Université du Québec à Rimouski
Département de Biologie, Chimie et Géographie

ANALYSES SPATIALES SOUS R

Nicolas Casajus

Formation donnée aux membres
du groupe de recherche sur les
environnements nordiques
BORÉAS
- Hiver 2013 -

Avant-propos

Apparu au milieu des années quatre-vingt-dix, le logiciel R est en passe de devenir la référence dans le monde des logiciels de statistiques. C'est maintenant un outil incontournable dans les laboratoires de recherche, reléguant parfois au second plan des logiciels comme SPSS, Systat, S-PLUS ou encore SAS. Certes, R est présenté comme un logiciel d'analyses statistiques et de représentation graphique, mais c'est avant tout un langage de programmation complet et extrêmement puissant. Et c'est probablement cette particularité qui lui confère sa très grande popularité.

Ce système *open-source* est distribué librement sous les termes de la licence publique générale GNU. Cette licence fixe les conditions légales de distribution des logiciels libres et stipule entre autres que le code source doit être accessible à tous. Toute personne peut ainsi accéder à la source du logiciel, la modifier selon ses propres besoins, et la redistribuer librement (mais pas forcément gratuitement) sous les termes de cette même licence. Ainsi, un tel système n'est la propriété de personne (ou plutôt la propriété de tous ; *copyleft*) et encourage fortement la collaboration. Et c'est là que réside la force de R : c'est une très belle réussite, fruit d'une coopération internationale intensive où des chercheurs d'horizons divