet donc d'assigner des graphiques (ici des cartes) dans des objets R et de les réutiliser par la suite.

```
map_gg1 <- ggplot(data = EU_ctr1_LAEA_df, mapping = aes(x = long, y = lat, group = group)) +
  geom_polygon() +
  coord_equal()
```

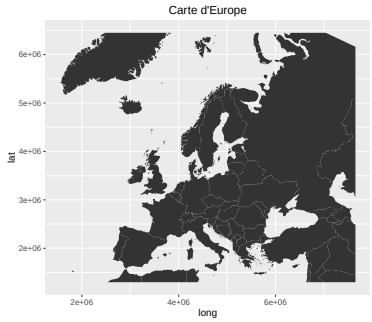
Comme c'est l'habitude avec ce package, les objets ainsi créés sont de classes **gg** et **ggplot**.

```
class(map_gg1)
## [1] "gg"     "ggplot"
str(map_gg1, max.level = 1)
## List of 9
##  $ data       :'data.frame': 20665 obs. of  21 variables:
##  $ layers     :List of 1
##  $ scales     :Classes 'ScalesList', 'ggproto' <ggproto object: Class ScalesList>
##     add: function
##     clone: function
##     find: function
##     get_scales: function
##     has_scale: function
##     input: function
##     n: function
##     non_position_scales: function
##     scales: list
##     super:  <ggproto object: Class ScalesList>
##  $ mapping    :List of 3
##  $ theme      : list()
##  $ coordinates:Classes 'CoordFixed', 'CoordCartesian', 'Coord', 'ggproto' <ggproto object: Class CoordFixed, Coo
##     aspect: function
##     distance: function
##     expand: TRUE
##     is_linear: function
##     labels: function
##     limits: list
##     range: function
##     ratio: 1
##     render_axis_h: function
##     render_axis_v: function
##     render_bg: function
##     render_fg: function
##     train: function
##     transform: function
##     super:  <ggproto object: Class CoordFixed, CoordCartesian, Coord>
##  $ facet      :List of 1
##   ..- attr(*, "class")= chr [1:2] "null" "facet"
##  $ plot_env   :<environment: R_GlobalEnv>
##  $ labels     :List of 3
##  - attr(*, "class")= chr [1:2] "gg" "ggplot"
```

6.3.1.3 Éléments contextuels

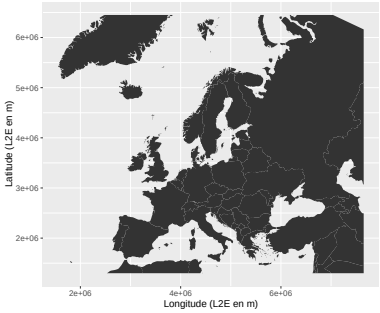
Titre. Pour écrire un titre, il suffit d'appeler la fonction **ggtitle()**.

```
map_gg1 + ggtitle("Carte d'Europe")
```



Axes. Comme d'usage avec **ggplot2**, pour écrire les noms des axes, il suffit d'appeler la fonction **labs()**, et pour masquer les axes, il faut utiliser le fonction **theme()** et définir les éléments adéquats avec la fonction **element_blank()**.

```
map_gg1 + labs(x = "Longitude (L2E en m)", y = "Latitude (L2E en m)")
```




```
map_gg1 +
  theme(axis.title = element_blank(),
        axis.text  = element_blank(),
        axis.ticks = element_blank())
```

Légende. Les légendes sont gérées comme à l'habitude avec ce package. Voici la liste des noms, classés par ordre alphabétique, des six premières fonctions que l'on peut utiliser.

```
head(apropos("^scale_"))
## [1] "scale_alpha"            "scale_alpha_continuous" "scale_alpha_discrete"
## [4] "scale_alpha_identity"   "scale_alpha_manual"     "scale_color_brewer"
```

Orientation. Par défaut, il n'existe pas de fonction qui permette de tracer un symbole d'orientation. Pour pallier à ce manque, on peut utiliser les fonctions **north()** et **north2()** que propose le package **ggsn** (Santos Baquero, 2016) (§ 6.3.3.1).

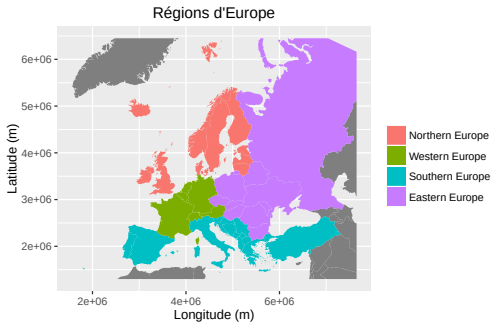
Échelle. C'est la même chose, pour en obtenir qui sont compatibles avec **ggplot2**, il faut charger le package **ggsn** qui propose la fonction **scalebar()** (§ 6.3.3.1).

6.3.1.4 Carte typologique

Bien entendu, on peut colorer la carte à partir d'une variable qualitative. Pour cela, on fournit simplement une variable qualitative à la fonction **geom_polygon()**. Comme dans tout graphique **ggplot2**, dans la fonction **aes**, que ce soit celle de la fonction **ggplot()** ou celle de la fonction **geom_polygon()** (selon comment on souhaite procéder), on peut choisir de colorer la bordure du polygone (**colour**), l'intérieur (**fill**), etc. Les données manquantes sont ici admises.

```
ggplot(data = EU_ctr1_LAEA_df, mapping = aes(x = long, y = lat, group = group)) +
  geom_polygon(mapping = aes(fill = part)) +
  coord_equal() +
  labs(x = "Longitude (m)", y = "Latitude (m)", fill = "") +
```

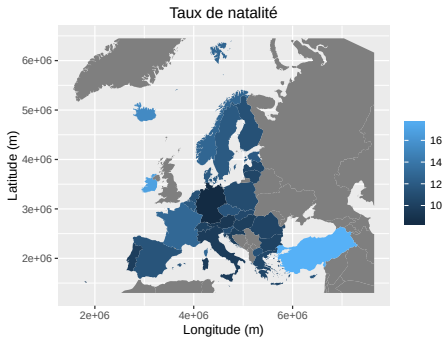
```
  ggtitle("Régions d'Europe")
```



6.3.1.5 Carte choroplèthe

On peut réaliser une carte choroplèthe si l'on fournit une variable quantitative à la fonction **aes()**. Notez que l'échelle de couleurs produite est continue, ce qui rend la lecture de la carte difficile. Les données manquantes sont autorisées.

```
ggplot(data = EU_ctr1_LAEA_df, mapping = aes(x = long, y = lat, group = group)) +
  geom_polygon(mapping = aes(fill = birth_rate)) +
  coord_equal() +
  labs(x = "Longitude (m)", y = "Latitude (m)", fill = "") +
  ggtitle("Taux de natalité")
```



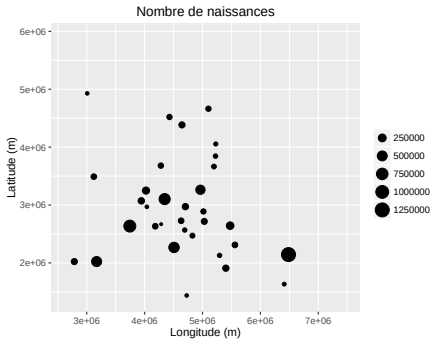
6.3.1.6 Carte en symboles proportionnels

On peut réaliser des cartes en symboles proportionnels en utilisant la fonction **geom_point()**. Dans notre exemple, on récupère tout d'abord les coordonnées des centroïdes des polygones grâce à l'application de la fonction **coordinates()** sur l'objet **SpatialPolygonsDataFrame** originel. On transforme alors la **matrix** obtenue en **data.frame**, et on lui adjoint les données attributaires.

```
EU_labpt_LAEA_df <- data.frame(long = coordinates(EU_ctr1_LAEA)[, 1],
                               lat  = coordinates(EU_ctr1_LAEA)[, 2])
EU_labpt_LAEA_df <- cbind(EU_labpt_LAEA_df, EU_ctr1_LAEA@data)
```

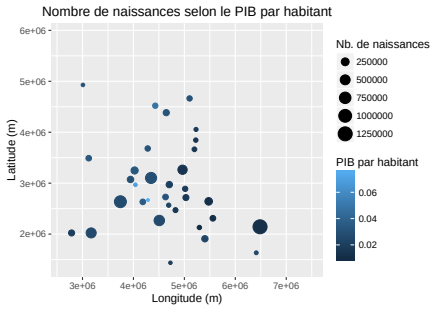
On peut alors réaliser une carte avec la fonction **geom_point()**. Pour faire varier la taille des points en fonction des valeurs de la variable d'intérêt, le nom de cette dernière est passée à l'argument **size** de la fonction **aes()**.

```
ggplot(data = EU_labpt_LAEA_df, mapping = aes(x = long, y = lat)) +
  geom_point(mapping = aes(size = birth)) +
  coord_equal() +
  labs(x = "Longitude (m)", y = "Latitude (m)", size = "") +
  ggtitle("Nombre de naissances")
```



On peut représenter les variations de deux variables. Dans l'exemple suivant, la taille des points (**size**) varie selon une variable, et leur couleur (**color**) selon une autre variable. Dans ce cas, les légendes respectives sont automatiquement positionnées l'une en dessous de l'autre. Notez que les données manquantes sont autorisées.

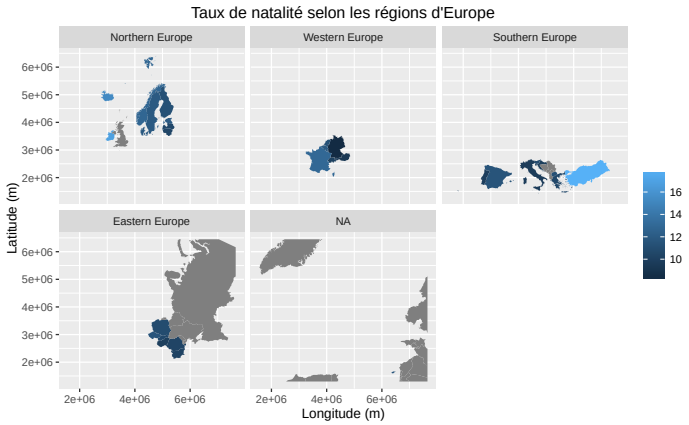
```
ggplot(data = EU_labpt_LAEA_df, mapping = aes(x = long, y = lat)) +
  geom_point(mapping = aes(size = birth, color = gdpps_pop)) +
  coord_equal() +
  labs(x = "Longitude (m)", y = "Latitude (m)",
       size = "Nb. de naissances", color = "PIB par habitant") +
    ggtitle("Nombre de naissances selon le PIB par habitant")
```



6.3.1.7 Faceting

Bien entendu, on peut faire du *faceting*. Pour cela, on utilise les fonctions usuelles **facet_wrap()** ou **facet_grid()**.

```
ggplot(data = EU_ctr1_LAEA_df, mapping = aes(x = long, y = lat, group = group)) +
  geom_polygon(mapping = aes(fill = birth_rate)) +
  facet_wrap(~ part) +
  coord_equal() +
  labs(x = "Longitude (m)", y = "Latitude (m)", fill = "") +
  ggtitle("Taux de natalité selon les régions d'Europe")
```

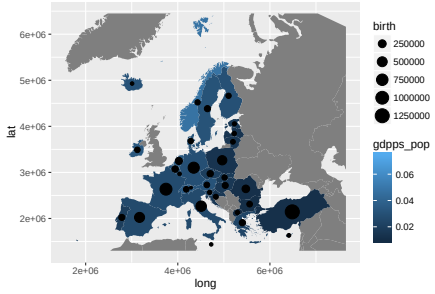


6.3.1.8 Superposer des couches

Pour superposer des couches, il n'y a pas de difficulté particulière. Si les données sont différentes pour chacune des couches dessinées, il faut simplement bien veiller à faire un premier appel vide à la fonction **ggplot()**. Dans ce cas, les données des différentes couches sont alors spécifiées à l'intérieur de ces dernières fonctions et non

pas dans la fonction **ggplot()** de départ. Les couches sont ajoutées dans l'ordre souhaité grâce aux fonctions adaptées aux différents types d'entités (**geom_point()**, **geom_path()**, **geom_polygon()**, ou **geom_raster()**).

```
ggplot() +
  geom_polygon(data = EU_ctr1_LAEA_df,
               mapping = aes(x = long, y = lat, group = group, fill = gdpps_pop)) +
  geom_point(data = EU_labpt_LAEA_df,
             mapping = aes(x = long, y = lat, size = birth)) +
  coord_equal()
```



6.3.1.9 Gérer le système de coordonnées de la carte

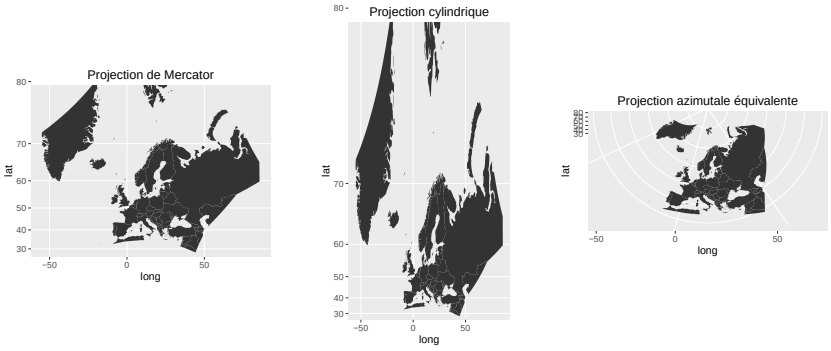
La fonction **coord_map()** permet de choisir le système de coordonnées de la carte produite. Mais pour ce faire, la carte que l'on souhaite représenter doit obligatoirement être issue d'un objet dont les coordonnées sont initialement exprimées en WGS 84 (code EPSG 4326). Par défaut, le système de coordonnées utilisé par la fonction **coord_map()** est la projection de Mercator (code EPSG 3857).

Dans l'exemple suivant, on commence par préparer les données. On dispose d'une couche des pays d'Europe exprimée en LAEA (code EPSG 3035). On la convertit en WGS 84 (code EPSG 4326), puis on la transforme en un **data.frame** que l'on fournit à la fonction **ggplot()**.

```
EU_ctr1_W84 <- spTransform(EU_ctr1_LAEA, CRSobj = CRS("+init=epsg:4326"))
EU_ctr1_W84_df <- fortify(EU_ctr1_W84, region = "id")
EU_ctr1_W84_df <- merge(EU_ctr1_W84_df, EU_ctr1_W84@data, by = "id")
map_gg2 <- ggplot(EU_ctr1_W84_df, mapping = aes(long, lat, group = group)) +
  geom_polygon()
```

On peut à présent représenter la carte dans d'autres systèmes de coordonnées grâce à la fonction **coord_map()**.

```
map_gg2 + ggtitle("Projection de Mercator") + coord_map()
map_gg2 + ggtitle("Projection cylindrique") + coord_map("cylindrical")
map_gg2 + ggtitle("Projection azimutale équivalente") + coord_map("azequalarea")
```



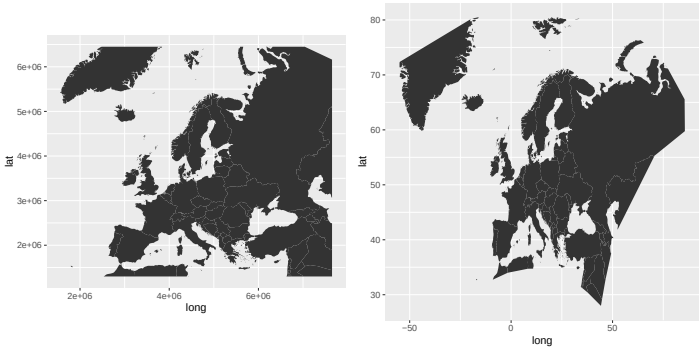
La fonction **coord_map()** dépend de la fonction **mapproject** du package **mapproj** (McIlroy *et al.*, 2015) ; on se référera donc à la documentation de cette fonction si l'on désire connaître les systèmes de coordonnées disponibles, ainsi que leurs paramètres.

Par ailleurs, le package **ggalt** (Rudis, 2016) propose la fonction **coord_proj()** qui, quant à elle, utilise la syntaxe PROJ.4 (Evenden *et al.*, 2015), ce qui permet de pouvoir se servir des codes EPSG.

6.3.1.10 Assembler des cartes

Pour assembler les graphiques de classe **ggplot**, on peut utiliser la fonction **grid.arrange()** du package **gridExtra** (Auguie, 2016). On fournit à la fonction les différents objets **ggplot** que l'on souhaite assembler, et l'on définit les nombres de lignes et de colonnes souhaités à l'aide des arguments **nrow** et **ncol**.

```
gridExtra::grid.arrange(map_gg1, map_gg2, ncol = 2)
```

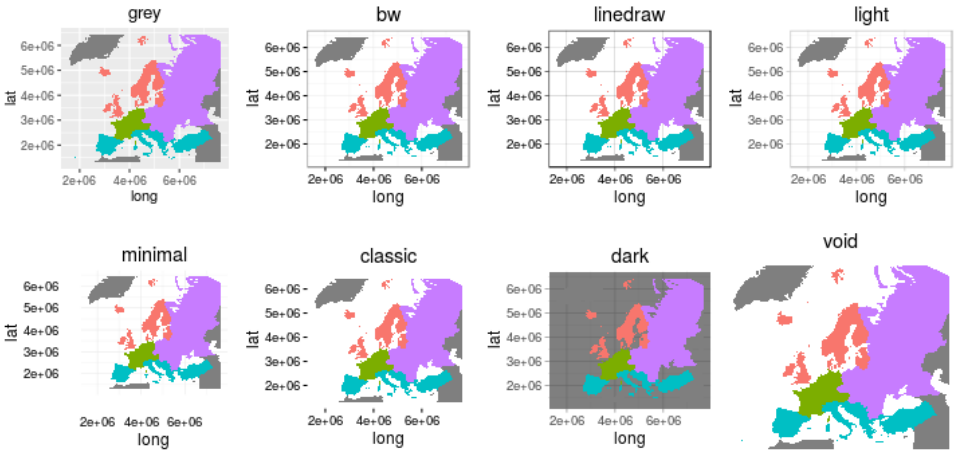


6.3.1.11 Thèmes graphiques

On peut utiliser les thèmes graphiques habituels de **ggplot2**, ou bien ceux proposés par le package **ggthemes** (Arnold, 2016), mêmes s'ils ne sont pas spécialement adaptés à la représentation cartographique.

```
library(ggthemes)
apropos("theme_")
```

```
##  [1] "theme_base"            "theme_bw"              "theme_calc"
##  [4] "theme_classic"         "theme_dark"            "theme_economist"
##  [7] "theme_economist_white" "theme_excel"           "theme_few"
## [10] "theme_fivethirtyeight" "theme_foundation"      "theme_gdocs"
## [13] "theme_get"             "theme_gray"            "theme_grey"
## [16] "theme_hc"              "theme_igray"           "theme_light"
## [19] "theme_linedraw"        "theme_map"             "theme_minimal"
## [22] "theme_pander"          "theme_par"             "theme_replace"
## [25] "theme_set"             "theme_solarized"       "theme_solarized_2"
## [28] "theme_solid"           "theme_stata"           "theme_tufte"
## [31] "theme_update"          "theme_void"            "theme_wsj"
```



6.3.2 Package rasterVis

Le package **rasterVis** (Perpiñán & Hijmans, 2014) qui, comme nous l'avons vu (§ ??), est spécialisé dans la visualisation des rasters, permet de gérer les rasters en interaction avec les fonctions du package **ggplot2** (Wickham, 2009).

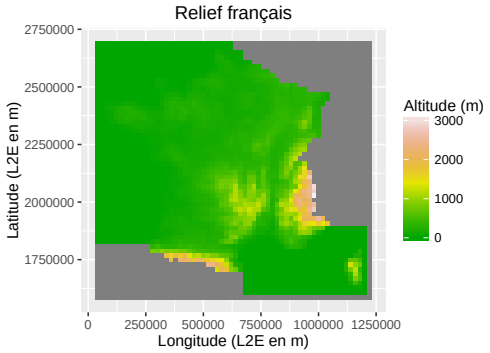
```
library(rasterVis)
```

6.3.2.1 Affichage

La fonction **gplot()**, compatible avec les fonctions du package **ggplot2**, permet de dessiner une carte directement à partir d'un objet de classe **raster**, ce que ne permet pas de faire **ggplot2** (§ 6.3.1.1). En revanche, elle ne gère pas les objets matriciels de classe **sp**.

```
library(ggplot2)
gplot(FR_MNT20000_L2E) +
```

```
  geom_tile(mapping = aes(fill = value)) +
  scale_fill_gradientn(name = "Altitude (m)", colours = terrain.colors(10)) +
  coord_equal() +
  labs(x = "Longitude (L2E en m)", y = "Latitude (L2E en m)") +
  ggtitle("Relief français")
```



6.3.3 Package ggsn

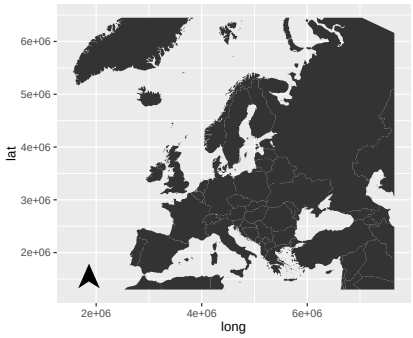
Le package **ggsn** (Santos Baquero, 2016) propose des fonctions de représentation d'éléments contextuels d'une carte compatibles avec celles du package **ggplot2**, et que ce dernier ne permet pas de dessiner (§ 6.3.1.1).

```
library(ggsn)
```

6.3.3.1 Éléments contextuels

Orientation. Les fonctions **north()** et **north2()** permettent de dessiner deux types de flèche du Nord sur une carte réalisée avec le package **ggplot2**. Avec la fonction **north()**, on ajoute le symbole à la carte déjà définie. Pour cela, il faut fournir à la fonction le **data.frame** ayant servie à construire la carte. On peut choisir la position du symbole grâce à l'argument **location**, et le type de représentation grâce à l'argument **symbol**.

```
map_gg1 + north(data = EU_ctr1_LAEA_df, location = "bottomleft", symbol = 12)
```

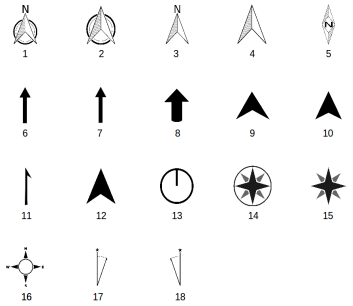


La fonction **north2()** fonctionne un peu différemment. En effet, en argument principal (**ggp**), elle prend la carte **ggplot**, et la position est gérée par les arguments **x** et **y** définissant la longitude et la latitude de l'emplacement du symbole.

```
north2(ggp = map_gg1, x = 0.24, y = 0.17, symbol = 12)
```

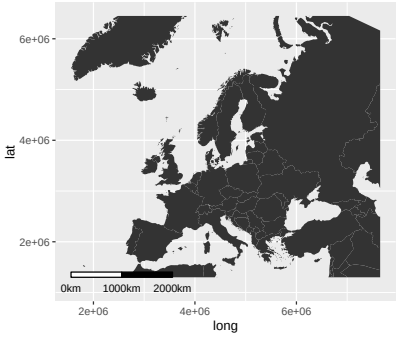
Les symboles proposées par les fonctions **north()** et **north2()** sont au nombre de 18. On peut les visualiser à l'aide de la fonction **northSymbols()**.

```
northSymbols()
```



Échelle. La fonction **scalebar()** permet de dessiner une échelle sur une carte réalisée avec le package **ggplot2**. On fournit à la fonction le **data.frame** ayant servi à dessiner la carte. On positionne l'échelle grâce à l'argument **location**, on choisit la distance (en km) à représenter avec **dist**. D'autres arguments permettent de gérer la taille du texte, l'espacement entre le texte et la barre, etc. Notez que l'on peut dessiner l'échelle dans un système non projeté (**dd2km = TRUE**) et définir l'ellipsoïde de référence (**model**).

```
map_gg1 + scalebar(data = EU_ctr1_LAEA_df, location = "bottomleft",
                   dist = 1000, height = 0.02, st.dist = 0.04, st.size = 3)
```



6.3.4 Package tmap

Le package **tmap** (Tennekes, 2016) est spécialisé dans la production cartographique, même s'il propose, par ailleurs, quelques outils d'analyse spatiale. Ses fonctions cartographiques reposent sur une syntaxe de type *Grammar of Graphics*, mais travaillant directement sur des objets spatiaux de classe **sp** ou **raster**, ce qui le rend plus facile d'utilisation que le package **ggplot2**. En revanche, même s'ils utilisent tous les deux le même type de syntaxe, les fonctions des packages **tmap** et **ggplot2** ne sont pas compatibles.

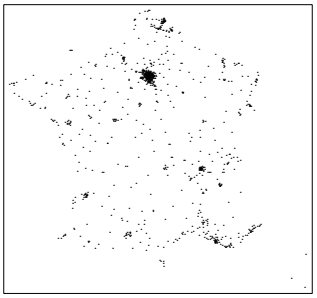
```
library(tmap)
```

6.3.4.1 Affichage

Quel que soit le type de carte que l'on souhaite dessiner, la syntaxe est toujours la même : il faut réaliser un premier appel avec la fonction **tm_shape()**, à laquelle on fournit un objet spatial, puis on utilise une fonction plus spécifique selon le type d'entités que l'on souhaite dessiner.

Pour tracer des points, il faut utiliser la fonction **tm_dots()**. On peut gérer la couleur avec l'argument **col**, le degré de transparence des ces dernières avec **alpha**, la taille avec **size**. Notez qu'il est possible de définir ces arguments avec des noms de variables de la table attributaire.

```
tm_shape(FR_com_L2E) +
  tm_dots()
```



Une autre possibilité est d'utiliser la fonction **qmap()** (pour *quick thematic map*), qui est un équivalent de la fonction **qplot()** (pour *quick plot*) du package **ggplot2** (Wickham, 2009). Il s'agit d'une fonction conçue pour les utilisateurs plus familiers de la syntaxe de type *Painter's Model*. Plus facile d'utilisation, elle offre cependant moins de souplesse que les fonctions reposant sur la *Grammar of Graphics* ; son utilisation ne sera donc pas détaillée ici.

Pour tracer des lignes, il faut utiliser la fonction **tm_lines()**. On peut gérer la couleur avec l'argument **col**, le degré de transparence des ces dernières avec **alpha**, l'empattement avec **lwd**, le type de rendu de la ligne avec **lty**. Les arguments **col** et **lwd** peuvent être définis avec des variables de la table attributaire.

```
tm_shape(FR_riv_L2E) +
  tm_lines()
```

Pour tracer des polygones, on peut utiliser plusieurs fonctions correspondant à plusieurs représentations graphiques :

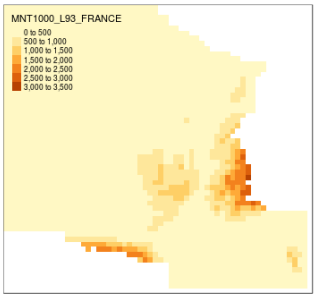
- **tm_polygons()** : colore l'intérieur des polygones et trace leur contour ;
- **tm_fill()** : colore l'intérieur des polygones et ne trace pas leur contour ;
- **tm_borders()** : ne colore pas les polygones, mais trace leur contour.

On peut gérer les couleurs avec les arguments **col** et **border.col**, le degré de transparence des ces dernières avec **alpha** et **border.alpha**, l'empattement des contours **lwd**, le style de rendu des contours avec **lty**. L'argument **col** peut être défini avec une variable de la table attributaire.

```
tm_shape(EU_ctr1_LAEA) +
    tm_polygons()
```

Pour tracer des rasters, il faut utiliser la fonction **tm_raster()**. Les objets de classes **sp** et **raster** sont tous les deux gérés par le package. Les différents arguments permettent, entre autres, de paramétrer les couleurs (**col**), la palette de couleurs (**palette**), le degré de transparence des couleurs (**alpha**), le nombre de classes (**n**), le type de discrétisation (**style**), les valeurs des bornes de classe de la discrétisation (**breaks**), etc.

```
tm_shape(FR_MNT20000_L2E) +
  tm_raster()
```

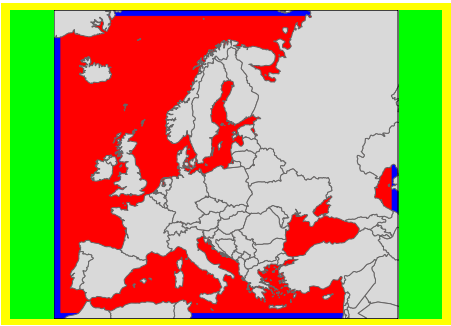


Notez que, contrairement à l'usage habituel dans R, lorsque la valeur de la couleur est égale à **NA**, ici, cela ne signifie pas l'absence de couleur, mais cette dernière prendra la teinte définie par défaut pour les valeurs manquantes.

La fonction **tm_layout()** permet de gérer de très nombreux paramètres graphiques tels que :
- le titre (position, taille, couleur) ;
- la palette de couleurs (teintes, saturation et teintes en transformées sépia) ;
- les marges ;
- la police de caractères ;
- la légende (position, taille, couleur) ;
- la type de fond de carte.

Les marges d'une carte produite avec la fonction **tm_shape()** se décomposent en plusieurs parties, que l'on peut visualiser par les couleurs suivantes, grâce à la fonction **tm_layout()** et l'argument **design.mode = TRUE**.
- jaune : les marges externes ;
- vert : la fenêtre de dessin ;
- bleu : les marges internes ;
- rouge : l'emprise géographique.

```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons() +
  tm_layout(design.mode = TRUE)
```



6.3.4.2 Assigner une carte dans un objet

Il est possible d'assigner une carte dans un objet R, comme avec les fonctions **spplot()** et **ggplot()**.

```
map_tm_eu <- tm_shape(EU_ctr1_LAEA) +
  tm_polygons(col = "death_rate")
map_tm_fr <- tm_shape(FR_ctr_L93) +
  tm_polygons(col = NA) +
  tm_shape(EU_ctr1_LAEA) +
  tm_polygons(col = "death_rate", legend.show = FALSE)
```

L'objet ainsi créé est de classe **tmap**. Il comporte autant d'éléments que de fonctions appelées lors de sa création. Ces éléments portent les noms des fonctions appelées.

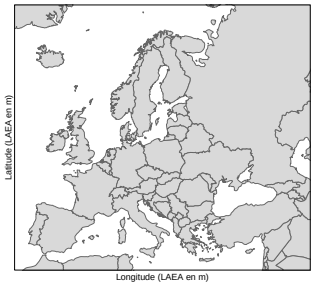
```
class(map_tm_eu)
## [1] "tmap"
class(map_tm_fr)
## [1] "tmap"
names(map_tm_eu)
## [1] "tm_shape"   "tm_fill"    "tm_borders"
names(map_tm_fr)
## [1] "tm_shape"   "tm_fill"    "tm_borders" "tm_shape"   "tm_fill"    "tm_borders"
```

6.3.4.3 Éléments contextuels

Titre. Pour Ajouter un titre il faut utiliser l'argument **title** de la fonction **tm_layout()**.

Axes. On peut choisir de nommer les axes grâce aux fonctions **tm_xlab()** et **tm_ylab()**. Des arguments permettent de gérer la taille (**size**) et l'orientation du texte (**rotation**).

```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons() +
  tm_xlab("Longitude (LAEA en m)", size = 0.8, rotation =  0) +
  tm_ylab("Latitude (LAEA en m)" , size = 0.8, rotation = 90)
```



Légende. On peut gérer les paramètres de la légende grâce à la fonction **`tm_legend()`** (ou **`tm_layout()`**, le comportement est identique) qui propose de nombreux arguments pour cela :

- **`legend.show`** : affiche ou masque la légende ;
- **`legend.only`** : trace la légende uniquement, sans dessiner la carte (utile pour afficher une légende commune à de nombreuses petites cartes) ;
- **`legend.outside`** : trace la légende à l'extérieur d'une carte ou d'un groupe de carte réalisé par *faceting* ;
- **`legend.position`** : la position du bloc ;
- **`legend.width`** : la largeur du bloc ;
- **`legend.height`** : la hauteur du bloc ;
- **`legend.bg.color`** : la couleur de fond du bloc ;
- **`legend.frame`** : affiche ou masque le contour du bloc, ou bien définit la couleur de ce dernier ;
- **`legend.title.size`** : la taille du titre ;
- **`legend.text.size`** : la taille du texte ;
- **`legend.text.color`** : la couleur du texte ;
- **`legend.format`** : le type d'arrondi et les textes de séparation des bornes des classes.

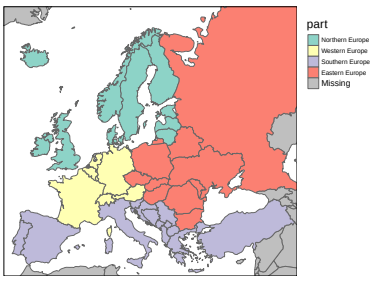
Par ailleurs, des styles de légendes prédéfinis sont proposés par les fonctions suivantes :

```
apropos("tm_format")
```

```
## [1] "tm_format_Europe"      "tm_format_Europe2"     "tm_format_Europe_wide"
## [4] "tm_format_NLD"         "tm_format_NLD_wide"    "tm_format_World"
## [7] "tm_format_World_wide"
```

Dans l'exemple ci-après, la fonction **`tm_format_Europe2()`**, dessine la légende en haut à gauche de la fenêtre, à l'extérieur de l'emprise de la carte. Par défaut, sans l'utilisation de cette dernière, elle serait également positionnée en haut à gauche, mais à l'intérieur de l'emprise de la carte.

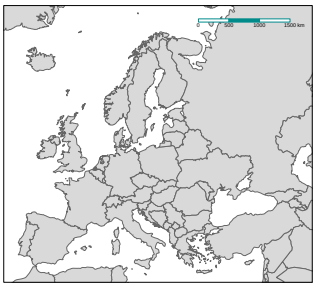
```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons(col = "part") +
  tm_format_Europe2()
```



Échelle. La fonction **`tm_scale_bar()`** permet de dessiner une échelle. Plusieurs arguments permettent de personnaliser, les valeurs des limites des subdivisions de l'échelle, la position, les couleurs, etc. La position peut être gérée de plusieurs manières :

- 2 valeurs numériques comprises entre 0 et 1 ;
- 2 chaînes de caractères (les valeurs en majuscules sont plus proches du bord de la carte) :
  - **`"left"`**, **`"LEFT"`**, **`"center"`**, **`"right"`** ou **`"RIGHT"`** ;
  - **`"top"`**, **`"TOP"`**, **`"center"`**, **`"bottom"`** ou **`"BOTTOM"`**.

```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons() +
  tm_scale_bar(position = c("right", "top"), color.dark = "cyan4")
```

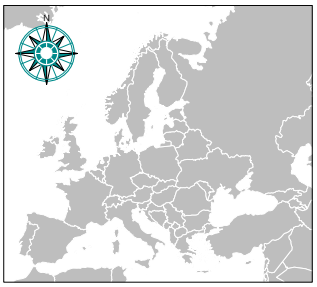


Orientation. La fonction **tm_compass()** propose divers types de symboles pour indiquer l'orientation d'une carte :

- **arrow** : une flèche ;
- **4star** : une étoile à 4 branches ;
- **8star** : une étoile à 8 branches ;
- **radar** : un radar ;
- **rose** : une rose des vents.

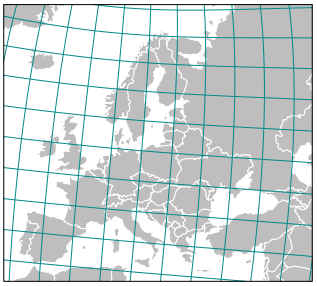
Par ailleurs, de nombreux arguments, permettent de choisir les couleurs (**text.color**, **color.dark**, **color.light**), le point cardinal à indiquer (**cardinal.directions**) et sa direction (**north**), la position du symbole d'orientation (**position** ; géré de la même manière que pour l'échelle), etc.

```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons(col = "grey", border.col = "white") +
  tm_compass(position = c("left", "top"), type = "rose", color.dark = "cyan4")
```



Graticules. On peut dessiner des graticules grâce à la fonction **tm_grid()**. Par défaut, le système de coordonnées utilisé est celui de la carte. On peut toutefois en utiliser un autre système en entrant un nom de projection ou un code EPSG dans l'argument **projection**. On peut choisir d'afficher ou non les coordonnées des graticules.

```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons(col = "grey", border.col = "white") +
  tm_grid(projection = "2154", col = "cyan4",
          labels.inside.frame = FALSE, labels.size = 0)
```

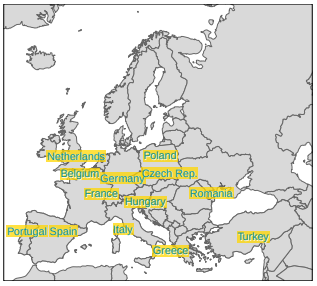


Étiquettes. Pour écrire des étiquettes, on peut utiliser la fonction **tm_text()**. Pour éviter les chevauchements d'étiquettes, on peut se servir de l'argument **auto.placement = TRUE**. De très nombreux arguments sont proposés pour paramétrer l'affichage du texte. On peut par exemple définir :

- **fontface** : l'empattement de la police de caractères ;
- **fontfamily** : la fonte de caractères ;
- **case** : le choix de la casse de caractères (**"upper"**, **"lower"**, **NA**) ;

- **shadow** : affiche ou non une ombre sous le texte ;
- **bg.color** : la couleur de l'étiquette de fond ;
- **bg.alpha** : la transparence de la couleur de l'étiquette de fond ;
- **auto.placement** : évite les chevauchements d'étiquettes ;
- **remove.overlap** : retire les étiquettes qui se chevauchent ;
- **along.lines** : aligne le texte le long des polylignes ;
- etc.

```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons() +
tm_shape(EU_ctr1_LAEA[EU_ctr1_LAEA@data$pop >= 10e6 & ! is.na(EU_ctr1_LAEA@data$pop), ]) +
  tm_polygons() +
  tm_text(text = "name", auto.placement = TRUE,
          col = "cyan4", shadow = TRUE, bg.color = "gold", bg.alpha = 0.75)
```



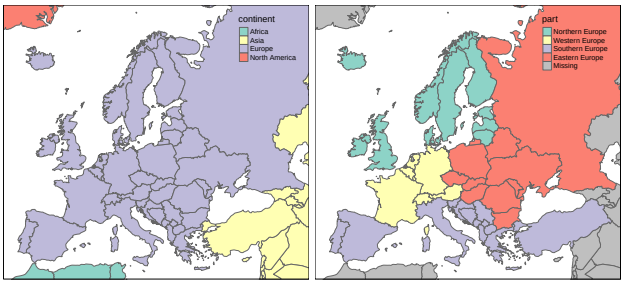
Sources des données. La fonction **tm_credits()** permet de renseigner les sources de données. L'emplacement est géré de la même manière que pour l'échelle ou le symbole d'orientation.

```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons(col = "grey", border.col = "white") +
  tm_credits(text = "Source : package tmap", position = c("right", "bottom"), col = "cyan4", size = 2)
```

6.3.4.4 Carte typologique

Pour dessiner une carte typologique, il faut utiliser les fonctions **tm_polygons()** ou **tm_fill()** et spécifier dans l'argument **col**, un vecteur des noms de colonnes de la table attributaire correspondant aux variables qualitatives que l'on désire représenter. Comme dans l'exemple ci-dessous, il y aura alors autant de cartes dessinées et de légendes associées que de variables définies dans l'argument **col**. Notez, que les valeurs manquantes sont gérées automatiquement.

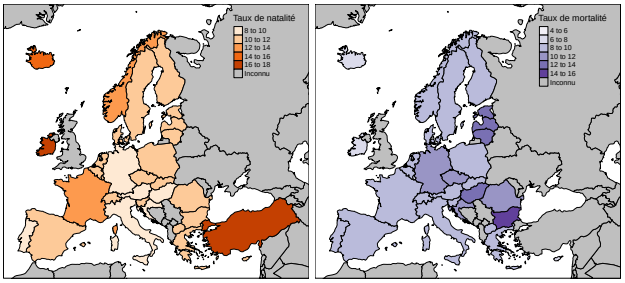
```
tm_shape(EU_ctr1_LAEA) +
    tm_polygons(col = c("continent", "part"))
```



6.3.4.5 Carte choroplèthe

Pour tracer une carte choroplèthe, c'est le même principe que pour une carte typologique, mais cette fois on fournit le nom d'une variable quantitative à l'argument **col** des fonctions **tm_polygons()** ou **tm_fill()**. Notez que les données manquantes sont autorisées dans les variables d'intérêt.

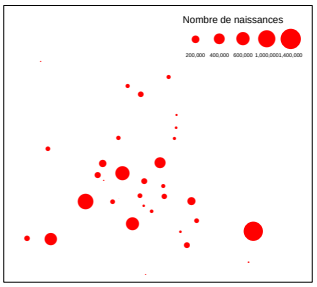
```
tm_shape(EU_ctr1_LAEA) +
    tm_polygons(col = c("birth_rate", "death_rate"),
                palette = list("Oranges", "Purples"),
                border.col = "black", textNA = "Inconnu",
                title = c("Taux de natalité", "Taux de mortalité"))
```



6.3.4.6 Carte en symboles proportionnels

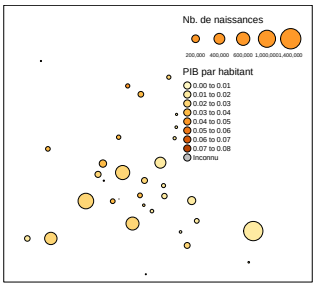
Pour tracer des symboles (cercles ou disques), il faut utiliser la fonction **tm_dots()**. Les données manquantes sont autorisées dans la variable d'intérêt qui est assignée à l'argument **size**. Dans l'exemple suivant, on fournit directement un objet de classe **SpatialPolygonsDataFrame**, et les symboles seront directement tracés à l'emplacement des centroïdes, il n'est donc pas nécessaire de travailler sur un objet de classe **SpatialPointsDataFrame**, même si c'est bien des entités ponctuelles que l'on représente.

```
tm_shape(EU_ctr1_LAEA) +
  tm_dots(size = "birth", col = "red", border.col = "black",
          scale = 2, title.size = "Nombre de naissances")
```



Dans le même genre de représentation, on peut choisir de faire varier la taille des disques selon une variable et leur couleur selon une autre variable. Pour cela, il faut utiliser la fonction **tm_bubbles()** et fournir à l'argument **col** le nom de la variable selon laquelle on souhaite colorer les disques. Les données manquantes sont autorisées dans la variable d'intérêt.

```
tm_shape(EU_ctr1_LAEA) +
  tm_bubbles(size = "birth", col = "gdpps_pop", border.col = "black",
             scale = 2, textNA = "Inconnu",
             title.size = "Nb. de naissances", title.col = "PIB par habitant")
```



6.3.4.7 Dessiner des isolignes

Le package **tmap** permet de dessiner des isolignes. Avant de les tracer, il faut tout d'abord les déterminer à l'aide de la fonction **smooth_map()**. Cette fonction propose de nombreux arguments d'entrée, parmi lesquels :

- **shp** : un objet de class **sp** (sauf **SpatialLines(DataFrame)**) ou **raster** ;
- **nlevels** : le nombre souhaité de classes ;
- **style** : la méthode de discrétisation (**"fixed"**, **"equal"**, **"pretty"**, **"quantile"** ou **"kmeans"**) ;
- **breaks** : la bornes des classes (si **style = "fixed"**) ;
- **bandwidth** : les dimsensions de la bande passante de la méthode d'estimation par noyau ;
- **to.Raster** : le type d'objet renvoyé (**FALSE** : **SpatialGridDataFrame** ; **TRUE** : **RasterLayer**).

```
FR_MNT20000_L2E_iso <- smooth_map(shp = FR_MNT20000_L2E, to.Raster = TRUE)
```

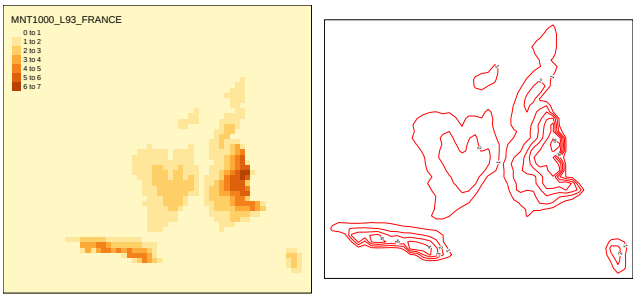
Cette fonction renvoie une liste contenant les éléments suivants :

- **raster** : un raster des zones définies par les iosolignes (un objet de classe **SpatialGridDataFrame** si l'argument **to.Raster = FALSE**, sinon un objet de classe **RasterLayer**) ;
- **iso** : les lignes de contours (un objet de classe **SpatialLinesDataFrame**) ;
- **bbox** : l'emprise géographique ;
- **ncol** : le nombre de colonnes du raster ;
- **nrow** : le nombre de lignes du raster ;
- **cell.area** : l'aire d'une maille ;
- **bandwidth** : la résolution de la bande passante.

```
names(FR_MNT20000_L2E_iso)
## [1] "raster"    "iso"       "polygons"  "bbox"      "nrow"      "ncol"      "cell.area"
## [8] "bandwidth"
```

Il ne reste plus qu'à utiliser les fonctions graphiques adaptées aux différents types de sorties que l'on souhaite utiliser.

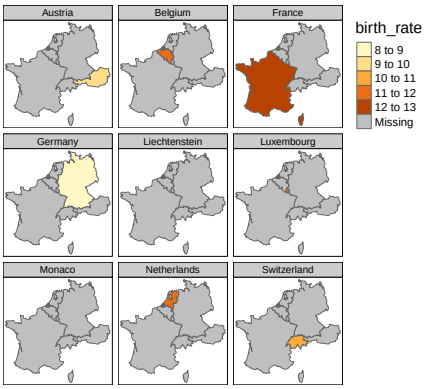
```
tm_shape(FR_MNT20000_L2E_iso$raster) +
  tm_raster()

tm_shape(FR_MNT20000_L2E_iso$iso) +
  tm_iso()
```



6.3.4.8 Faceting

On peut faire du *faceting* grâce à la fonction **tm_facets()**.

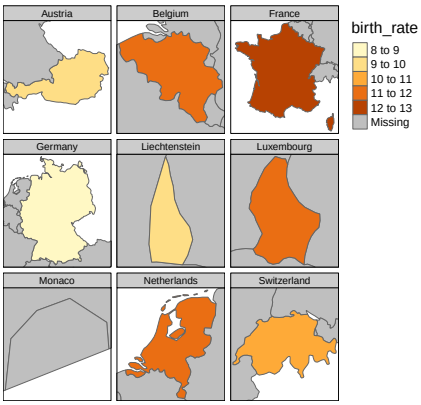
```
tm_shape(EU_ctr1_LAEA[EU_ctr1_LAEA@data$part %in% "Western Europe", ]) +
  tm_polygons("birth_rate") +
  tm_facets(by = "name")
```



On peut zoomer directement sur l'entité d'intérêt en utilisant l'argument **free.coords = TRUE**.

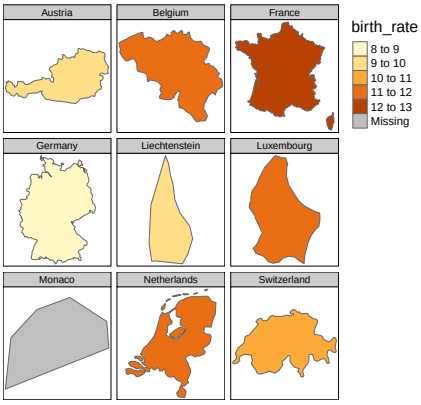
```
tm_shape(EU_ctr1_LAEA[EU_ctr1_LAEA@data$part %in% "Western Europe", ]) +
  tm_polygons("birth_rate") +
```

```
  tm_facets(by = "name", free.coords = TRUE, drop.units = FALSE)
```



On peut retirer les entités autres celle d'intérêt en utilisant l'argument **`drop.units = TRUE`**.

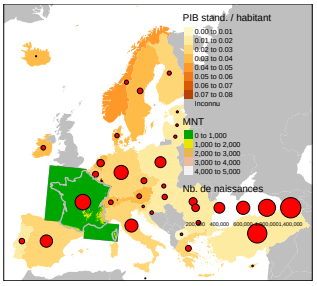
```
tm_shape(EU_ctr1_LAEA[EU_ctr1_LAEA@data$part %in% "Western Europe", ]) +
  tm_polygons("birth_rate") +
  tm_facets(by = "name", free.coords = TRUE, drop.units = TRUE)
```



6.3.4.9 Superposer des couches

Pour superposer des couches, il ne faut pas procéder de la même manière que ce qui est fait lorsqu'on utilise la package **`ggplot2`**. En effet, il faut réaliser autant d'appels à la fonction **`tm_shape()`** qu'il y a de couches à dessiner. Ces appels successifs sont réalisé à l'aide du signe plus ("**+**").

```
tm_shape(EU_ctr1_LAEA) +
  tm_fill("gdpps_pop", title = "PIB stand. / habitant", textNA = "Inconnu") +
tm_shape(FR_MNT1000_LAEA) +
  tm_raster(palette = terrain.colors(5), title = "MNT") +
tm_shape(EU_ctr1_LAEA, is.master = TRUE) +
  tm_bubbles("birth", col = "red", border.col = "black",
             scale = 2, title.size = "Nb. de naissances") +
tm_shape(FR_ctr_L93) +
  tm_borders(col = "grey", lwd = 2)
```



Par défaut, la couche principale est la première, sinon c'est celle pour la quelle la fonction **`tm_shape`** présente l'argument **`is.master = TRUE`**. Elle définit l'emprise géographique de la carte et son système de coordonnées, même si elle n'apparaît pas en premier. Par ailleurs, comme on peut le constater, les différentes légendes appelées ne se superposent pas ; elles sont dessinées automatiquement les unes en dessous des autres.

6.3.4.10 Gérer le système de coordonnées de la carte

La fonction **tm_shape()** permet de choisir le système de coordonnées de la carte produite. Il peut donc être différent de celui de la couche que l'on souhaite dessiner. Le système de coordonnées est fourni à l'argument **projection** de la fonction sous la forme d'un code EPSG. Ici, on dispose d'une couche de l'Europe en LAEA (code EPSG 3035) que l'on reprojette en NSIDC Sea Ice Polar Stereographic North (code EPSG 3413).

```
tm_shape(EU_ctr1_LAEA, projection = "3413") +
  tm_polygons(col = "grey", border.col = "white")
```

On vient de le voir, le système de coordonnées de la carte n'est donc pas obligatoirement celui de la couche dessinée. Avec ce package, il est donc possible de réaliser des cartes "projetées à la volée". C'est-à-dire qu'il n'est pas obligatoire que toutes les couches soient définies dans le même système de coordonnées. Par défaut, c'est le système de la première couche appelée qui définira celui de la carte, mais on peut décider de la couche qui servira de référence en définissant l'argument **is.master = TRUE** dans l'appel à la fonction **tm_shape()** de la couche principale.

Dans l'exemple ci-dessous, on dispose de deux couches référencées dans deux systèmes de coordonnées différents (l'Europe en LAEA et la France en WGS 84).

```
proj4string(EU_ctr1_LAEA)
## [1] "+proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +units=m +no_defs"
proj4string(FR_ctr_W84)
## [1] "+init=epsg:4326 +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

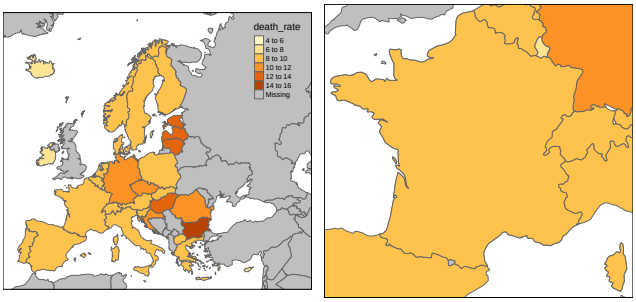
On peut pourtant les superposer sans aucun problème en réalisant deux appels successifs à la fonction **tm_shape()**. La carte est produite en WGS 84, le système de coordonnées de la couche principale définissant l'emprise (**is.master = TRUE**), même si elle n'est pas dessinée en premier.

```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons() +
tm_shape(FR_ctr_W84, is.master = TRUE) +
  tm_polygons(col = "cyan4")
```

6.3.4.11 Assembler des cartes

Le package **gridExtra** (Auguie, 2016) ne permet de pas gérer les objets de classe **tmap**. Aussi, pour assembler des cartes sur une même fenêtre graphique, il faut directement utiliser le package **grid** (R Core Team, 2016), ce qui peut être un peu fastidieux.

```
library(grid)
grid.newpage()
pushViewport(viewport(layout = grid.layout(1, 2)))
print(map_tm_eu, vp = viewport(layout.pos.col = 1))
print(map_tm_fr, vp = viewport(layout.pos.col = 2))
```



6.3.4.12 Thèmes graphiques

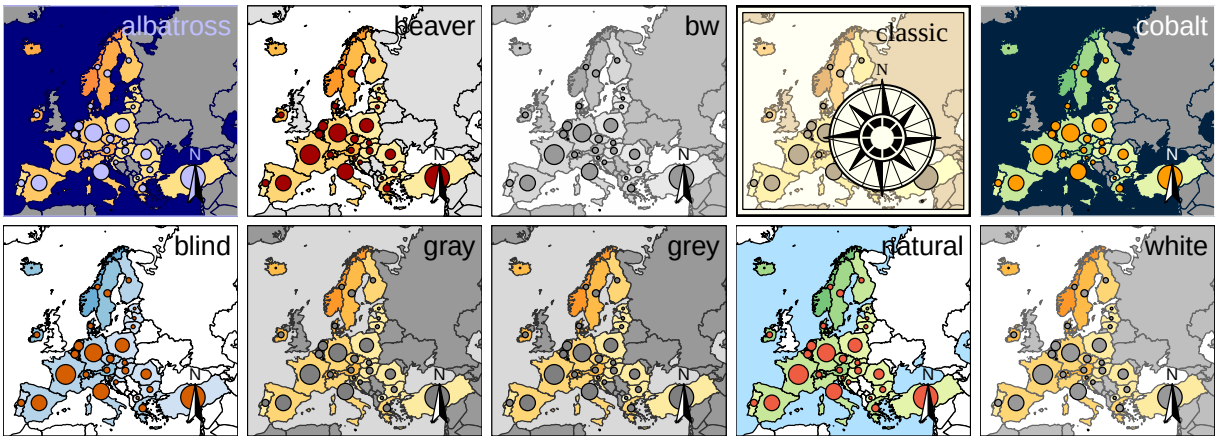
Le package **tmap**, comme **ggplot2**, propose plusieurs thèmes graphiques. Ces derniers définissent par défaut des couleurs de fonds de carte, de polygones, de bordures et de palette de l'échelle, mais aussi le style de la boîte autour de la carte, le style du symbole d'orientation, etc. Neuf styles, spécialement adaptés à la production cartographique, sont ainsi proposés :

- **tm_style_white** : affiche un fond blanc (défaut) ;
- **tm_style_gray** : affiche un fond gris (utile pour faire ressortir les palettes séquentielles, e.g. dans les choroplèthes) ;
- **tm_style_grey** : *idem* ;
- **tm_style_natural** : propose une émulation de vue naturelle (eaux en bleu et terres émergées en vert) ;
- **tm_style_bw** : affiche un dégradé de gris (utile pour l'impression en niveaux de gris) ;
- **tm_style_classic** : produit un style de carte à l'ancienne ;
- **tm_style_cobalt** : inspiré du style "cobalt" sous Latex/Beamer ;
- **tm_style_albatross** : inspiré du style "albatross" sous Latex/Beamer ;
- **tm_style_beaver** : inspiré du style "beaver" sous Latex/Beamer.

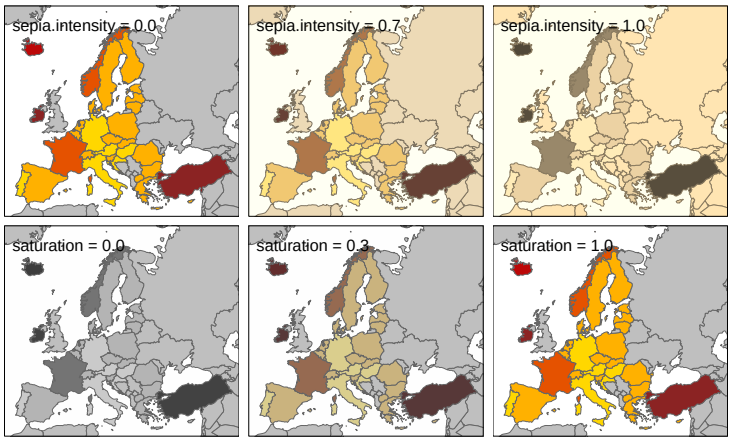
La syntaxe à adopter est la suivante :

```
tm_shape(EU_ctr1_LAEA) +
  tm_polygons() +
  tm_bubbles("birth") +
  tm_style_cobalt()
```

Graphiquement les différents styles permettent de réaliser les représentations suivantes (le rendu final exact dépendant de la fonction de forme utilisée (i.e. **tm_polygon()**, **tm_bubbles()**, **tm_raster()**, etc.) :



Par ailleurs, pour gérer les couleurs, la fonction **tm_layout()** propose une teinte sépia plus ou moins prononcée en fonction d'un degré de transparence (**sepia.intensity** ; 0 : transparent et 1 : opaque), ou encore de faire varier la saturation des couleurs (**saturation** ; 0 : niveaux de gris et 1 : couleurs inchangées).


sepia.intensity = 0.0
sepia.intensity = 0.7
sepia.intensity = 1.0
saturation = 0.0
saturation = 0.3
saturation = 1.0

Chapitre 7

Fonds de cartes

R dispose de plusieurs packages permettant d'interagir avec des fonds de cartes. Ces derniers correspondent à des photos aériennes, des images satellites, mais aussi à des plans, des cartes topographiques, des cartes routières, etc. Pour récupérer ces fonds, les packages utilisent des interfaces de programmation applicative (API, pour *Application Programming Interface*). Nous ne présenterons pas ici tous les packages existant et nous ne détaillerons pas non plus leur utilisation de manière fine. Nous présenterons brièvement la manière dont on affiche un fond de carte et comment on y superpose des entités spatiales. Ceci permettra donc à l'utilisateur d'avoir un aperçu des points communs et des différences de l'usage des principales fonctionnalités de ces packages, qui sont souvent relativement similaires.

Nature des fonds de cartes

Les fonds de cartes produits peuvent être de deux natures : statiques ou dynamiques. Dans le cas des fonds statiques, les objets spatiaux sont placés sur un fond de carte fixe, et l'utilisateur ne peut pas se déplacer librement dans la carte sans avoir à retracer celle-ci en modifiant l'emprise géographique et le niveau de zoom. Les fichiers sont alors produits à des formats d'images standards (e.g. PNG, JPEG, PDF). Dans le cas des fonds dynamiques, la carte entière est construite pour chacune des requêtes de l'utilisateur, qui peut alors se déplacer librement dans la carte et zoomer sur une position particulière sans à avoir à relancer de ligne de commande. Les fichiers sont alors produits aux formats HTM ou HTML. Notez qu'il est possible de réaliser automatiquement des captures d'écran des pages web ainsi crées grâce au package **`webshot`** (Chang, 2016).

Serveurs disponibles

Selon les packages, on peut attaquer un ou plusieurs des serveurs suivants : Apple iPhoto (Apple Inc., 2016), Bing Maps (Microsoft, 2016), CartoDB (CartoDB Inc., 2016), CloudMade (CloudMade, 2016), ESRI Web Map (ESRI, 2016b), Google Maps (Google, 2016), Mapnik (Pavlenko, 2016), MapQuest (MapQuest Inc., 2013), Maptoolkit (Toursprung GmbH, 2016), NASA Global Imagery Browse Services (National Aeronautics and Space Administration, 2016), NPMap (National Park Service of the United States Department of the Interior, 2016), Naver (Next Human Network, 2016), OpenStreetMap (Haklay & Weber, 2008 ; OpenStreetMap contributors, 2016), OpenWeatherMap (OpenWeatherMap Inc., 2016), Skobbler (Telenav Inc., 2016), Stamen (Stamen Design, 2016), Thunderforest (Thunderforest & OpenStreetMap contributors, 2016).

Attention, pour avoir accès aux données, certains serveurs nécessitent une clé d'identification ; c'est notamment le cas de Bing Maps ou MapQuest. D'autre part, certains serveurs ne permettent qu'un accès restreint (nombre limité de requêtes) si l'utilisateur n'as pas de clé ; c'est par exemple le cas de Google Maps.

Syntaxe

Quelque soit le package utilisé et quelque soit la nature du fond de carte, les représentations cartographiques peuvent être assignées dans des objets R.

Description des données utilisées

Dans les exemples de cette partie, nous allons utiliser la localisation spatiale de la tour imaginée et conçue par Maurice Koechlin, Émile Nouguier et Stephen Sauvestre d'Eiffel & Cie.

Nous disposons ainsi de plusieurs objets correspondants à la position de la flèche, à l'amplitude Est–Ouest de l'édifice et à son emprise au sol. Ces derniers sont stockés sous la forme d'objets conventionnels de classe **data.frame** :

```
eiffel_pt_W84 <- data.frame(x = 2.294511, y = 48.858269, name = "Tour Eiffel")
eiffel_li_W84 <- data.frame(x = c(2.293310, 2.295698), y = c(48.858248, 48.858273))
eiffel_po_W84 <- data.frame(x = c( 2.293310,  2.294470,  2.295698,  2.294523,  2.293310),
                            y = c(48.858248, 48.859040, 48.858273, 48.857474, 48.858248))
```

Il sont également disponibles sous la formes d'objets spatiaux de classe **sp** dont les coordonnées sont exprimées en WGS 84 (code EPSG 4326) ou en Mercator (code EPSG 3857) :

```
eiffel_pt_W84sp <- SpatialPoints(eiffel_pt_W84[, c("x", "y")],
                                 proj4string = CRS("+proj=longlat +datum=WGS84"))
eiffel_li_W84sp <- SpatialLines(list(Lines(list(Line(eiffel_li_W84)), "e1")),
                                proj4string = CRS("+proj=longlat +datum=WGS84"))
eiffel_po_W84sp <- SpatialPolygons(list(Polygons(list(Polygon(eiffel_po_W84)), "e1")),
                                   proj4string = CRS("+proj=longlat +datum=WGS84"))
eiffel_pt_MCTsp <- spTransform(eiffel_pt_W84sp, "+init=epsg:3857")
eiffel_li_MCTsp <- spTransform(eiffel_li_W84sp, "+init=epsg:3857")
eiffel_po_MCTsp <- spTransform(eiffel_po_W84sp, "+init=epsg:3857")
```

7.1 Cartes statiques

7.1.1 Package OpenStreetMap

Le package **OpenStreetMap** (Fellows *et al.*, 2013) permet d'accéder à divers fonds de cartes topographiques, mais aussi à des images satellite ou des plans.

```
library(OpenStreetMap)
```

7.1.1.1 Types de fonds de cartes

Ce package permet la prise en charge des fonds de cartes suivants :

- Apple Iphoto ;
- Bing Maps (nécessite une clé d'identification) ;
- CloudMade ;
- Esri ;
- Esri Topo ;
- Mapnik ;
- MapQuest (nécessite une clé d'identification) ;
- MapQuest Aerial (*idem*) ;
- Maptoolkit Topo ;
- National Park Service ;
- OpenStreetMap (par défaut) ;
- Skobbler ;
- Stamen Toner ;
- Stamen Watercolor.

Les serveurs sont décrits dans une matrice renvoyée par la fonction **getMapInfo()**. On a ainsi accès au nom du serveur et à son URL, au nom du détenteur des droits d'attribution, ainsi qu'à l'URL décrivant les droits d'attribution et d'utilisation.

```
head(getMapInfo(), 3)
##      name              url                         attribution
## [1,] "osm"             "http://openstreetmap.org/" "Â© OpenStreetMap contributors, CC-BY-SA "
## [2,] "osm-bw"          "http://openstreetmap.org/" "Â© OpenStreetMap contributors, CC-BY-SA "
## [3,] "maptoolkit-topo" "http://maptoolkit.net/"    "www.maptoolkit.net"
##      attributionTerms
## [1,] "http://www.openstreetmap.org/copyright"
## [2,] "http://www.openstreetmap.org/copyright"
## [3,] "http://www.maptoolkit.net"
```

7.1.1.2 Définition du fond de carte

L'emprise géographique de la carte est définie par deux couples de coordonnées (obligatoirement exprimées en WGS 84, et dans l'ordre suivant : latitude puis longitude), correspondant au coin supérieur gauche et au coin inférieur droit :

```
eiffel_ext_W84_tl <- c(y = 48.863453, x = 2.285242)
eiffel_ext_W84_br <- c(y = 48.853278, x = 2.302507)
```

La fonction **openmap()** permet de récupérer les tuiles du fond de carte à partir des coordonnées de l'emprise définie telle que nous venons de le mentionner. L'argument **type** permet de définir le type de fond de carte souhaité, et **zoom**, le niveau de zoom (géré automatiquement si non défini).

```
eiffel_osm_MCT <- openmap(eiffel_ext_W84_tl, eiffel_ext_W84_br, type = "osm")
```

L'objet renvoyé est une liste, de classe **OpenStreetMap**, contenant notamment une matrice de couleurs correspondant à l'image du fond de carte, l'emprise géographique de ce dernier, ainsi que le système de coordonnées et la résolution des pixels.

```
str(eiffel_osm_MCT)
## List of 2
##  $ tiles:List of 1
##   ..$ :List of 5
##   .. ..$ colorData : chr [1:575118] "#CAC9C8" "#C3B5AA" "#D9D0C9" "#D9D0C9" ...
##   .. ..$ bbox      :List of 2
##   .. .. ..$ p1: num [1:2] 254392 6251724
##   .. .. ..$ p2: num [1:2] 256314 6250002
##   .. ..$ projection:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. .. ..@ projargs: chr "+proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0 +un
##   .. ..$ xres      : int 718
##   .. ..$ yres      : int 801
##   .. ..- attr(*, "class")= chr "osmtile"
##  $ bbox :List of 2
##   ..$ p1: num [1:2] 254392 6251724
##   ..$ p2: num [1:2] 256314 6250002
##  - attr(*, "zoom")= int 16
##  - attr(*, "class")= chr "OpenStreetMap"
```

Pour afficher le fond de carte, il suffit de fournir l'objet **OpenStreetMap** à la fonction **plot()**.

```
plot(eiffel_osm_MCT)
```

7.1.1.3 Gestion du système de coordonnées

Par défaut, la carte est projetée en Mercator (code EPSG 3857). La fonction **osm()** renvoie un objet de classe **CRS** contenant les paramètres de la projection Mercator selon la syntaxe PROJ.4 (Evenden *et al.*, 2015).

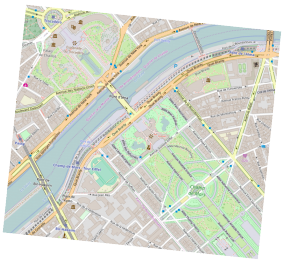
```
osm()
## CRS arguments:
##  +proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0
## +units=m +nadgrids=@null +no_defs
raster::compareCRS(osm(), CRS("+init=epsg:3857"))
## [1] TRUE
```

Si l'on souhaite superposer des entités spatiales sur le fond de carte récupéré, il faut alors qu'elles soient exprimées dans la même projection que ce dernier. La fonction **projectMercator()** permet de convertir des coordonnées depuis le WGS 84 (code EPSG 4326) vers le Mercator (code EPSG 3857). Cette fonction prend deux vecteurs de longitudes (**long**) et latitudes (**lat**) en entrée, et elle renvoie un objet de classe **matrix**. Cette fonction ne permet pas de gérer les objets de classe **sp**.

```
(eiffel_pt_MCT <- projectMercator(lat = eiffel_pt_W84$y, long = eiffel_pt_W84$x))
##         x         y
##  255423.8 6250846.7
```

On peut librement modifier le système de coordonnées du fond de carte en convertissant l'objet de classe **OpenStreetMap** grâce à la fonction **openproj()**. Dans l'exemple suivant, on reprojette le fond de carte en LAEA (code EPSG 3035).

```
eiffel_osm_LAEA <- openproj(eiffel_osm_MCT, projection = CRS("+init=epsg:3035"))
plot(eiffel_osm_LAEA)
```

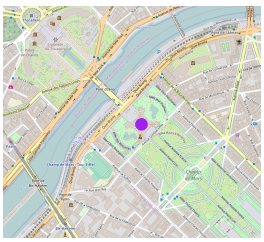


7.1.1.4 Affichage des entités spatiales

Avec ce package, on peut superposer des objets de base au fond de carte (i.e. **vector**, **matrix**, **data.frame**, etc.), grâce aux fonctions du package **graphics** (R Core Team, 2016). Pour superposer des objets spatiaux de classes **sp** ou **raster**, il faut utiliser la fonction **plot()**, avec l'argument **add = TRUE**.

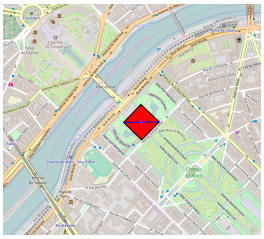
Ici, on dessine des points, contenus dans un **data.frame**, par un simple appel à la fonction **points()**.

```
plot(eiffel_osm_MCT)
points(x = eiffel_pt_MCT["x"], y = eiffel_pt_MCT["y"],
       col = "purple", pch = 19, cex = 3)
```



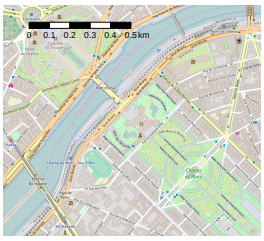
Pour superposer plusieurs couches spatiales, il n'y a pas de difficulté particulière, il suffit de réaliser les appels successifs aux différentes fonctions : **points()**, **lines()**, **polygon()** pour des objets de type conventionnel, ou bien des appels successifs à la fonction **plot()** avec l'argument **add = TRUE** pour des objets spatiaux. Ici, on se sert de la deuxième possibilité, car on dispose de trois objets de classe **sp**.

```
plot(eiffel_osm_MCT)
plot(eiffel_po_MCTsp, col = "red"   , add = TRUE)
plot(eiffel_li_MCTsp, col = "blue"  , add = TRUE)
plot(eiffel_pt_MCTsp, col = "purple", add = TRUE)
```



Notez que le package **OSMscale** (Boessenkool, 2016) propose la fonction **scaleBar()**, qui permet de dessiner une échelle sur les cartes du package **OpenStreetMap**.

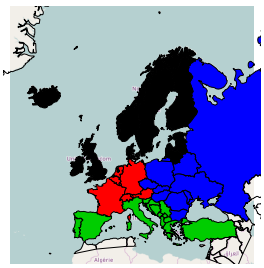
```
plot(eiffel_osm_MCT)
OSMscale::scaleBar(eiffel_osm_MCT)
```



7.1.1.5 Produire différents types de cartes

Comme il est possible de superposer aisément des objets spatiaux de classes **sp** ou **raster** à l'aide de la fonction **plot()** (§ 6.1.1.6 et 6.1.2.9), pour dessiner des cartes typologiques, des cartes choroplèthes, des cartes en symboles proportionnels, etc., il faut appliquer les même méthodes explicitées précédemment (§ 6.1.1.1 et 6.1.2.1).

```
EU_ext_W84_tl <- c(y = 75, x = -30)
EU_ext_W84_br <- c(y = 30, x =  50)
eiffel_osm_MCT <- openmap(EU_ext_W84_tl, EU_ext_W84_br, type = "osm")
plot(eiffel_osm_MCT)
plot(EU_ctr1_MCT, col = EU_ctr1_LAEA@data$part, add = TRUE)
```



7.1.2 Package RgoogleMaps

Le package **RgoogleMaps** (Loecher & Ropkins, 2015) permet de fournir une interface R pour interroger le serveur de fonds de cartes statiques de Google Maps, mais aussi ceux de Bing Maps et d'OpenStreetMap.

```
library(RgoogleMaps)
```

7.1.2.1 Types de fonds de cartes

Le package **RgoogleMaps** permet de disposer des fonds de cartes Google Maps, mais également des fonds de base d'OpenstreetMap et de Bing Maps :

- **satellite** : image satellite ;
- **terrain** : carte de caractéristiques physiques telles que le terrain et la végétation ;
- **hybrid** : couche transparente de la carte des rues sur les images satellites.plan OpenstreetMap.

7.1.2.2 Définition du fond de carte

Pour récupérer un fond de carte Google Maps, il faut utiliser la fonction **GetMap()**. La carte peut être définie à partir d'une requête de *geocoding* ou d'un couple de coordonnées (latitude et longitude), définis tous deux par l'argument **center**, et d'un niveau zoom défini par l'argument **zoom**. L'argument **maptype**, quant à lui, permet de choisir le type de fond de carte souhaité.

```
eiffel_rgm1 <- GetMap(center = "Tour Eiffel, Paris",
                      zoom = 15, maptype = "terrain")
eiffel_rgm1 <- GetMap(center = c(lat = eiffel_pt_W84$y, lon = eiffel_pt_W84$x),
                      zoom = 15, maptype = "terrain")
```

L'objet renvoyé par la fonction **GetMap()** est une liste structurée qui contient notamment les coordonnées du centre du fond de carte (**lat.center** et **lon.center**), le niveau de zoom (**zoom**), un **array** à 4 dimensions (**myTile**) dont les trois premières couches correspondent aux couleurs de l'image (valeurs entre 0 et 1 de rouge, de vert et de bleu), l'emprise géographique (**BBOX**), la résolution de l'image (**size**).

```
str(eiffel_rgm1)
## List of 8
##  $ lat.center: Named num 48.9
##   ..- attr(*, "names")= chr "lat"
##  $ lon.center: Named num 2.29
##   ..- attr(*, "names")= chr "lon"
##  $ zoom      : num 15
##  $ myTile    : num [1:640, 1:640, 1:4] 0.984 0.949 0.925 0.925 0.922 ...
##  $ BBOX      :List of 2
##   ..$ ll: num [1, 1:2] 48.85 2.28
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "lat" "lon"
##   ..$ ur: num [1, 1:2] 48.87 2.31
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "lat" "lon"
##  $ url       : chr "google"
##  $ size      : num [1:2] 640 640
##  $ SCALE     : num 1
##  - attr(*, "class")= chr "staticMap"
```

Pour récupérer le fond de carte OpenstreetMap, il faut utiliser la fonction **GetOsmMap()**. La carte se définit à partir des coordonnées d'une emprise. La structure des données obtenues est inchangée.

```
eiffel_rgm2 <- GetOsmMap(lonR = c( 2.285242,  2.302507),
                         latR = c(48.853278, 48.863453))
## [1] "http://tile.openstreetmap.org/cgi-bin/export?bbox=2.285242,48.853278,2.302507,48.863453&scale=20000&format=
```

Pour récupérer le fond de carte Bing Maps, il faut utiliser la fonction **GetBingMap()**, mais il faut disposer d'une clé d'identification. La carte se définit à partir des coordonnées du centre de la carte (**center**) et d'un niveau de zoom (**zoom**). La structure des données récupérées est toujours la même.

```
eiffel_rgm3 <- GetBingMap(center = c(lat = eiffel_pt_W84$y, lon = eiffel_pt_W84$x),
                          zoom = 15, maptype = "Road",
                          apiKey = scan("bingAPIkey.txt", what = ""))
```

Pour dessiner la carte, on utilise la fonction **PlotOnStaticMap()**.

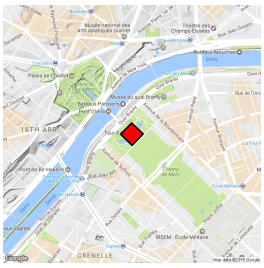
```
PlotOnStaticMap(eiffel_rgm1, mar = rep(0.5, 4))
```

7.1.2.3 Affichage des entités spatiales

De manière générale, pour dessiner des entités spatiales, il faut utiliser la fonction **PlotOnStaticMap()**. Cette fonction ne permet pas l'utilisation d'objets spatiaux de classe **sp**. Comme nous allons le voir, selon le type d'entité considéré, d'autres possibilités existent.

Pour dessiner des points, lignes ou polygones, on fournit les coordonnées aux arguments **lat** et **lon** de la fonction **PlotOnStaticMap()**. L'argument **FUN** doit, quant à lui, être défini à la valeur **points** (valeur par défaut), **lines**, ou **polygon** selon le type d'entité considérée.

```
PlotOnStaticMap(eiffel_rgm1, FUN = polygon,
                lat = c(48.858248, 48.859040, 48.858273, 48.857474, 48.858248),
                lon = c( 2.293310,  2.294470,  2.295698,  2.294523,  2.293310),
                col = "red", lwd = 3)
```



On peut également dessiner des segments grâce à la fonction **PlotArrowsOnStaticMap()**. Il faut alors définir les arguments **lat0**, **lon0**, **lat1**, **lon1**. L'argument **FUN** permet le choix entre le dessin de flèches (**arrows**, par défaut) ou de segments (**segments**).

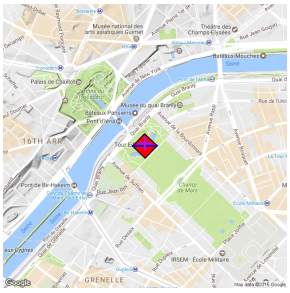
```
PlotArrowsOnStaticMap(eiffel_rgm1, FUN = arrows,
                      lat0 = 48.858248, lon0 = 2.293310,
                      lat1 = 48.858273, lon1 = 2.295698,
                      col = "blue", lwd = 3)
```

Pour dessiner des polygones, on peut également utiliser la fonction **PlotPolysOnStaticMap()**. On peut ainsi utiliser un objet de classe **SpatialPolygons** (mais pas de classe **SpatialPolygonsDataFrame**), ou bien un objet de classe **PolySet** (§ 1.3.2). Par défaut, cette fonction possède un argument **add = FALSE**, qui permet l'ajout à une carte déjà existante.

```
PlotPolysOnStaticMap(eiffel_rgm1, polys = eiffel_po_W84sp, col = "red", add = FALSE)
```

Avec toutes ces fonctions, on peut librement superposer des couches en définissant l'argument **add = TRUE**.

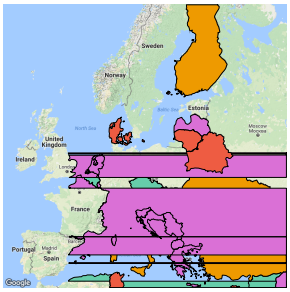
```
PlotPolysOnStaticMap(eiffel_rgm1, polys = eiffel_po_W84sp, col = "red", add = FALSE)
PlotOnStaticMap(eiffel_rgm1, FUN = lines,
                lat = c(48.858248, 48.858273),
                lon = c( 2.293310,  2.295698),
                col = "blue", lwd = 3, add = TRUE)
PlotOnStaticMap(eiffel_rgm1, FUN = points,
                lat = 48.858269, lon = 2.294511,
                col = "purple", pch = "+", cex = 3, add = TRUE)
```



7.1.2.4 Carte typologique

On peut réaliser des cartes typologiques à l'aide de la fonction **PlotPolysOnStaticMap()**, dans laquelle on passe un vecteur de couleurs, correspondant à une variable qualitative) à l'argument **col** (une couleur par entité dessinée). Ici, on peut constater que la fonction gère mal le fait que l'objet de classe **SpatialPolygons** dessiné présente des éléments **Polygons** formés de plusieurs **Polygon** et non d'un seul (§ 1.1.4), comme c'est par exemple le cas de la France avec un polygone pour la partie continentale et un autre pour la Corse. Pour que la représentation soit correcte, il faudrait donc recréer un objet **SpatialPolygons** où chaque élément **Polygons** ne contiendrait qu'un seul **Polygon**.

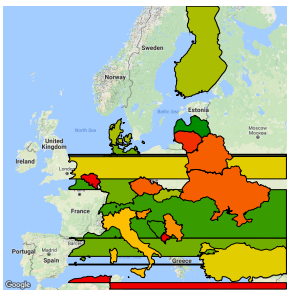
```
pal_col <- c("orchid", "aquamarine3", "orange2", "tomato2")
col_vec <- as.character(factor(EU_ctr1_LAEA@data$part, labels = pal_col))
EU_rgm <- GetMap(center = "Europe", zoom = 4, maptype = "terrain")
PlotPolysOnStaticMap(EU_rgm, polys = SpatialPolygons(EU_ctr1_W84@polygons),
                     col = col_vec, add = FALSE)
```



7.1.2.5 Carte choroplèthe

C'est le même principe, mais cette fois l'argument **col** de la fonction **PlotPolysOnStaticMap()** contient un vecteurs de couleurs correspondant à la discrétisation d'une variable quantitative.

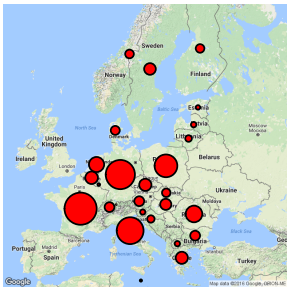
```
pal_nb <- 10
pal_col <- colorRampPalette(c("red2", "gold", "green4"))(pal_nb)
birth_rate <- EU_ctr1_LAEA@data$birth_rate
col_vec <- as.character(cut(birth_rate, include.lowest = TRUE,
                            breaks = quantile(birth_rate, probs = seq(0, 1, length.out = pal_nb+1),
                                              na.rm = TRUE),
                            labels =  pal_col))

EU_rgm <- GetMap(center = "Europe", zoom = 4, maptype = "terrain")
PlotPolysOnStaticMap(EU_rgm, polys = SpatialPolygons(EU_ctr1_W84@polygons),
                     col = col_vec, add = FALSE)
```



7.1.2.6 Carte en symboles proportionnels

Pour représenter une carte en symbole proportionnels, on utilise la fonction **PlotOnStaticMap()** avec l'argument **FUN = points**. Comme on ne peut pas utiliser d'objets de classe **SpatialPointsDataFrame**, il faut récupérer les coordonnées des centroïdes avec la fonction **coordinates()**. Pour gérer le type de symboles, leurs tailles et leurs couleurs, on appelle les arguments habituels de la fonction **points()** du package **graphics** (R Core Team, 2016), comme vu précédemment (§ 6.1.3.1).

```
EU_rgm <- GetMap(center = "Europe", zoom = 4, maptype = "terrain")
PlotOnStaticMap(EU_rgm, FUN = points,
                lat = coordinates(EU_labpt_W84)[, 2],
                lon = coordinates(EU_labpt_W84)[, 1],
                cex = sqrt(EU_labpt_W84@data$birth) / 120,
                pch = 21, col = "black", bg  = "red", lwd = 3)
```



7.1.2.7 Assembler des cartes

Pour assembler des cartes dans une même fenêtre graphique, il n'y pas de difficulté particulière ; il faut utiliser les mêmes méthodes que pour la syntaxe de type *Painter's Model* (§ 6.1).

7.1.3 Package dismo

Comme déjà mentionné (§ 4.8), le package **dismo** (Hijmans *et al.*, 2014) fournit des fonctions pour la modélisation de la distribution des espèces. Par ailleurs, il permet de réaliser des représentations sur des fonds de cartes statiques.

```
library(dismo)
```

7.1.3.1 Types de fonds de cartes

Seuls les fonds de cartes Google Maps sont disponibles :

- **roadmap** : carte des rues ;
- **satellite** : image satellite ;
- **terrain** : carte avec des caractéristiques physiques comme le terrain et la végétation ;
- **hybrid** : couche transparente de la carte des rues sur les images satellites.

7.1.3.2 Définition du fond de carte

La carte est définie par l'utilisation de la fonction **gmap()**. En entrée, elle admet des objets de classes :

- **matrix** ;
- **data.frame** (avec colonnes nommées x et y) ;
- **sp** (vectoriels ou matriciels) ;
- **raster** (dont les objets **extent**).

Ci-dessous, on définit l'emprise de la carte à partir d'un objet de classe **extent** issu des coordonnées obtenues par *geocoding*.

```
(eiffel_geo_W84 <- geocode("paris tour eiffel"))
##       originalPlace                                                    interpretedPlace
## 1 paris tour eiffel Eiffel Tower, Champ de Mars, 5 Avenue Anatole France, 75007 Paris, France
##   longitude latitude     xmin    xmax     ymin     ymax uncertainty
## 1  2.294481 48.85837 2.293132 2.29583 48.85702 48.85972         180
eiffel_ext_W84 <- extent(unlist(eiffel_geo_W84[, c("xmin", "xmax", "ymin", "ymax")]))
```

On pourrait également utiliser la matrice de coordonnées suivante :

```
eiffel_ext_W84 <- rbind(c(2.285242, 2.302507), c(48.853278, 48.863453))
```

On utilise alors la fonction **gmap()**, qui permet de récupérer un fond de carte au format **RasterLayer**.

```
eiffel_rgm_MCT <- gmap(eiffel_ext_W84, type = "roadmap")
class(eiffel_rgm_MCT)
## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"
```

La carte est dessinée à l'aide de la fonction **plot()**.

```
plot(eiffel_rgm_MCT)
```

7.1.3.3 Gestion du système de coordonnées

Par défaut, la carte est projetée en Mercator (code EPSG 3857) ; l'argument **lonlat** de la fonction **gmap()** vaut alors **FALSE**. S'il vaut **TRUE**, on peut travailler en WGS 84 (code EPSG 4326). Si l'on souhaite superposer des couches, il faut alors qu'elles soient exprimées dans le même système de coordonnées que celui imposé par **gmap()**. La fonction **Mercator()** permet de convertir des objets de classes **sp** ou **matrix** depuis le WGS 84 vers le Mercator. L'argument **inverse** permet de faire la conversion dans le sens inverse.

```
(eiffel_pt_MCT <- Mercator(eiffel_pt_W84sp))
##             x       y
## [1,] 255423.8 6250847
```

7.1.3.4 Affichage des entités spatiales

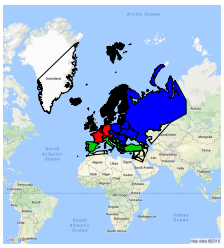
Avec ce package, on peut superposer des objets de base au fond de carte (i.e. **vector**, **matrix**, **data.frame**, etc.), grâce aux fonctions du package **graphics** (R Core Team, 2016). Pour superposer des objets spatiaux de classe **sp** ou **raster**, il faut utiliser la fonction **plot()** en utilisant l'argument **add = TRUE**.

```
plot(eiffel_rgm_MCT)
plot(eiffel_po_MCTsp, col = "red", add = TRUE)
plot(eiffel_li_MCTsp, col = "blue", lwd = 2, add = TRUE)
plot(eiffel_pt_MCTsp, col = "purple", cex = 2, add = TRUE)
```

7.1.3.5 Produire différents types de cartes

Comme il est possible de superposer aisément des objets spatiaux de classes **sp** ou **raster** à l'aide de la fonction **plot()**, pour dessiner des cartes typologiques, des cartes choroplèthes, des cartes en symboles proportionnels, etc., il faut appliquer les même méthodes explicitées précédemment (§ 6.1.1.1 et 6.1.2.1).

```
EU_ext_MCT <- as.vector(extent(EU_ctr1_W84))
dsm_map <- gmap(EU_ext_MCT, type = "roadmap")
plot(dsm_map)
plot(EU_ctr1_MCT, col = EU_ctr1_LAEA@data$part, add = TRUE)
```



7.1.3.6 Assembler des cartes

Pour assembler plusieurs cartes dans une même fenêtre, il faut utiliser les mêmes méthodes que pour la syntaxe de type *Painter's Model* (§ 6.1).

7.1.4 Package ggmap

Le package **ggmap** (Kahle & Wickham, 2013) fournit des fonctions pour la visualisation spatiale avec Google Maps et OpenStreetMap. Il utilise la syntaxe de type *Grammar of Graphics* telle qu'implémentée dans le package **ggplot2** (Wickham, 2009), dont il dépend.

```
library(ggmap)
```

7.1.4.1 Types de fonds de cartes

Ce package permet notamment la prise en charge des serveurs suivants :

- CloudMade (nécessite une clé d'identification) ;
- Google Maps Hybrid ;
- Google Maps Roadmap ;
- Google Maps Satellite ;
- Google Maps Terrain ;
- Naver Map ;
- OpenStreetMap ;
- Stamen Terrain-labels ;
- Stamen Terrain-lines ;
- Stamen Toner ;
- Stamen Toner 2010 ;
- Stamen Toner 2011 ;
- Stamen Toner Background ;
- Stamen Toner Hybrid ;
- Stamen Toner Labels ;
- Stamen Toner Lines ;
- Stamen Toner Lite ;
- Stamen Watercolor.

7.1.4.2 Définition du fond de carte

On peut importer les fonds de cartes avec la fonction **get_map()**. La carte se contruit *via* l'utilisation de l'argument **location** , qui peut être défini à partir :

- d'une requête ;
- du couple de coordonnées (dans l'ordre longitude puis latitude) du point central ;
- des coordonnées (dans l'ordre longitude puis latitude) de l'emprise.

Le niveau de zoom (**zoom**) est facultatif. Le type de fond de carte est, quant à lui, défini grâce à l'argument **maptype**.

Notez que, selon le serveur désiré, il est également possible d'utiliser directement les fonctions suivantes : **get_cloudmademap()**, **get_googlemap()**, **get_navermap()**, **get_openstreetmap()**, **get_stamenmap()**. En réalité, **get_map()** n'est qu'une fonction wrapper faisant appel à ces fonctions.

```
eiffel_ggm <- get_map(location = "Tour Eiffel, Paris",
                      zoom = 15, source = "google", maptype = "terrain")
eiffel_ggm <- get_map(location = c(2.294511, 48.858269),
                      zoom = 15, source = "google", maptype = "terrain")
eiffel_ggm <- get_map(location = c(2.285242, 48.863453, 2.302507, 48.853278),
                      zoom = 15, source = "google", maptype = "terrain")
```

L'objet renvoyé par la fonction **get_map()**, de double classe **ggmap** et **raster**, est une liste structurée contenant une matrice des couleurs du fonds de cartes exprimées en système hexadécimal.

```
class(eiffel_ggm)
## [1] "ggmap"  "raster"
str(eiffel_ggm)
```

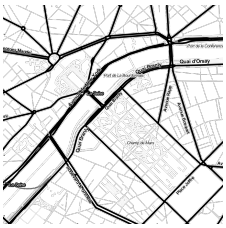
```
##  chr [1:1280, 1:1280] "#FEFEFE" "#FCFCFC" "#F6F6F6" "#F6F6F6" "#F6F6F6" ...
##  - attr(*, "class")= chr [1:2] "ggmap" "raster"
##  - attr(*, "bb")='data.frame': 1 obs. of  4 variables:
##   ..$ ll.lat: num 48.8
##   ..$ ll.lon: num 2.28
##   ..$ ur.lat: num 48.9
##   ..$ ur.lon: num 2.31
##  - attr(*, "source")= chr "google"
##  - attr(*, "maptype")= chr "terrain"
##  - attr(*, "zoom")= num 15
```

Pour dessiner la carte, on utilise la fonction **ggmap()**.

```
ggmap(eiffel_ggm)
```

Il est possible d'effectuer simultanément la requête de localisation et le tracé de la carte, et ce grâce à la fonction **qmap()** (pour *quick map plot*). Il s'agit là de l'équivalent de la fonction **qplot()** du package **ggplot2** (§ 6.3.1.1).

```
qmap(location = "Tour Eiffel, Paris",
     zoom = 15, source = "google", maptype = "toner-hybrid")
```



7.1.4.3 Affichage des entités spatiales

La syntaxe a adopter est celle du package **ggplot2** (Wickham, 2009), telle que décrite précédemment (§ 6.3.1). Nous ne la détaillerons donc pas à nouveau dans cette partie. Pour les différentes entités, les coordonnées peuvent êtres définies dans les arguments **x** et **y** de la fonction **aes**. Si l'on dispose d'objets spatiaux de classes **sp** ou **raster**, il faut alors convertir les objets en **data.frame** à l'aide des fonctions **base::as.data.frame()** (pour les objets **SpatialPoints(DataFrame)**), **ggplot2::fortify()** (pour les objets **SpatialLinesDataFrame** et les objets **SpatialPolygonsDataFrame**), ou **raster::rasterToPoints()** et **base::as.data.frame()** (pour les objets **RasterLayer**) (§ 6.3.1.1).

Pour dessiner les entités spatiales, on utilise les fonctions du package **ggplot2** vues précédemment (§ 6.3.1.1) :

- points : **geom_point()** ;
- lignes : **geom_path()** ou **geom_segment()** ;
- polygones : **geom_polygon()**.

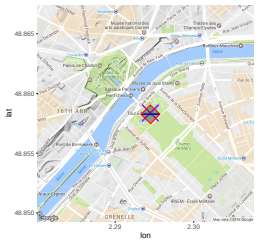
```
ggmap(eiffel_ggm) +
  geom_point(aes(x = 2.294481, y = 48.85837),
             size = 10, colour = "red", shape = 4)
```

Notez que si l'on utilise la fonction **geom_segment()**, l'argument **arrow** permet de dessiner une flèche par appel à la fonction **arrow()** du package **grid** (R Core Team, 2016).

```
ggmap(eiffel_ggm) +
  geom_segment(aes(x    = 2.294470, y    = 48.859040,
                   xend = 2.294523, yend = 48.857474), colour = "blue", arrow = grid::arrow())
```

Pour superposer des couches, il n'y a pas de difficulté particulière. On procède comme à l'accoutumé.

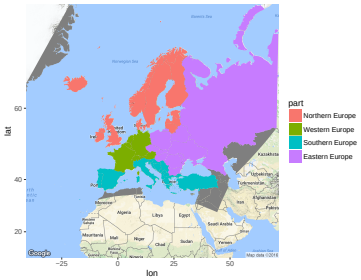
```
ggmap(eiffel_ggm) +
  geom_point(aes(x = 2.294481, y = 48.85837),
             size = 10, colour = "purple", shape = 4) +
  geom_segment(aes(x    = 2.293310, y    = 48.858248,
                   xend = 2.295698, yend = 48.858273), colour = "blue") +
  geom_polygon(aes(x = x, y = y), data = eiffel_po_W84,
               colour = "red", alpha = 0.4)
```



7.1.4.4 Produire différents types de cartes

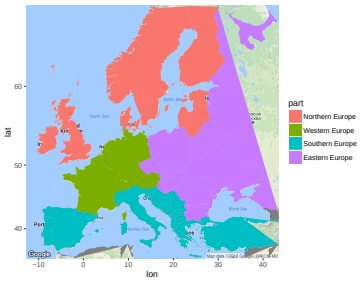
Pour dessiner des cartes typologiques, des cartes choroplèthes, des cartes en symboles proportionnels, etc. Pour cela, l faut appliquer les même méthodes explicitées précédemment (§ 6.3.1.4, 6.3.1.5 et 6.3.1.6).

```
EU_ggm <- get_map(location = "Europe", zoom = 3)
ggmap(EU_ggm) +
  geom_polygon(data = EU_ctr1_W84_df, mapping = aes(x = long, y = lat, group = group, fill = part))
```



Attention, selon le niveau de zoom choisi, il y a parfois des erreurs d'affichage. Ici, on peut constater que les polygones qui touchent le bord de la carte sont mal affichés.

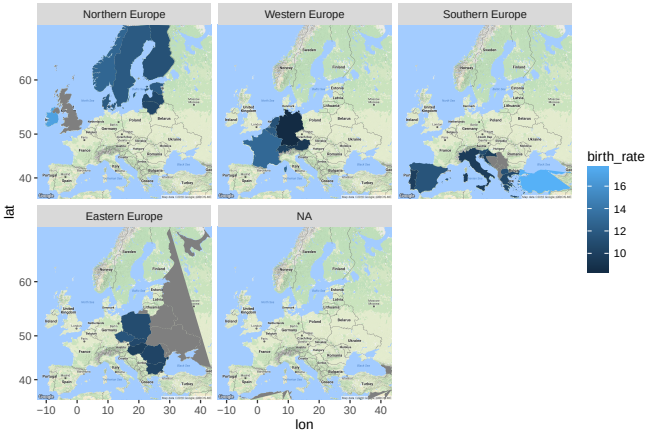
```
EU_ggm <- get_map(location = "Europe", zoom = 4)
ggmap(EU_ggm) +
  geom_polygon(data = EU_ctr1_W84_df, mapping = aes(x = long, y = lat, group = group, fill = part))
```



7.1.4.5 Faceting

Le *faceting* est compatible avec le package **ggmap**. Il n'y a pas de difficulté particulière. On procède comme d'usage avec le package **ggplot2** (Wickham, 2009) (§ 6.3.1.7).

```
ggmap(EU_ggm) +
  geom_polygon(data = EU_ctr1_W84_df, mapping = aes(x = long, y = lat, group = group, fill = birth_rate)) +
  facet_wrap(~ part)
```



7.1.4.6 Assembler des cartes

Il faut procéder de la même manière qu'avec le package **ggplot2** (Wickham, 2009) (§ 6.3.1.10).

7.1.4.7 Requêtes géographiques

Il est possible de faire une requête sur la localisation à l'aide de la fonction **geocode()**. Pour extraire l'ensemble des résultats, il faut utiliser l'argument **output = "all"**.

```
geocode(location = "Tour Eiffel, Paris")
##        lon      lat
## 1 2.294481 48.85837
geocode(location = "Tour Eiffel, Paris", output = "latlona")
##        lon      lat                                                                    address
## 1 2.294481 48.85837 eiffel tower, champ de mars, 5 avenue anatole france, 75007 paris, france
str(geocode(location = "Tour Eiffel, Paris", output = "all"),  max.level = 2)
## List of 2
##  $ results:List of 1
##   ..$ :List of 5
##  $ status : chr "OK"
```

L'inverse est également possible. On peut ainsi fournir un couple de coordonnées à la fonction **revgeocode()**, qui permet de récupérer alors une adresse.

```
revgeocode(c(x = 2.294481, y = 48.85837))
## [1] "5 Avenue Anatole France, 75007 Paris, France"
```

On peut faire une requête de calcul de distance avec la fonction **mapdist()**. Pour extraire l'ensemble des résultats, il faut utiliser l'argument **output = "all"**.

```
mapdist(from = "Tour Eiffel, Paris", to = "place de l Etoile",
        mode = "walking")
##                 from                to km miles minutes hours
## 1 Tour Eiffel, Paris place de l Etoile NA    NA      NA    NA
```

On peut réaliser une requête de calcul de trajet avec la fonction **route()**. Ici aussi, l'argument **output = "all"** permet d'extraire l'ensemble des résultats.

```
route1 <- route(from = "Tour Eiffel, Paris", to = "Place de l Etoile, Paris",
                mode = "walking", structure = "route")
head(route1, 2)
##     m    km     miles seconds minutes       hours leg      lon      lat
## 1   8 0.008 0.0049712       6     0.1 0.001666667   1 2.293407 48.85897
## 2 225 0.225 0.1398150     204     3.4 0.056666667   2 2.293482 48.85902
```

7.1.5 Package osmar

Le package **osmar** (Eugster & Schlesinger, 2010) permet d'extraire des caractéristiques physiques du terrain d'OpenStreetMap, de les manipuler dans R, et de les convertir dans d'autres formats afin de pouvoir les manipuler avec d'autres packages (e.g. **sp** ou **igraph**).

```
library(osmar)
```

7.1.5.1 Établissement de la connexion

Il est possible de réaliser plusieurs types de connexions vers des sources de données, selon que l'on souhaite la récupérer depuis un fichier ou depuis un serveur.

osmsource_file() permet l'import complet d'un fichier au format OSM (extension ".osm").

```
osmsource_file("MyFile.osm")
```

osmsource_osmosis() permet l'import partiel (selon une emprise) d'un fichier au format OSM au moyen d'Osmosis [1], une application Java, en ligne de commande, permettant le traitement des données OpenStreetMap.

```
osmsource_osmosis("MyFile.osm")
```

osmsource_api() permet l'import partiel (selon une emprise) depuis l'API d'OpenStreetMap.

```
osmsource_api()
## $url
## [1] "http://api.openstreetmap.org/api/0.6/"
##
## attr(,"class")
## [1] "api"       "osmsource"
```

7.1.5.2 Définition du fond de carte

On définit tout d'abord l'emprise de la carte en donnant les coordonnées, exprimées en WGS 84 (code EPSG 4326), de son centre ou des coins au moyen des fonctions **center_bbox()** ou **corner_bbox()**.

```
eiffel_osr_bb <- corner_bbox(left  = 2.285242, bottom = 48.853278,
                             right = 2.302507, top    = 48.863453)
```

On importe les données avec la fonction **get_osm()**, dans laquelle on définit la source des données (ici l'API).

```
eiffel_osr <- get_osm(eiffel_osr_bb, source = osmsource_api())
```

L'objet est une liste de classe **osmar** sur laquelle on peut faire des requêtes, afin d'extraire des entités.

```
class(eiffel_osr)
## [1] "osmar" "list"
```

Les objets de classe **osmar** se présentent sous la forme d'une liste de trois éléments de classe **data.frame**. Le premier correspond aux attributs et aux étiquettes des nœuds, le second aux attributs, tags, et références des chemins, le troisième aux attributs, tags, et références des relations.

```
eiffel_osr
## osmar object
## 25412 nodes, 2076 ways, 246 relations
names(eiffel_osr)
## [1] "nodes"     "ways"      "relations"
```

Voici la structure des données pour l'exemple des nœuds :

```
str(eiffel_osr$nodes)
## List of 2
##  $ attrs:'data.frame': 25412 obs. of  9 variables:
##   ..$ id       : num [1:25412] 368274 368288 368286 368305 470180 ...
##   ..$ visible  : Factor w/ 1 level "true": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ timestamp: POSIXlt[1:25412], format: "2011-03-01 23:16:24" "2015-03-27 21:47:00" "2011-03-28 22:08:39" ...
##   ..$ version  : num [1:25412] 5 12 3 6 3 14 9 8 16 18 ...
##   ..$ changeset: num [1:25412] 7434166 29790266 7702155 5907812 3973646 ...
##   ..$ user     : chr [1:25412] "Pieren" "mapper999" "Pieren" "Pieren" ...
##   ..$ uid      : Factor w/ 100 levels "1044414","1072240",..: 27 23 27 27 63 63 27 27 27 27 ...
##   ..$ lat      : num [1:25412] 48.9 48.9 48.9 48.9 48.9 ...
##   ..$ lon      : num [1:25412] 2.3 2.3 2.3 2.29 2.3 ...
##  $ tags :'data.frame': 31959 obs. of  3 variables:
##   ..$ id: num [1:31959] 368274 368288 368288 368288 470188 ...
##   ..$ k : Factor w/ 162 levels "addr:city","addr:country",..: 59 32 33 59 59 59 59 59 1 2 ...
##   ..$ v : Factor w/ 3642 levels "-1","-2","#706550",..: 3599 3610 3641 3076 3599 3599 3599 3599 3351 3141 ...
##  - attr(*, "class")= chr [1:3] "nodes" "osmar_element" "list"
```

1. Page Wiki d'Osmosis : https://wiki.openstreetmap.org/wiki/Osmosis.

7.1.5.3 Requêtes sur les métadonnées

Il est possible d'extraire des caractéristiques physiques de terrain [2] en réalisant des requêtes sur les métadonnées contenues dans les objets **osmar**. Ici, on extrait les entités correspondant aux bâtiments présents dans l'emprise de la carte, et ce à l'aide des fonctions **find()** et **find_down()**.

```
BDG_ids <- find(eiffel_osr, way(tags(k == "building")))
BDG_ids <- find_down(eiffel_osr, way(BDG_ids))
BDG <- subset(eiffel_osr, ids = BDG_ids)
```

Un fois la requête effectuée, l'objet récupéré reste de classe **osmar**, mais de dimensions plus réduites que celui de départ, bien évidemment.

```
BDG
## osmar object
## 14166 nodes, 928 ways, 0 relations
```

De la même manière, on peut extraire les bassins :

```
BAS_ids <- find(eiffel_osr, way(tags(grepl("water", v))))
BAS_ids <- find_down(eiffel_osr, way(BAS_ids))
BAS <- subset(eiffel_osr, ids = BAS_ids)
```

On peut faire de même avec les surfaces enherbées :

```
HRB_ids <- find(eiffel_osr, way(tags(k %in% c("leisure"))))
HRB_ids <- find_down(eiffel_osr, way(HRB_ids))
HRB <- subset(eiffel_osr, ids = HRB_ids)
```

Ou encore avec les rues secondaires :

```
RUE_ids <- find(eiffel_osr, way(tags(v %in% c("residential", "tertiary",
                                              "pedestrian","footway", "secondary"))))
RUE_ids <- find_down(eiffel_osr, way(RUE_ids))
RUE <- subset(eiffel_osr, ids = RUE_ids)
```

7.1.5.4 Affichage des entités spatiales

Pour pouvoir dessiner les objets de classe **osmar** qui viennent d'être extraits, il faut préalablement les convertir en objets de classe **sp**. Pour ce faire, il faut utiliser la fonction **as_sp()**. Cette dernière permet de gérer les différents types d'entités vectorielles (points, lignes et polygones).

```
BDG_sp <- as_sp(BDG, what = "polygons")
BAS_sp <- as_sp(BAS, what = "polygons")
HRB_sp <- as_sp(HRB, what = "polygons")
RUE_sp <- as_sp(RUE, what = "lines")
class(BDG_sp)
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
class(RUE_sp)
## [1] "SpatialLinesDataFrame"
## attr(,"package")
## [1] "sp"
```

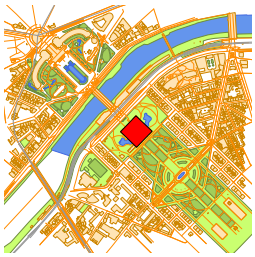
On peut bien évidemment dessiner un objet **sp** en utilisant la fonction **plot()**.

```
plot(HRB_sp, col = "darkolivegreen1", border = "darkolivegreen4")
```

```
par(mfrow = c(1, 4))
plot(BDG_sp, col = "moccasin", border = "orange3")
plot(BAS_sp, col = "cornflowerblue", border = "royalblue")
plot(HRB_sp, col = "darkolivegreen1", border = "darkolivegreen4")
plot(RUE_sp, col = "darkorange", lwd = 1)
```

2. Description des caractéristiques physiques de terrain disponibles dans OpenStreetMap : http://wiki.openstreetmap.org/wiki/Map_Features.

La carte étant produite en WGS 84 (code EPSG 4326), pour superposer des objets, il suffit qu'ils soient exprimés dans ce système de coordonnées (il convient alors d'utiliser l'argument **add = TRUE**). En superposant plusieurs couches à partir des extractions et l'emprise de la tour Eiffel, on peut obtenir la carte suivante :



7.1.5.5 Assembler des cartes

Pour assembler des cartes, il faut utiliser les mêmes méthodes que pour la syntaxe de type *Painter's Model* (§ 6.1).

7.1.6 Package osmplotr

Tout comme **osmar**(Eugster & Schlesinger, 2010) dont il dépend, le package **osmplotr** (Padgham, 2016) permet d'accéder aux données de caractéristiques physiques du terrain d'OpenStreetMap. dépendant également du package **ggplot2** (Wickham, 2009), il utilise une syntaxe de type *Grammar of Graphics*.

7.1.6.1 Définition du fond de carte

On définit tout d'abord l'emprise de la carte en donnant un vecteur de coordonnées, exprimées en WGS 84 (code EPSG 4326), de son centre ou des coins avec les fonctions **get_bbox()** ou **corner_bbox()**.

```
eiffel_opr_bb <- get_bbox(c(xmin = 2.285242, ymin = 48.853278, xmax = 2.302507, ymax = 48.863453))
```

Le fond de carte est défini à l'aide de la fonction **osm_basemap()**, qui renvoie un objet de double classe **gg** et **ggplot**.

```
eiffel_opr <- osm_basemap(bbox = eiffel_opr_bb, bg = "white")
class(eiffel_opr)
## [1] "gg"     "ggplot"
```

7.1.6.2 Requêtes sur les métadonnées

On peut effectuer des requêtes sur les métadonnées avec la fonction **extract_osm_objects()**. On choisit la caractéristique physique du terrain[3] qui nous intéresse dans l'argument **key**, l'emprise sur laquelle réaliser la requête avec l'argument **bbox**, et l'on peut choisir le type d'entité que l'on souhaite récupérer *via* l'argument **return_type**.

```
BAS_sp <- extract_osm_objects(key = "natural", bbox = eiffel_opr_bb, return_type = "polygon")
class(BAS_sp)

## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
```

7.1.6.3 Gestion du système de coordonnées de la carte

Les objets **sp** récupérés sont en WGS 84 (code EPSG 4326) ; la carte produite à partir de ces données sera donc dessinée dans ce système de coordonnées. Si l'on souhaite dessiner une carte dans système projeté (ce qui est préférable), il faudra préalablement convertir les objets **sp** (§ 3.1).

7.1.6.4 Affichage des entités spatiales

Pour afficher la carte, les objets préalablement récupérés étant de classe **sp**, on peut utiliser la fonction **plot()**. Le package propose également ses propres fonctions. Ainsi la fonction **add_osm_objects()** permet-elle de superposer des couches les unes aux autres (autant d'appel que de couches à dessiner) avant de dessiner la carte à l'aide de la fonction **print_osm_map()**.

```
eiffel_opr <- add_osm_objects(eiffel_opr, BAS_sp, col = "cornflowerblue", border = "royalblue")
class(eiffel_opr)
```

3. Description des caractéristiques physiques de terrain disponibles dans OpenStreetMap : http://wiki.openstreetmap.org/wiki/Map_Features.

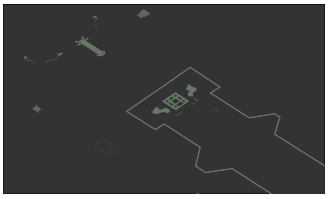
```
## [1] "gg"     "ggplot"
print_osm_map(eiffel_opr)
```

Il est possible de dessiner simultanément plusieurs couches, mais la procédure est un peu différente. Dans ce cas, on commence par définir les caractéristiques physiques qui nous intéressent et l'on fournit le vecteur ainsi créé à la fonction **osm_structures()**. À cette étape, on définit également un style de représentation *via* l'argument **col_scheme**. On récupère un **data.frame** définissant notamment les arguments **key** et **value** vus précédemment, ainsi qu'une couleur automatiquement associée.

```
structures <- c("building", "highway", "leisure", "park", "natural")
eiffel_opr2_str <- osm_structures(structures = structures, col_scheme = "dark")
head(eiffel_opr2_str)
```

Ensuite, il faut utiliser la fonction **make_osm_map()** afin de construire la carte à partir d'une emprise et du **data.frame** que l'on vient de récupérer. Il ne reste alors plus qu'à dessiner la carte avec la fonction **print_osm_map()**.

```
eiffel_opr2 <- make_osm_map(bbox = eiffel_opr_bb, structures = eiffel_opr2_str)
print_osm_map(eiffel_opr2$map)
```



7.2 Cartes dynamiques

7.2.1 Package plotGoogleMaps

Le package **plotGoogleMaps** (Kilibarda & Bajat, 2012) produit des cartes interactives dans les navigateurs Web et permet l'utilisation des couches Google Maps.

```
library(plotGoogleMaps)
```

7.2.1.1 Types de fonds de cartes

Le package **plotGoogleMaps** ne prend en compte que le fonds de cartes du serveur Google Maps :
- **roadmap** : carte des rues ;
- **satellite** : image satellite ;
- **terrain** : carte avec des caractéristiques physiques comme le terrain et la végétation ;
- **hybrid** : couche transparente de la carte des rues sur les images satellites.

7.2.1.2 Gestion du système de coordonnées

Ce package permet d'utiliser des couches définies dans n'importe quel système de coordonnées. Dans tous les cas, l'affichage se fera toujours en Mercator (code EPSG 3857).

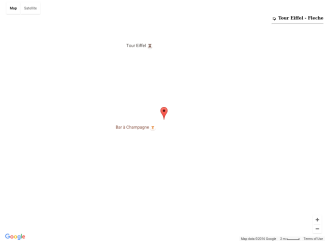
7.2.1.3 Affichage de la carte

Pour dessiner le fond de carte, il faut faire appel à la fonction **plotGoogleMaps()**. On lui fournit directement un objet de classe **sp** (y compris les objets matriciels). En effet, le package **plotGoogleMaps** travaille uniquement avec des données aux formats **sp** ; les objets de classes **raster** ne sont pas pris en compte. Il est capable de gérer les format suivants :
- **SpatialPoints(DataFrame)** : points ;
- **SpatialLines(DataFrame)** : lignes ;
- **SpatialPolygons(DataFrame)** : polygones ;

- **SpatialPixels(DataFrame)** : pixels formant une grille rectangulaire possiblement incomplète ;
- **GridTopology(DataFrame)** : grille rectangulaire vide ;
- **SpatialGrid(DataFrame)** : grille rectangulaire pleine.

De très nombreux arguments sont disponibles. Ils permettent de gérer le type de fond de carte (**mapTypeId**), le choix de la variable de la table attributaire à afficher (**zcol**), la palette de couleurs des entités relative à cette variable (**colPalette**, **strokeColor**, **strokeOpacity**...), le symbole de l'icône (**iconMarker**), le niveau de zoom (**zoom**), le nom de la couche (**layerName**), les options de navigation (**navigationControlOptions**, **streetViewControl**...), etc. La fonction **plotGoogleMaps()**, exporte obligatoirement la carte dans un fichier aux format HTM ou HTML. Si ce dernier n'est pas défini (**filename = ""**), le fichier sera automatiquement créé dans un répertoire temporaire. Pour autant, on peut assigner le résultat dans un objet R, sous la forme d'une liste.

```
eiffel_pgm <- plotGoogleMaps(SP = eiffel_pt_W84sp, layerName = "Tour Eiffel - Fleche",
                             mapTypeId = "TERRAIN",
                             filename = "eiffel_pgm1.html", openMap = FALSE)
str(eiffel_pgm)
## List of 4
##  $ starthtm : chr "<!DOCTYPE html> \n <html> \n <head> \n <meta name=\"viewport\" content=\"initial-scale=1.0, u
##  $ var      : chr " var marker \n var  map  \n var  markerseiffelxptxW84sp1259 =[] ; var  marker = new google.ma
##  $ functions: chr "function showR(R,boxname, map) {\n  R.setMap(map);\n  document.getElementById(boxname).checke
##  $ endhtm   : chr "</script> \n </head> \n <body onload=\"initialize()\"> \n \n                                <div
```



Pour superposer des couches, on fait un premier appel à la fonction **plotGoogleMaps()** en définissant l'argument **add = TRUE**, en choisissant le type de fond de carte, mais sans définir de fichier de sortie. Puis, on fait un second appel en définissant l'argument **previousMap** égal au premier appel, et en faisant un export dans un fichier *via* l'argument **filename**.

```
eiffel_pgm_base <- plotGoogleMaps(eiffel_po_W84sp, layerName = "Tour Eiffel - Base",
                                  mapTypeId = "TERRAIN", add = TRUE)
## [1] "using original PolyCol"
eiffel_pgm_cent <- plotGoogleMaps(eiffel_pt_W84sp, layerName = "Tour Eiffel - Fleche",
                                  previousMap = eiffel_pgm_base,
                                  filename = "eiffel_pgm2.html", openMap = FALSE)
```

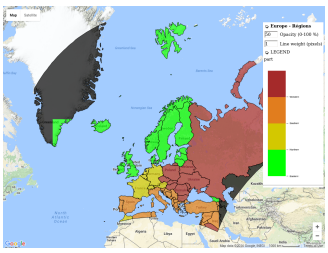
D'autres représentations graphiques sont disponibles grâces aux fonctions suivantes :

- **mcGoogleMaps()** : grappe de balises ;
- **bubbleGoogleMaps()** : trace des bulles ;
- **ellipseGoogleMaps()** : trace des ellipses ;
- **segmentGoogleMaps()** : graphe en secteurs ;
- **stplotGoogleMaps()** : pour les données spatio-temporelles (STDIF, STFDF) ;
- **stfdfGoogleMaps()** : pour les données spatio-temporelles (STFDF).

7.2.1.4 Carte typologique

Pour dessiner une carte typologique, c'est très simple, il suffit de fournir le nom de la variable d'intérêt de la table attributaire à l'argument **zcol** de la fonction **plotGoogleMaps()**.

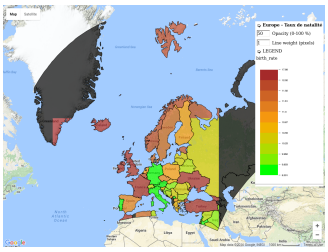
```
EU_pgm_part <- plotGoogleMaps(EU_ctr1_W84, zcol = "part", layerName = "Europe - Régions",
                              mapTypeId = "TERRAIN", filename = "EU_pgm_part.html", openMap = FALSE)
```



7.2.1.5 Carte choroplèthe

C'est le même principe, on utilise la même fonction et l'on fournit le nom de la variable d'intérêt de la table attributaire à l'argument **zcol**.

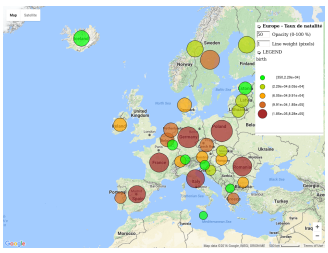
```
EU_pgm_birth_rate <- plotGoogleMaps(EU_ctr1_W84, zcol = "birth_rate", layerName = "Europe - Taux de natalité",
                                    mapTypeId = "TERRAIN", filename = "EU_pgm_birth_rate.html", openMap = FALSE)
```



7.2.1.6 Carte en symboles proportionnels

On peut utiliser la fonction **bubbleGoogleMaps()**, on donne également la variable d'intérêt de la table attributaire à l'argument **zcol** (les valeurs manquantes ne sont pas admises). On gère la taille maximale du cercle (en mètres) à l'aide de l'argument **max.radius**.

```
EU_pgm_birth <- bubbleGoogleMaps(EU_labpt_W84, zcol = "birth", max.radius = 2e5, layerName = "Europe - Taux de nata
                                 mapTypeId = "TERRAIN", filename = "EU_pgm_birth.html", openMap = FALSE)
## [1] "using original PolyCol"
```



7.2.2 Package googleVis

Le package **googleVis** (Gesmann & de Castillo, 2011) fournit une interface entre R et l'API Google Charts [4]. Il permet aux utilisateurs de créer des pages web avec des graphiques interactifs basés sur des trames de données de R. Les données restent au niveau local, sur l'ordinateur, et ne sont pas téléchargées sur Google.

```
library(googleVis)
```

7.2.2.1 Types de fonds de cartes

XXXX

7.2.2.2 Définition de l'emprise de la carte

L'emprise de la carte peut être définie de deux manières : à partir d'un nom de localisation, ou à partir d'un couple de coordonnées.

Si l'on opte pour le nom de localisation, il faut construire un **data.frame** dont la première colonne correspond à la localisation recherchée (adresse la plus complète possible), pour laquelle les coordonnées seront déterminées automatiquement par *geocoding*. La seconde colonne doit, quant à elle, contenir une chaîne de caractères qui sera associée à l'icône une fois la carte dessinée.

4. Site web de Google Charts : https://developers.google.com/chart/.

```
eiffel_pt_W84b <- data.frame(loc = "paris tour eiffel",
                             tip = "Tour Eiffel")
eiffel_pt_W84b
##                 loc         tip
## 1 paris tour eiffel Tour Eiffel
```

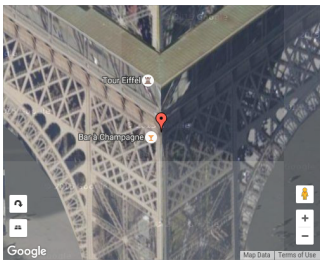
S l'on souhaite définir l'emprise de la carte à partir d'un couple de coordonnées, il faut également construire un **data.frame**, mais cette fois, la première colonne doit correspondre à une chaîne de caractères contenant la longitude et la latitude du centre de la carte, séparées par le caractère deux-points (“**:**”).

```
eiffel_pt_W84b <- paste(eiffel_pt_W84$y, eiffel_pt_W84$x, sep = ":")
eiffel_pt_W84b <- data.frame(loc = eiffel_pt_W84b,
                             tip = "Tour Eiffel", stringsAsFactors = FALSE)
eiffel_pt_W84b
##                  loc         tip
## 1 48.858269:2.294511 Tour Eiffel
```

7.2.2.3 Définition du fond de carte

On définit le fond de carte à l'aide de la fonction **gvisMap()**. XXXX

```
eiffel_ggv <- gvisMap(eiffel_pt_W84b,
                      locationvar = "loc",
                      tipvar      = "tip",
                      options     = list(showTip = TRUE))
class(eiffel_ggv)
## [1] "gvis" "list"

plot(eiffel_ggv)
```



L'argument **option** est une liste définie avec les éléments suivants :

- **gvis.editor** : modification en ligne ;
- **enableScrollWheel** : la barre de zoom ;
- **showTip** : l'étiquettes pour chaque point ;
- **showLine** : relie les points ;
- **lineColor** : la couleur des lignes ;
- **lineWidth** : l'épaisseur des lignes ;
- **mapType** : le type de fond de carte [“normal”, “terrain”, “satellite”, “hybrid”] ;
- **useMapTypeControl** : changement de type de fond de carte en ligne ;
- **zoomLevel** : le niveau de zoom ;
- **width** : la largeur de la carte [pixels] ;
- **height** : la hauteur de la carte [pixels].

D'autres représentations graphiques permettent de tracer des cartes à l'échelle du continent, du pays ou de la région pouvant être inclue dans une page web ou être autonome, grâces aux fonctions suivantes :

- **gvisGeoChart()** ;
- **gvisGeoMap()** ;
- **gvisIntensityMap()**.

7.2.2.4 Carte typologique

XXXX

7.2.2.5 Carte choroplèthe

XXXX

7.2.2.6 Carte en symboles proportionnels

XXXX

7.2.3 Package leaflet

Le package **leaflet** (Cheng & Xie, 2015) [5] fournit des fonctions produisant des cartes interactives dans le navigateur Web au moyen de JavaScript.

```
library(leaflet)
```

Comme ce package dépend du package **magrittr** (Bache & Wickham, 2014), il est possible d'utiliser l'opérateur pipe ("**%>%**") qui permet de passer un résultat d'une fonction à l'autre.

7.2.3.1 Types de fonds de cartes

Le package **leaflet** permet d'accéder à de nombreux fonds, parmi lesquels [6] :

- OpenStreetMap Mapnik ;
- OpenStreetMap BlackAndWhite ;
- OpenStreetMap DE ;
- OpenStreetMap France ;
- Thunderforest OpenCycleMap ;
- Thunderforest Transport ;
- Thunderforest TransportDark ;
- Stamen TonerLite ;
- Stamen Watercolor ;
- NASA GIBS ViirsEarthAtNight2012 ;
- etc.

Attention, tous les fonds de cartes ne sont pas disponibles pour l'ensemble des niveaux de zoom et l'ensemble des pays.

7.2.3.2 Gestion du système de coordonnées

Le package **leaflet** autorise de ne travailler qu'avec des objets dont les coordonnées sont exprimées en WGS 84 (code EPSG 4326). Les cartes, quant à elles, sont projetées en Mercator (code EPSG 3857).

7.2.3.3 Définition du fond de carte

Pour définir le fond de carte, on fournit la fonction **leaflet()** à la fonction **addTiles()**. L'ensemble des options d'affichage se définissent dans l'argument **options** et ce, grâce à la fonction **tileOptions()**.

```
eiffel_lfl <- leaflet() %>% addTiles()
eiffel_lfl
```

L'objet ainsi créé est de double classe **leaflet** et **htmlwidget**. On peut noter que le package **htmlwidgets** (Vaidyanathan *et al.*, 2016), dont le package **leaflet** dépend, permet de gérer les objets HTML.

```
class(eiffel_lfl)
## [1] "leaflet"    "htmlwidget"
```

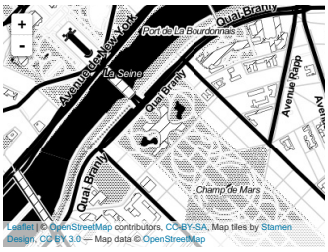
Pour gérer l'emprise de la carte, on peut utiliser les fonctions **setView()**, **fitBounds()** ou **setMaxBounds()**.

```
eiffel_lfl <- setView(map = eiffel_lfl, lng = 2.294511, lat = 48.858269, zoom = 15)
eiffel_lfl
```

5. Site web du package **leaflet** : https://rstudio.github.io/leaflet/.
6. Pour prévisualiser les fonds disponibles, on peut se rendre sur le site de la bibliothèque Leaflet : http://leaflet-extras.github.io/leaflet-providers/preview/.

Le choix du fond de carte est, quant à lui, géré par la fonction **addProviderTiles()**. Sinon, par défaut, c'est le fond de base d'OpenStreetMap qui est utilisé.

```
eiffel_lfl <- addProviderTiles(map = eiffel_lfl, provider = "Stamen.Toner")
eiffel_lfl
```

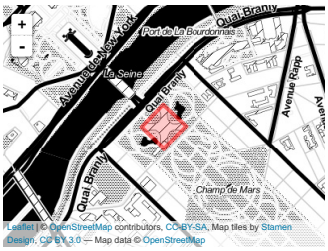


7.2.3.4 Affichage des entités spatiales

De manière générale, pour les différents types d'entités, on peut soit fournir directement les coordonnées aux arguments **lng** et **lat**, soit fournir un objet de classe **Spatial*(DataFrame)** à l'argument **data**.

Pour les entités ponctuelles, on peut superposer des icônes avec la fonction **addMarkers()**. En fonction du rendu souhaité, on peut également utiliser les fonctions **addPopups()**, **addCircles()** et **addCircleMarkers()**. On peut dessiner des polylignes avec la fonction **addPolylines()**. Pour tracer des polygones, il faut utiliser la fonction **addPolygons()**.

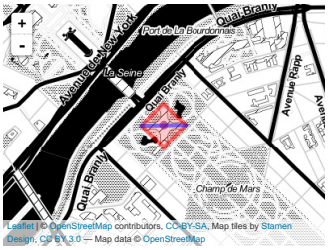
```
addPolygons(eiffel_lfl, data = eiffel_po_W84sp, color = "red")
```



Il est également possible de dessiner un rectangle avec la fonction **addRectangles()**, mais cette dernière ne gère pas les objets de classe **sp**. Cette fonction sert simplement à dessiner une emprise géographique.

Pour superposer des couches, il suffit de stocker au fur et à mesure les objets **leaflet** créés par les fonctions précédemment citées. Lorsqu'on dessine une couche secondaire, il suffit alors d'appeler l'objet précédemment créé.

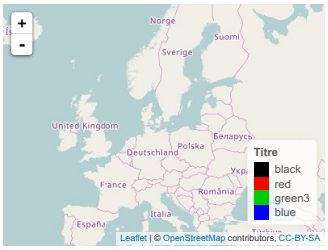
```
eiffel_lfl %>%
  addPolygons(data = eiffel_po_W84sp, color = "red") %>%
  addPolylines(data = eiffel_li_W84sp, color = "blue") %>%
  addCircles(data = eiffel_pt_W84sp, color = "purple")
```



7.2.3.5 Éléments contextuels

Légende. Il est possible de définir, la position (**position**), le titre (**title**). Malheureusement, il n'existe pas de manière simple de réaliser de légende pour les points.

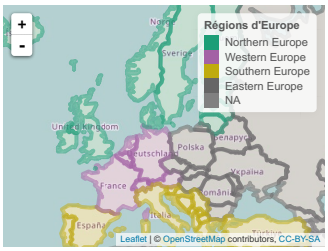
```
leaflet() %>% addTiles %>% setView(lng = 13, lat = 55, zoom = 3) %>%
  addLegend(position = "bottomright",
            colors   = rgb(t(col2rgb(palette())) / 255),
            labels   = palette(), opacity = 1,
            title    = "Titre")
```



7.2.3.6 Carte typologique

Pour réaliser une carte typologique, on dessine des polygones grâce à la fonction **addPolygons()**. La variable qualitative est passée à l'argument **color**, après avoir été convertie en vecteur de couleurs. Il existe plusieurs manières de gérer la palette de couleurs. On peut fournir directement un vecteur de couleurs où chacune d'entre-elles correspondent à une entité dessinée. Une autre solution consiste à passer par l'intermédiaire de la fonction **colorFactor()**, comme il s'agit là de données qualitatives. Notez que cette fonction peut utiliser les palettes de couleurs du package **RColorBrewer** (Neuwirth, 2014).

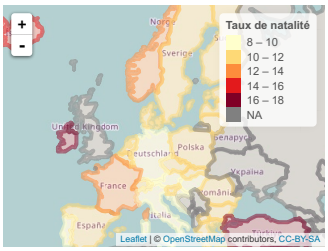
```
pal_f <- colorFactor(palette = "Dark2", domain = EU_ctr1_W84@data$part)
leaflet() %>% addTiles %>% setView(lng = 13, lat = 55, zoom = 3) %>%
  addPolygons(data = EU_ctr1_W84, color = ~pal_f(part)) %>%
  addLegend(position = "topright", title = "Régions d'Europe",
            pal = pal_f, values = EU_ctr1_W84@data$part, opacity = 1)
```



7.2.3.7 Carte choroplèthe

Pour dessiner une carte choroplèthe, c'est le même principe. La variable quantitative d'intérêt est convertie en vecteur de couleurs et passée à l'argument **color**. Ici aussi, on peut fournir directement son propre vecteur ou bien passer par l'intermédiaire d'une fonction. Dans le cas d'une variable quantitative, si elle est continue, on utilise la fonction **colorNumeric()**, si elle est discrète, on utilise les fonctions **colorBin()** ou **colorQuantile()**.

```
pal_q <- colorBin("YlOrRd", EU_ctr1_W84@data$birth_rate, bins = 5)
leaflet() %>% addTiles %>% setView(lng = 13, lat = 55, zoom = 3) %>%
  addPolygons(data = EU_ctr1_W84, color = ~pal_q(birth_rate)) %>%
  addLegend(position = "topright", title = "Taux de natalité",
            pal = pal_q, values = EU_ctr1_W84@data$birth_rate, opacity = 1)
```



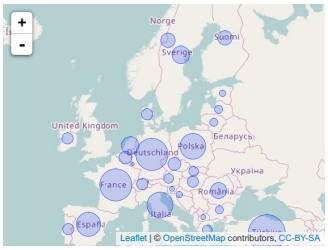
7.2.3.8 Carte en symboles proportionnels

Pour représenter une carte en symboles proportionnels, on utilise la fonction **addCircles()**. Dans notre exemple on souhaite représenter des disques aux positions des centroïdes des polygones. Si l'on dispose d'un objet de classe **SpatialPointsDataFrame** correspondant aux centroïdes, on utilisera simplement l'argument **data**.

```
leaflet() %>% addTiles %>% setView(lng = 13, lat = 55, zoom = 3) %>%
  addCircles(data = EU_ctr1_W84,
             lng = coordinates(EU_ctr1_W84)[, 1], lat = coordinates(EU_ctr1_W84)[, 2],
             weight = 1, radius = ~sqrt(birth)*300)
```

En revanche, si l'on ne dispose que d'un objet de classe **SpatialPolygonsDataFrame**, on pourra fournir les coordonnées aux arguments **lng** et **lat** (en utilisant, par exemple, la fonction **coordinates()**). L'objet **SpatialPolygonsDataFrame** devra, quant à lui, tout de même être passé à l'argument **data**.

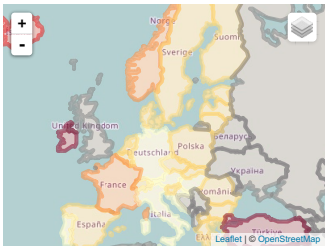
```
leaflet() %>% addTiles %>% setView(lng = 13, lat = 55, zoom = 3) %>%
  addCircles(data = EU_ctr1_W84,
             lng = coordinates(EU_ctr1_W84)[, 1], lat = coordinates(EU_ctr1_W84)[, 2],
             weight = 1, radius = ~sqrt(birth)*300)
```



7.2.3.9 Menu interactif d'affichage des couches

La fonction **addLayersControl()** permet d'afficher un menu interactif sur la carte. Ce menu permet à l'utilisateur de choisir le fond de carte et les couches géographiques à afficher. Pour ce faire, il faut définir des groupes. Pour le groupe des fonds de cartes, la manière la plus explicite est de faire des appels successifs à la fonction **addProviderTiles()**, où l'on définit le fond que l'on souhaite (**provider**) et le nom que l'on attribue au groupe (**group**). Les noms des groupes des couches spatiales doivent être définis dans l'argument **group** de chaque fonction d'affichage d'entité (e.g. **addMarkers()**, **addCircles()**, **addPolygons()**, etc.). Pour que l'utilisateur puisse cocher ou décocher ces groupes dans le menu, il faut faire le lien dans la fonction **addLayersControl()** où l'on rappel le noms les groupes concernant les fonds de cartes ou les couches géographiques dans les argument **baseGroups** et **overlayGroups**. On peut également utiliser les fonctions **showGroup()** et **hideGroup()** pour afficher ou masquer des groupes. Par défaut, le menu ainsi créé s'affiche dans le coin supérieur droit de la carte, mais on peut choisir un autre emplacement à l'aide de l'argument **position** de la fonction **addLayersControl()**.

```
leaflet() %>% setView(lng = 13, lat = 55, zoom = 3) %>%
  addProviderTiles(provider = "OpenStreetMap", group = "OpenStreetMap"  ) %>%
  addProviderTiles(provider = "Stamen.Toner" , group = "Stamen Toner") %>%
  addPolygons(data = EU_ctr1_W84, color = ~pal_q(birth_rate), group = "Europe - Birth") %>%
  addLayersControl(baseGroups = c("OpenStreetMap", "Stamen Toner"), overlayGroups = "Europe - Birth")
```



7.2.4 Package mapview

Le package **mapview** (Appelhans *et al.*, 2016)[7] fournit des fonctions produisant des cartes interactives dans le navigateur Web avec JavaScript.

```
library(mapview)
```

Ce package est compatible avec le package **leaflet** (Cheng & Xie, 2015) (§ 7.2.3), dont il dépend.

7.2.4.1 Types de fonds de cartes

Ce package permet d'accéder à des fonds de cartes tels que :

- CartoDB Positron ;
- Esri WorldImagery ;
- OpenStreetMap ;
- OpenTopoMap ;
- Thunderforest.

7. Site web du package **mapview** : http://environmentalinformatics-marburg.github.io/mapview/introduction.html.

7.2.4.2 Nature des données d'entrée

Le package **mapview** autorise de travailler avec des données aux formats **sp** (aussi bien vectorielles que matricielles) :

- **SpatialPoints(DataFrame)** : points ;
- **SpatialLines(DataFrame)** : lignes ;
- **SpatialPolygons(DataFrame)** : polygones ;
- **SpatialPixels(DataFrame)** : pixels formant une grille rectangulaire possiblement incomplète ;
- **GridTopology(DataFrame)** : grille rectangulaire vide ;
- **SpatialGrid(DataFrame)** : grille rectangulaire pleine.

Il permet aussi d'utiliser des objets de classe **raster** :

- **RasterLayer** : raster simple ;
- **RasterStack** : raster multiple ;
- **RasterBrick** : raster multiple.

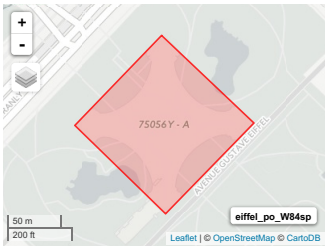
7.2.4.3 Gestion du système de coordonnées

Le package **mapview** permet d'utiliser des couches définies dans n'importe quel système de coordonnées. Dans tous les cas, l'affichage se fera toujours en Mercator (code EPSG 3857).

7.2.4.4 Affichage de la carte

Pour dessiner une carte, il suffit de fournir un objet spatial de classe **sp** ou **raster** à la fonction **mapView()**. On peut choisir de dessiner ou non l'entité grâce au menu dynamique s'affichant sur la carte. L'emprise de la carte est automatiquement définie en fonction de celle de l'objet dessiné. Si l'on s'est déplacé dans la page, on peut se repositionner par rapport à l'emprise d'un objet en cliquant sur le bouton portant le nom de ce dernier, en bas à droite de la carte.

```
mapView(eiffel_po_W84sp, color = "red")
```

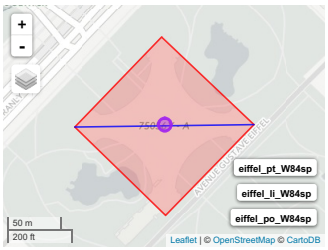


Le choix du type de fond de carte se fait au sein des commandes permettant le dessin de la carte, *via* l'argument **map.types** de la fonction **mapView()**. Si l'on ne fait pas appel à cette fonction, tous les fonds de cartes seront disponibles et l'on pourra passer de l'un à l'autre grâce au menu dynamique s'affichant sur la carte (par défaut, c'est le type CartoDB Positron qui sera présélectionné).

Notez que si l'on clique sur une entité avec le pointeur de la souris, les données de la table attributaire s'affichent dans une infobulle.

Pour superposer des couches, on peut tracer une première fois une carte avec la fonction **mapView()** (objet de classes **leaflet** et **htmlwidget**). Puis, pour dessiner une couche secondaire, il faut superposer les objets **mapview** à l'aide du signe plus ("**+**").

```
eiffel_rse_1 <- mapView(eiffel_po_W84sp, color = "red") +
                mapView(eiffel_li_W84sp, color = "blue") +
                mapView(eiffel_pt_W84sp, color = "purple")
eiffel_rse_1@map
```



Les objets ainsi créés sont de classe **mapview**. La carte est stockée dans l'élément **map**, lui même de double classe **leaflet** et **htmlwidget**. L'élément **object**, correspond à la liste de objets spatiaux dessinés sur la carte.

```
class(eiffel_rse_1)
## [1] "mapview"
## attr(,"package")
## [1] "mapview"
str(eiffel_rse_1, max.level = 2)
## Formal class 'mapview' [package "mapview"] with 2 slots
##   ..@ object:List of 3
##   ..@ map   :List of 8
##   .. ..- attr(*, "class")= chr [1:2] "leaflet" "htmlwidget"
##   .. ..- attr(*, "package")= chr "leaflet"
eiffel_rse_1@object
## [[1]]
## class       : SpatialPolygons
## features    : 1
## extent      : 2.29331, 2.295698, 48.85747, 48.85904  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
##
## [[2]]
## SpatialLines:
## class       : SpatialLines
## features    : 1
## extent      : 2.29331, 2.295698, 48.85825, 48.85827  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## Coordinate Reference System (CRS) arguments: +proj=longlat +datum=WGS84 +no_defs
## +ellps=WGS84 +towgs84=0,0,0
##
## [[3]]
## SpatialPoints:
##             x        y
## [1,] 2.294511 48.85827
## Coordinate Reference System (CRS) arguments: +proj=longlat +datum=WGS84 +no_defs
## +ellps=WGS84 +towgs84=0,0,0
```

7.2.4.5 Éléments contextuels

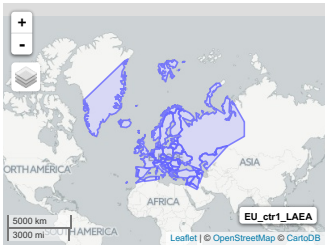
La fonction **mapviewOptions()** permet de gérer les options d'affichage de la carte, telles que :

- **basemaps** : les type de fond de carte ;
- **vector.palette** : la palette pour les données matricielles ;
- **raster.palette** : la palette pour les données vectorielles ;
- **na.color** : la couleur représentant les données manquantes ;
- **legend.pos** : la position de la légende ;
- **layers.control.pos** : la position du menu d'affichage des couches et des fonds de cartes.

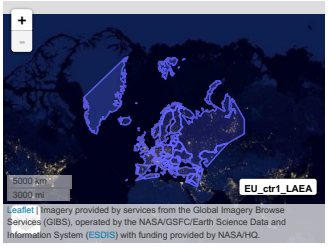
Il faut appeler cette fonction avant de tracer la carte. L'effet sera permanent sur l'ensemble des cartes tracées par la suite, tant que les valeurs d'origines n'auront pas été réinitialisées.

Dans l'exemple suivant, on affiche une première carte avec le fond par défaut (**"CartoDB.Positron)"**, puis on modifie les options d'affichage, notamment le fond (**"NASAGIBS.ViirsEarthAtNight2012"**), la pelette de couleurs, la position de la légende, etc. Puis, on modifie à nouveau le fond (**"OpenStreetMap"**), et l'on conserve les autres options modifiées précédemment.

```
mapView(EU_ctr1_LAEA)
```




```
mapviewOptions(basemaps = c("NASAGIBS.ViirsEarthAtNight2012", "OpenTopoMap"),
               vector.palette = colorRampPalette(RColorBrewer::brewer.pal(4, "Dark2")),
               na.color = "white",
               legend.pos = "topright",
               layers.control.pos = "bottomleft")
mapView(EU_ctr1_LAEA)
```




```
mapviewOptions(basemaps = "OpenStreetMap")
```

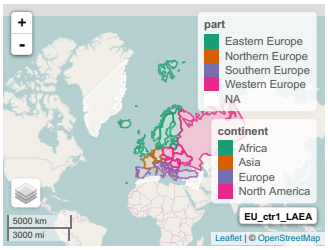
Légende. Il est possible d'ajouter une légende à une carte. Pour cela, il suffit que l'argument **legend** de la fonction **mapView()** soit égal à **TRUE**. Pour quelle s'affiche, il faut qu'une variable soit fournie à l'argument **zcol**.

Menu d'affichage des couches. Si l'argument **burst** de la fonction **mapView()** vaut **TRUE**, chaque entité d'une couche est considérée comme une couche en tant que tel. C'est-à-dire que dans le menu de gestion de l'affichage on peut faire apparaître ou bien masquer indépendamment chacune des entités.

7.2.4.6 Carte typologique

Pour réaliser une carte typologique, il suffit de fournir le nom d'une variable qualitative de la table attributaire à l'argument **zcol** de la fonction **mapView()**. Si plusieurs variables sont fournies, plusieurs couches seront disponibles sur la carte, et l'on pourra passer de l'une à l'autre à l'aide du menu interactif.

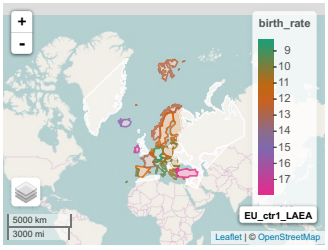
```
mapView(EU_ctr1_LAEA, zcol = c("part", "continent"), legend = TRUE)
```



7.2.4.7 Carte choroplèthe

Pour réaliser une carte choroplèthe, c'est le même principe ; il faut de passer le nom d'une variable quantitative de la table attributaire de l'objet **sp** à l'argument **zcol** de la fonction **mapView()**. Ici aussi, si plusieurs variables sont fournies, le menu permet de choisir la carte à dessiner.

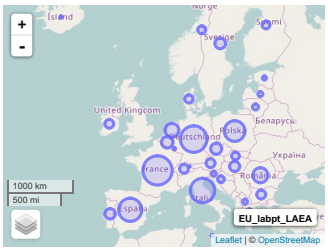
```
mapView(EU_ctr1_LAEA, zcol = c("birth_rate", "death_rate"), legend = TRUE)
```



7.2.4.8 Carte en symboles proportionnels

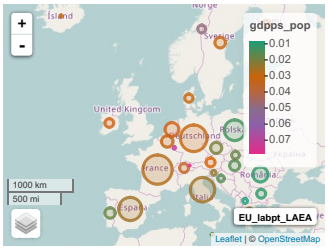
Concernant les cartes en symboles proportionneles, on fournit un objet **SpatialPointsDataFrame** à la fonction **mapView()**. Pour gérer la taille des cercles en fonction d'une variable, il suffit de fournir un vecteur à l'argument **cex**. Il n'est pas possible d'éfficher de manière simple une légende décrivant la taille des cercles.

```
mapView(EU_labpt_LAEA, cex = sqrt(EU_labpt_LAEA@data$birth)/50, legend = TRUE)
```



Si l'on souhaite représenter deux variables simultanément, on procède comme précédemment pour la taille des cercles, et pour les colorer en fonction d'une variable, on passe le nom de cette dernière à l'argument **zcol**.

```
mapView(EU_labpt_LAEA, zcol = "gdpps_pop", cex = sqrt(EU_labpt_LAEA@data$birth)/50, legend = TRUE)
```



7.2.4.9 Carte comparative interactive

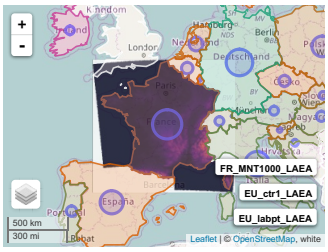
La fonction **slideView()** permet de superposer deux images matricielles et permet de disposer d'un curseur permettant de comparer de manière interactive les deux images de façon avant-après comme. Les objets à afficher peuvent être de classes **RasterLayer**, **RasterBrick**, **RasterStack** ou bien encore deux chaînes de caractères, qui correspondent alors à des chemins pointant vers des fichiers images au format PNG.

```
class(marylin)
marylin1 <- marylin2 <- marylin
marylin2[][, c(1, 2, 3)] <- marylin2[][, c(3, 1, 2)]
slideView(img1 = marylin1, img2 = marylin2, r = 1, g = 2, b = 3)
```

7.2.4.10 Superposer des couches

Pour superposer des couches, il suffit de réaliser autant un appel à la fonction **mapView()** pour chacune des couches que l'on souhaite dessiner. On les ajoute les unes aux autres en séparant les commandes par le signe plus ("+").

```
mapView(EU_labpt_LAEA, cex = sqrt(EU_labpt_LAEA@data$birth)/50) +
  mapView(EU_ctr1_LAEA, zcol = "birth_rate") +
  mapView(FR_MNT1000_LAEA, maxpixels =  2e5)
```



7.2.4.11 Assembler des cartes

Bien que produites aux format HTML, il est possible d'assembler des cartes dans une même fenêtre à l'aide des fonctions **latticeView()** ou **sync()**. La première fonction permet d'effectuer un assemblage simple. La seconde, quant à elle, permet de synchroniser toutes ou une partie des cartes, c'est-à-dire que la navigation effectuée dans une carte sera appliquée aux autres, si l'argument **no.initial.sync** vaut **FALSE** (ou un vecteur de valeurs entières définissant les numéro des cartes à synchroniser). On peut synchroniser le curseur de la souris sur les cartes jumelles à l'aide de l'argument **sync.cursor = TRUE**. L'argument **ncol** permet de définir le nombre de colonnes souhaité ; le nombre de lignes, quant à lui, s'ajuste automatiquement en fonction du nombre de cartes à tracer.

```
mpv1 <- mapView(EU_ctr1_LAEA, zcol = "part")
mpv2 <- mapView(EU_labpt_LAEA, cex = sqrt(EU_labpt_LAEA@data$birth)/50)
sync(mpv1, mpv2, no.initial.sync = FALSE, ncol = 2)
```

7.2.5 Package tmap

Comme nous l'avons déjà vu, le package **tmap** (Tennekes, 2016) est spécialisé dans la production cartographique. Il utilise une syntaxe de type *Grammar of Graphics* (Wilkinson, 2005) (§ 6.3). Toutes les fonctions cartographiques peuvent être utilisées pour afficher des données en mode statique (sans fond de carte) ou bien sur un fond de carte dynamique. C'est ce dernier cas qui nous intéresse ici.

```
library(tmap)
```

Le package **tmap** dépend du package **leaflet** (Cheng & Xie, 2015), que nous avons déjà présenté (§ 7.2.3).

7.2.5.1 Types de fonds de cartes

Le package **tmap** permet la prise en charge des fonds de cartes suivants :
- CartoDB Positron (par défaut) ;
- OpenStreetMap ;
- Esri WorldTopoMap.

7.2.5.2 Nature des données d'entrée

Ce package permet la prise en compte des données au format **sp** (aussi bien vectorielles que matricielles) :
- **SpatialPoints(DataFrame)** : points ;
- **SpatialLines(DataFrame)** : lignes ;
- **SpatialPolygons(DataFrame)** : polygones ;
- **SpatialPixels(DataFrame)** : pixels formant une grille rectangulaire possiblement incomplète ;
- **SpatialGrid(DataFrame)** : grille rectangulaire pleine.

Il permet également d'utiliser des objets de classe **raster** :
- **RasterLayer** : raster simple ;
- **RasterStack** : raster multiple ;
- **RasterBrick** : raster multiple.

7.2.5.3 Gestion du système de coordonnées

Le package **tmap** permet d'utiliser des couches définies dans n'importe quel système de coordonnées. Dans tous les cas, l'affichage se fera toujours en Mercator (code EPSG 3857).

7.2.5.4 Affichage de la carte

Pour afficher le fond de carte, avant de réaliser le dessin de la carte avec la fonction **tm_shape()**, il faut préalablement définir le mode d'affichage avec la fonction **tm_mode()**, en définissant l'argument **mode = "view"** (par défaut, **mode = "plot"**). Le mode choisi restera actif pour l'ensemble de cartes dessinées par la suite. Si l'on souhaite revenir au mode statique, il faudra faire un nouvel appel à cette fonction en choisissant le mode **"plot"**.

```
tmap_mode(mode = "view")
```

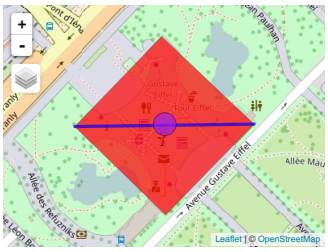
Le choix du type de fond de carte se fait au sein des commandes permettant le dessin de la carte, *via* l'argument **basemaps** de la fonction **tm_view()**. Si l'on ne fait pas appel à cette fonction, tous les fonds de cartes seront disponibles et l'on pourra passer de l'un à l'autre grâce au menu dynamique s'affichant sur la carte (par défaut, c'est le type CartoDB Positron qui sera présélectionné).

```
eiffel_pt_W84_tmp <- tm_shape(eiffel_pt_W84sp) +
  tm_dots(col = "purple", size = 0.5) +
  tm_view(basemaps = "CartoDB.Positron")
class(eiffel_pt_W84_tmp)
## [1] "tmap"
eiffel_pt_W84_tmp
```

Notez que si l'on clique sur une entité avec le pointeur de la souris, les données de la table attributaire s'affichent dans une infobulle. Par ailleurs, on peut choisir de dessiner ou non l'entité grâce au menu interactif.

Pour superposer des couches, il suffit d'additionner les différents appels, comme vu précédemment (§ 6.3.4).

```
eiffel_pt_W84_tmp <- tm_shape(eiffel_pt_W84sp) +
  tm_dots(col = "purple", size = 0.5) +
  tm_view(basemaps = "CartoDB.Positron")
eiffel_li_W84_tmp <- tm_shape(eiffel_li_W84sp) +
  tm_lines(col = "blue", lwd = 4) +
  tm_view(basemaps = "OpenStreetMap")
eiffel_po_W84_tmp <- tm_shape(eiffel_po_W84sp) +
  tm_polygons(col = "red") +
  tm_view(basemaps = "Esri.WorldTopoMap")
eiffel_po_W84_tmp + eiffel_li_W84_tmp + eiffel_pt_W84_tmp
```

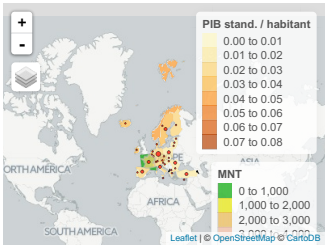


Si les fonds de cartes ne sont pas les mêmes dans les différents appels, ce sera celui défini dans la couche principale qui sera utilisé (**`tm_shape(is.master = TRUE)`** ; par défaut la première).

7.2.5.5 Différents types de cartes

Pour dessiner des cartes typologiques, des cartes choroplèthes, des cartes en symboles proportionnels, etc., il n'y a pas de difficulté particulière, il faut procéder comme nous l'avons décrit précédemment (§ 6.3.4). La différence qui existe avec une carte statique est qu'on ne peut travailler ici qu'avec une seule variable ; avec une carte statique on avait automatiquement autant de cartes que de variables utilisées.

```
tm_shape(EU_ctr1_LAEA) +
  tm_fill("gdpps_pop", title = "PIB stand. / habitant", textNA = "Inconnu") +
tm_shape(FR_MNT1000_LAEA) +
  tm_raster(palette = terrain.colors(5), title = "MNT") +
tm_shape(EU_ctr1_LAEA, is.master = TRUE) +
  tm_bubbles("birth", col = "red", border.col = "black",
             scale = 2, title.size = "Nb. de naissances") +
tm_shape(FR_ctr_L93) +
  tm_borders(col = "grey", lwd = 2)
```



7.2.6 Package leafletR

Le package **leafletR** (Graul, 2015) fournit des fonctions produisant des cartes interactives dans le navigateur Web avec JavaScript.

```
library(leafletR)
```

Attention, ce package peut rentrer en conflit avec les packages **leaflet** (Cheng & Xie, 2015), **mapview** (Appelhans *et al.*, 2016) et **tmap** (Tennekes, 2016). Il faut donc éviter d'utiliser **leafletR** avant d'appeler ces derniers, sinon, ils risquent ne pas fonctionner correctement.

7.2.6.1 Types de fonds de cartes

Le package **leafletR** permet d'accéder à différents fonds de cartes topographiques :

- CartoDB Dark matter ;
- CartoDB Positron ;
- MapQuest Open Aerial ;
- MapQuest OSM ;
- OpenStreetMap (par défaut) ;
- Stamen Toner ;
- Stamen Toner background ;
- Stamen Toner lite ;
- Stamen Watercolor ;
- Thunderforest Landscape.

7.2.6.2 Nature des données d'entrée

Le package **leafletR** travaille avec des données aux formats GeoJSON et TopoJSON, utilisant la norme JSON (pour *JavaScript Object Notation*). Il propose la fonction **toGeoJSON()**, qui permet de convertir un **data.frame**, ou un objet **sp**, au format GeoJSON. Attention, les objet spatiaux contenant des points doivent obligatoirement être de la classe **SpatialPointsDataFrame** et non de la classe **SpatialPoints** ; il n' y pas de problème particulier pour les autres types d'entités. La fonction **toGeoJSON()** exporte un fichier ; il faut donc lui spécifier son nom dans l'argument **name**, et le répertoire de destination dans l'argument **dest** (par défaut, le répertoire courant).

```
dest <- tempdir()
eiffel_po_W84js <- toGeoJSON(data = eiffel_po_W84sp, name = "Tour_Eiffel_Polygone" , dest = dest)
eiffel_li_W84js <- toGeoJSON(data = eiffel_li_W84sp, name = "Tour_Eiffel_Polyligne", dest = dest)
eiffel_pt_W84js <- toGeoJSON(data = eiffel_pt_W84  , name = "Tour_Eiffel_Point"    , dest = dest)
eiffel_pt_W84js
## [1] "/tmp/Rtmpki1dVa/Tour_Eiffel_Point.geojson"
```

7.2.6.3 Affichage de la carte

Pour dessiner la carte, il suffit de faire appel à la fonction **leaflet()**. Le résultat est obligatoirement sauvé dans un fichier au format HTML, dans un répertoire défini par l'argument **dest** (par défaut, le répertoire courant). Le nom du fichier est celui défini dans l'argument **name** de la fonction **toGeoJSON()**. L'argument **base.map** permet de choisir le type de fond de carte souhaité. Le style de la carte, quant à lui, se définit dans l'argument **style** auquel on passe le résultat d'une fonction de style. Pour utiliser une style unique pour l'ensemble des entités, il faut utiliser la fonction **styleSingle()**. Nous verrons par la suite quelles sont les autres fonctions de style disponibles.

```
eiffel_pt_W84_llr <- leaflet(eiffel_pt_W84js, base.map = "water",
                             style = styleSingle(col = "purple"), dest = dest)

eiffel_pt_W84_llr
```

L'objet ainsi créé est de classe **leaflet**, mais n'a rien à voir avec la classe **leaflet** du package **leaflet** (Cheng & Xie, 2015), présenté auparavant (§ 7.2.3).

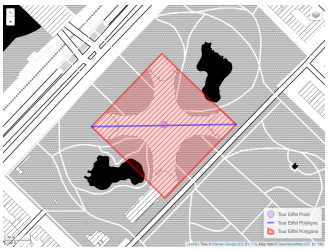
```
class(eiffel_pt_W84_llr)
```

Pour superposer des couches, il suffit de fournir une liste de fichiers GeoJSON à l'argument **data** de la fonction **leaflet()**. Il faut également définir une liste de styles correspondant à chacune des couches à dessiner.

```
eiffel_llr <- leaflet(data = list(eiffel_po_W84js, eiffel_li_W84js, eiffel_pt_W84js),
                      base.map = "tonerbg",
                      style = list(styleSingle(col = "red"),
```

```
                                    styleSingle(col = "blue"),
                                    styleSingle(col = "purple")),
                        dest = dest)
```

```
eiffel_llr
```



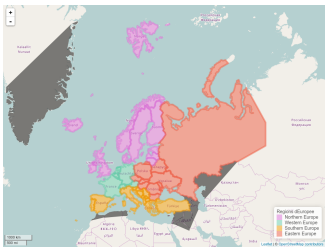
7.2.6.4 Carte typologique

Pour tracer une carte typologique, il suffit d'utiliser la fonction **leaflet()**, après avoir créé le style adéquat à l'aide la fonction **styleCat()**. Ce style est spécifique aux variables catégorielles. On définit le nom de la variable qualitative d'intérêt des données attributaires dans l'argument **prop**. L'argument **val**, contient, quant à lui, les modalités possibles de la variable, et l'on attribue a chacune d'entre-elle une couleur *via* l'argument **style.val**. On donne un titre à la légende dans l'argument **leg** (attention, il ne faut pas utiliser de caractère spécial ou d'accent). Pour cette représentation, on passe des entités de type "polygone" à l'argument **data** de fonction **leaflet()**.

```
EU_ctr1_W84js <- toGeoJSON(data = EU_ctr1_W84, name = "Europe_ctr", dest = dest)
```

```
sty <- styleCat(prop = "part", val = levels(unique(EU_ctr1_W84@data$part)),
                style.val = c("orchid", "aquamarine3", "orange2", "tomato2"),
                leg = "Regions dEuropee")
EU_lfr1 <- leaflet(data = EU_ctr1_W84js, style = sty, dest = dest)
```

```
EU_lfr1
```

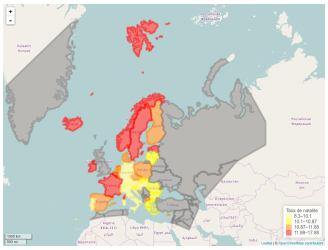


7.2.6.5 Carte choroplèthe

Pour dessiner une carte choroplèthe, c'est le même principe, mais comme on s'intéresse à une variable quantitative, le style créé doit être adapté ; pour cela, il faut utiliser la fonction **styleGrad()**. Cette dernière prend elle aussi le nom de la variable dans l'argument **prop**. On la discrétise en définissant des bornes de classes dans **breaks** et on associe les couleurs dans **style.val**. Pour cette représentation, on passe des entités de type "polygone" à l'argument **data** de fonction **leaflet()**.

```
sty <- styleGrad(prop = "birth_rate",
                 breaks = round(quantile(EU_ctr1_W84@data$birth_rate, na.rm = TRUE), dig = 2),
                 style.val = rev(heat.colors(4)),
                 leg = "Taux de natalite")
EU_lfr2 <- leaflet(data = EU_ctr1_W84js, style = sty, dest = dest)
```

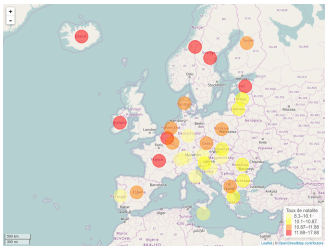
```
EU_lfr2
```



7.2.6.6 Carte en symboles proportionnels

La démarche est toujours la même. Selon le type de variable que l'on souhaite représenter, on se sert des fonctions **styleSingle()**, **styleCat()** ou **styleGrad()**. La taille des cercles est gérée par l'argument **rad**. La fonction **styleSingle()** propose également de représenter des symboles de type "marqueur". Pour cette représentation, on passe des entités de type "point" à l'argument **data** de fonction **leaflet()**.

```
EU_labpt_W84js <- toGeoJSON(data = EU_labpt_W84, name = "Europe_cen", dest = dest)

sty <- styleGrad(prop = "birth_rate",
                 breaks = round(quantile(EU_ctr1_W84@data$birth_rate, na.rm = TRUE), dig = 2),
                 style.val = rev(heat.colors(4)),
                 rad = 20,
                 leg = "Taux de natalite")
EU_lfr3 <- leaflet(data = EU_labpt_W84js, style = sty, dest = dest)

EU_lfr3
```



7.3 Autres solutions

7.3.1 Logiciel Google Earth

Il est possible de produire des fichiers au format KML afin de visualiser les données dans **Google Earth** [8]. Plusieurs packages permettent d'écrire ce format de fichiers :

- **maptools** (Bivand & Lewin-Koh, 2015) ;
- **rgdal** (Bivand *et al.*, 2014) ;
- **raster** (Hijmans, 2015) ;
- **plotKML** (Hengl *et al.*, 2015) ;
- **RKML** (Lang & Nolan, 2011) [9].

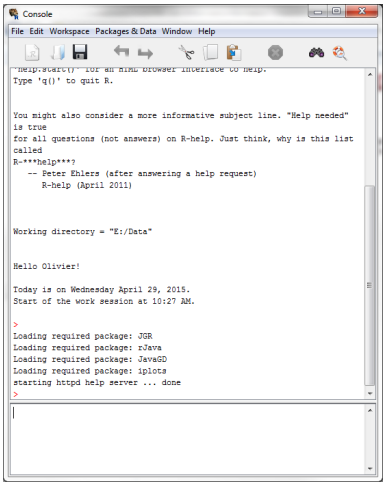
7.3.2 Package DeducerSpatial

Le package **DeducerSpatial** propose un plugin pour l'interface graphique JGR (Helbig *et al.*, 2013) [10] (pour *Java GUI for R*).

```
library(JGR)
```

On lance l'interface graphique JGR en appelant la fonction **JGR()** :

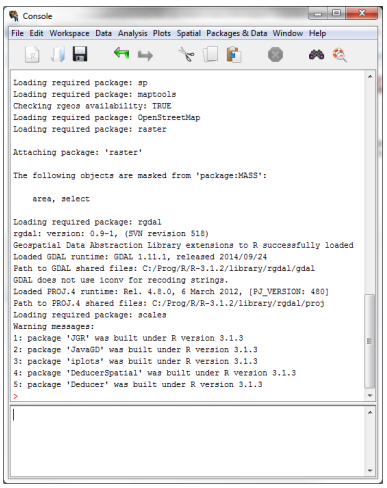
```
JGR()
```



Une fois dans JGR, on appelle le package **DeducerSpatial**, ce qui ajoute automatiquement des menus à l'interface existante.

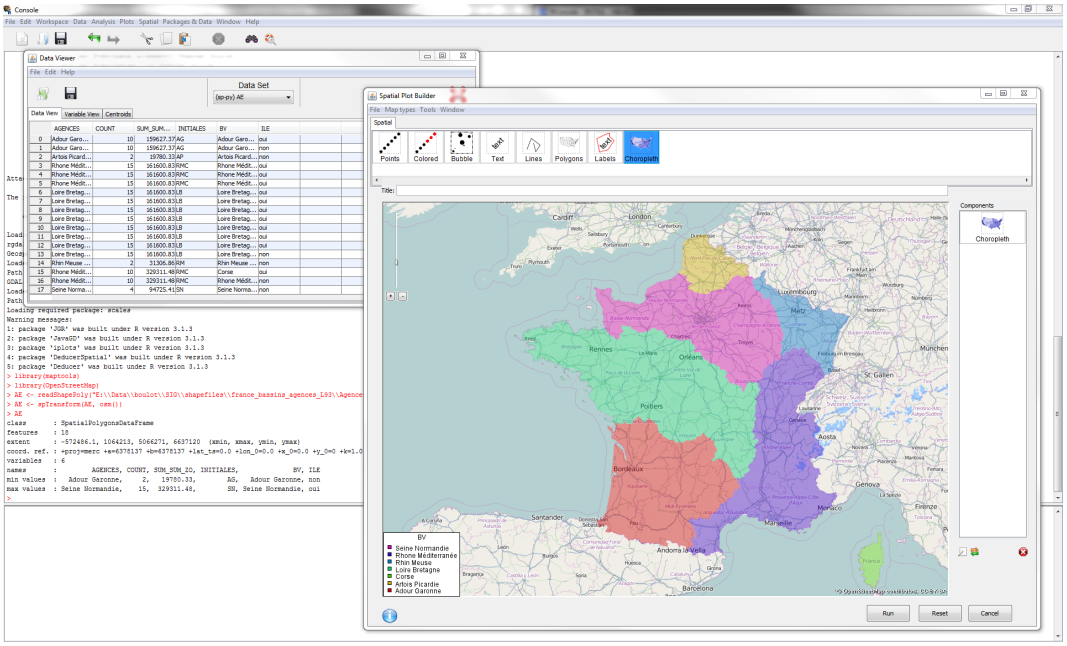
```
library(DeducerSpatial)
```

8. Site web de Google Earth : https://www.google.fr/intl/fr/earth/.
9. Le package RKML n'est pas disponible sur le site web du CRAN, on peut le télécharger à l'adresse suivante : http://www.omegahat.org/RKML/.
10. Site web de JGR : https://www.rforge.net/JGR/.



Il est alors possible de charger des objets et de disposer des interfaces graphiques Data Viewer et Spatial Plot Builder. Le fond de carte est celui d'**OpenStreetMap**, le système est donc le Mercator (code EPSG 3857).

```
AE <- rgdal::readOGR(dsn = ".", layer = "Agences_L93", p4s = "+init=epsg:2154")
AE <- spTransform(AE, OpenStreetMap::osm())
```



Bibliographie

Adler, D., Murdoch, D. & others (2014). *rgl: 3D visualization device system (OpenGL)*. URL http://CRAN.R-project.org/package=rgl.

Appelhans, T., Detsch, F., Reudenbach, C. & Woellauer, S. (2016). *mapview: Interactive Viewing of Spatial Objects in R*. URL http://CRAN.R-project.org/package=mapview.

Apple Inc. (2016). Apple Maps Connect. URL https://mapsconnect.apple.com/.

Arnold, J.B. (2016). *ggthemes: Extra Themes, Scales and Geoms for ggplot2*. URL https://CRAN.R-project.org/package=ggthemes.

Auguie, B. (2016). *gridExtra: Miscellaneous Functions for Grid Graphics*. URL https://CRAN.R-project.org/package=gridExtra.

Bache, S.M. & Wickham, H. (2014). *magrittr: A Forward-Pipe Operator for R*. URL https://CRAN.R-project.org/package=magrittr.

Baddeley, A. & Turner, R. (2005). spatstat: An R package for analyzing spatial point patterns. *Journal of Statistical Software* **12**, 1–42, URL http://www.jstatsoft.org/v12/i06/.

Baddeley, A., Turner, R., Mateu, J. & Bevan, A. (2013). Hybrids of gibbs point process models and their implementation. *Journal of Statistical Software* **55**, 1–43, URL http://www.jstatsoft.org/v55/i11/.

Becker, R.A. & Chambers, J.M. (1977). GR-Z: A System of Graphical Subroutines for Data Analysis. *Computer Science and Statistics*, p. 409–415, Gaithersburg, Maryland.

Becker, R.A. & Chambers, J.M. (1984a). Design of the S system for data analysis. *Communications of the ACM* **27**, 486–495, URL http://portal.acm.org/citation.cfm?doid=358189.358078.

Becker, R.A. & Chambers, J.M. (1984b). *S: an interactive environment for data analysis and graphics*. The Wadsworth statistics/probability series, Wadsworth Advanced Book Program, Belmont, Calif.

Becker, R.A. & Chambers, J.M. (1985). *Extending the S system*. Wadsworth Advanced Books and Software, Monterey, Calif.

Becker R.A. (original S code), Wilks A.R. (original S code) & Brownrigg R. (R version) (2014a). *mapdata: Extra Map Databases*. URL http://CRAN.R-project.org/package=mapdata.

Becker R.A. (original S code), Wilks A.R. (original S code), Brownrigg R. (R version) & Minka T.P. (enhancements) (2014b). *maps: Draw Geographical Maps*. URL http://CRAN.R-project.org/package=maps.

Bivand, R., Pebesma, E. & Gómez-Rubio, V. (2013a). *Applied spatial data analysis with R, Second edition*. Springer, New York, USA, URL http://www.asdar-book.org/.

Bivand, R. (2007). Using the R–GRASS interface: Current status. *OSGeo Journal* **1**, 36–38, URL http://www.osgeo.org/files/journal/final_pdfs/OSGeo_vol1_GRASS-R.pdf.

Bivand, R. (2014). *spgrass6: Interface between GRASS 6 and R*. URL http://CRAN.R-project.org/package=spgrass6.

Bivand, R. (2015). *rgrass7: Interface Between GRASS 7 Geographical Information System and R*. URL http://CRAN.R-project.org/package=rgrass7.

Bivand, R., Hauke, J. & Kossowski, T. (2013b). Computing the jacobian in gaussian spatial autoregressive models: An illustrated comparison of available methods. *Geographical Analysis* **45**, 150–179, URL http://www.jstatsoft.org/v63/i18/.

Bivand, R., Keitt, T. & Rowlingson, B. (2014). *rgdal: Bindings for the Geospatial Data Abstraction Library*. URL http://CRAN.R-project.org/package=rgdal.

Bivand, R. & Lewin-Koh, N. (2015). *maptools: Tools for Reading and Handling Spatial Objects*. URL http://CRAN.R-project.org/package=maptools.

Bivand, R., Pebesma, E.J. & Gomez-Rubio, V. (2008). *Applied Spatial Data Analysis with R*. Springer, New York.

Bivand, R. & Piras, G. (2015). Comparing implementations of estimation methods for spatial econometrics. *Journal of Statistical Software* **63**, 1–36, URL http://www.jstatsoft.org/v63/i18/.

Bivand, R. & Rundel, C. (2014). *rgeos: Interface to Geometry Engine - Open Source (GEOS)*. URL http://CRAN.R-project.org/package=rgeos.

Boessenkool, B. (2016). *OSMscale: Add a Scale Bar to OpenStreetMap Plots*. URL http://CRAN.R-project.org/package=OSMscale.

Brenning, A. (2008). Statistical geocomputing combining R and SAGA: The example of landslide susceptibility analysis with generalized additive models. *SAGA – Seconds Out (= Hamburger Beitraege zur Physischen Geographie und Landschaftsoekologie, vol. 19)*, p. 23–32, J. Boehner, T. Blaschke, L. Montanarella.

Brenning, A. (2012). *RPyGeo: ArcGIS Geoprocessing in R via Python*. URL http://CRAN.R-project.org/package=RPyGeo.

Brunsdon, C. & Charlton, M. (2014). *getcartr: Front end for Rcartogram package*.

Brunsdon, C. & Chen, H. (2014). *GISTools: Some further GIS capabilities for R*. URL http://CRAN.R-project.org/package=GISTools.

Carr, D., ported by Nicholas Lewin-Koh, Maechler, M. & contains copies of lattice functions written by Deepayan Sarkar (2015). *hexbin: Hexagonal Binning Routines*. URL https://CRAN.R-project.org/package=hexbin.

CartoDB Inc. (2016). CartoDB. URL https://carto.com/.

Chang, W. (2016). *webshot: Take Screenshots of Web Pages*. URL https://github.com/wch/webshot/.

Cheng, J. & Xie, Y. (2015). *leaflet: Create Interactive Web Maps with the JavaScript Leaflet Library*. URL http://CRAN.R-project.org/package=leaflet.

Cleveland, W.S. (1993). *Visualizing data*. At&T Bell Laboratories; Published by Hobart Press, Murray Hill, N.J.

CloudMade (2016). CloudMade. URL http://cloudmade.com/.

ESRI (2013). ArcGIS Desktop: Release 10.2. Redlands, CA: Environmental Systems Research Institute.

ESRI (2016a). *arcgisbinding: Bindings for ArcGIS*. URL http://esri.com.

ESRI (2016b). ESRI Web Map. URL https://www.arcgis.com/home/webmap/viewer.html.

Eugster, M.J.A. & Schlesinger, T. (2010). osmar: OpenStreetMap and R. *R Journal* URL http://osmar.r-forge.r-project.org/RJpreprint.pdf.

Evenden, G., Warmerdam, F. & Butler, H. (2015). *PROJ.4 – Cartographic Projections Library*. URL http://trac.osgeo.org/proj/.

Fellows, I., & Stotz J.P. (JMapViewer library) (2013). *OpenStreetMap: Access to open street map raster images*. URL http://CRAN.R-project.org/package=OpenStreetMap.

GDAL Development Team (2012). *GDAL - Geospatial Data Abstraction Library, Version 1.9.2*. Open Source Geospatial Foundation, URL http://www.gdal.org.

Gesmann, M. & de Castillo, D. (2011). googleVis: Interface between r and the google visualisation api. *The R Journal* **3**, 40–44, URL http://journal.r-project.org/archive/2011-2/RJournal_2011-2_Gesmann+de~Castillo.pdf.

Giraud, T. & Lambert, N. (2016). *cartography: Thematic Cartography*. URL http://CRAN.R-project.org/package=cartography.

Giraud, T. (2013). *rCarto: This package builds maps with a full cartographic layout.* URL http://CRAN.R-project.org/package=rCarto.

Google (2016). Google Maps. URL https://www.google.fr/maps/.

GRASS Development Team (2015). *Geographic Resources Analysis Support System (GRASS GIS) Software.* Open Source Geospatial Foundation, USA, URL http://grass.osgeo.org.

Graul, C. (2015). *leafletR: Interactive Web-Maps Based on the Leaflet JavaScript Library.* URL http://cran.r-project.org/package=leafletR.

Greenberg, J.A. & Mattiuzzi, M. (2014). *gdalUtils: Wrappers for the Geospatial Data Abstraction Library (GDAL) Utilities.* URL http://CRAN.R-project.org/package=gdalUtils.

Haklay, M. & Weber, P. (2008). Openstreetmap: User-generated street maps. *IEEE Pervasive Computing* **7**, 12–18.

Helbig, M., Urbanek, S. & Fellows, I. (2013). *JGR: Java GUI for R.* URL http://CRAN.R-project.org/package=JGR.

Hengl, T., Roudier, P., Beaudette, D. & Pebesma, E. (2015). plotKML: Scientific visualization of spatio-temporal data. *Journal of Statistical Software* **63**, 1–25, URL http://www.jstatsoft.org/v63/i05/.

Hijmans, R.J. (2014). *geosphere: Spherical Trigonometry.* URL http://CRAN.R-project.org/package=geosphere.

Hijmans, R.J. (2015). *raster: Geographic data analysis and modeling.* URL http://CRAN.R-project.org/package=raster.

Hijmans, R.J., Phillips, S., Leathwick, J. & Elith, J. (2014). *dismo: Species distribution modeling.* URL http://CRAN.R-project.org/package=dismo.

Jeworutzki, S. (2016). *cartogram: Create Cartograms with R.* URL https://CRAN.R-project.org/package=cartogram.

Kahle, D. & Wickham, H. (2013). ggmap: Spatial visualization with ggplot2. *The R Journal* **5**, 144–161, URL http://journal.r-project.org/archive/2013-1/kahle-wickham.pdf.

Kilibarda, M. & Bajat, B. (2012). plotGoogleMaps: the R-based web-mapping tool for thematic spatial data. *GEOMATICA* **66**, 37–49.

Lang, D.T. (2011). *Rcartogram: Interface to Mark Newman's cartogram software.*

Lang, D.T. & Nolan, D. (2011). *RKML: Simple tools for creating KML displays from R.*

Lewis, B.W. (2016). *threejs: Interactive 3D Scatter Plots, Networks and Globes.* URL http://CRAN.R-project.org/package=threejs.

Loecher, M. & Ropkins, K. (2015). RgoogleMaps and loa: Unleashing R graphics power on map tiles. *Journal of Statistical Software* **63**, 1–18, URL http://www.jstatsoft.org/v63/i04/.

MapQuest Inc. (2013). MapQuest. URL https://www.mapquest.com/.

McIlroy, D., Brownrigg R. (R package), Minka T.P. (R package) & Bivand R. (transition to Plan 9 codebase). (2015). *mapproj: Map Projections.* URL http://CRAN.R-project.org/package=mapproj.

Microsoft (2016). Bing Maps. URL https://www.bing.com/maps/.

Muenchow, J. & Schratz, P. (2016). *RQGIS: Integrating R with QGIS.* URL http://CRAN.R-project.org/package=RQGIS.

National Aeronautics and Space Administration (2016). NASA Global Imagery Browse Services. URL https://earthdata.nasa.gov/about/science-system-description/eosdis-components/global-imagery-browse-services-gibs/.

National Park Service of the United States Department of the Interior (2016). NPMap. URL https://www.nps.gov/npmap/.

Neuwirth, E. (2014). *RColorBrewer: ColorBrewer Palettes.* URL https://CRAN.R-project.org/package=RColorBrewer.

Next Human Network (2016). Naver. URL http://map.naver.com/.

Nychka, D., Furrer, R. & Sain, S. (2014). *fields: Tools for spatial data*. URL http://CRAN.R-project.org/package=fields.

OpenStreetMap contributors (2016). OpenStreetMap. URL https://www.openstreetmap.org/.

OpenWeatherMap Inc. (2016). OpenWeatherMap. URL https://openweathermap.org/.

Padgham, M. (2016). *osmplotr: Customisable Images of OpenStreetMap Data*. URL https://CRAN.R-project.org/package=osmplotr.

Pavlenko, A. (2016). Mapnik. URL http://mapnik.org/.

Pebesma, E.J. & Bivand, R.S. (2005). Classes and methods for spatial data in R. *R News* **5**, 9–13, URL http://CRAN.R-project.org/doc/Rnews/.

Perpiñán, O. & Hijmans, R. (2014). *rasterVis*. URL http://oscarperpinan.github.io/rastervis/.

QGIS Development Team (2016). *QGIS Geographic Information System*. Open Source Geospatial Foundation, URL http://qgis.osgeo.org.

R Core Team (2015). *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ...* URL http://CRAN.R-project.org/package=foreign.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, URL http://www.R-project.org/.

Renka R.J. (Fortran code), Gebhardt A. (R port), Eglen S. (contributions), Zuyev S. (contributions) & White D. (contributions) (2013). *tripack: Triangulation of irregularly spaced data*. URL http://CRAN.R-project.org/package=tripack.

Rudis, B. (2016). *ggalt: Extra Coordinate Systems, Geoms and Statistical Transformations for ggplot2*. URL http://CRAN.R-project.org/package=ggalt.

SAGA Development Team (2008). *System for Automated Geoscientific Analyses (SAGA GIS)*. Germany, URL http://www.saga-gis.org/.

Santos Baquero, O. (2016). *ggsn: North Symbols and Scale Bars for Maps Created with ggplot2 or ggmap*. URL https://CRAN.R-project.org/package=ggsn.

Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. Springer, New York, URL http://lmdvr.r-forge.r-project.org.

Sarkar, D. & Andrews, F. (2016). *latticeExtra: Extra Graphical Utilities Based on Lattice*. URL https://CRAN.R-project.org/package=latticeExtra.

Schnute, J.T., Boers, N. & Haigh, R. (2004). *PBS Mapping 2: User's Guide*.

Schnute, J.T., Boers, N., Haigh, R., Grandin, C., Chabot, D., Johnson, A., Wessel, P., Antonio, F., Lewin-Koh, N.J. & Bivand, R. (2015). *PBSmapping: Mapping Fisheries Data and Spatial Analysis Tools*. URL http://CRAN.R-project.org/package=PBSmapping.

Stamen Design (2016). Stamen. URL http://maps.stamen.com/.

Tarboton, D.G. (1997). A new method for the determination of flow directions and upslope areas in grid digital elevation models. *Water Resources Research* **33**, 309–319.

Tarboton, D.G. (2013). *TauDEM 5.1. Guide to using the TauDEM command line functions*. Utah university.

Telenav Inc. (2016). Skobbler. URL http://www.skobbler.com/.

Tennekes, M. (2016). *tmap: Thematic Maps*. URL http://CRAN.R-project.org/package=tmap.

Thunderforest & OpenStreetMap contributors (2016). Thunderforest. URL https://www.thunderforest.com/.

Toursprung GmbH (2016). Maptoolkit. URL http://www.maptoolkit.net/.

Vaidyanathan, R., Xie, Y., Allaire, J., Cheng, J. & Russell, K. (2016). *htmlwidgets: HTML Widgets for R*. URL https://CRAN.R-project.org/package=htmlwidgets.

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer New York, URL http://ggplot2.org.

Wikipedia (2015a). Plagiarism – Wikipedia, the free encyclopedia, ArcGIS. URL https://fr.wikipedia.org/wiki/ArcGIS.

Wikipedia (2015b). Plagiarism – Wikipedia, the free encyclopedia, GDAL. URL https://fr.wikipedia.org/wiki/GDAL.

Wikipedia (2015c). Plagiarism – Wikipedia, the free encyclopedia, GRASS GIS. URL https://fr.wikipedia.org/wiki/GRASS_GIS.

Wikipedia (2015d). Plagiarism – Wikipedia, the free encyclopedia, SAGA GIS. URL https://fr.wikipedia.org/wiki/SAGA_GIS.

Wilkinson, L. (2005). *The grammar of graphics*. Statistics and computing, Springer, New York, 2nd éd.

Table des matières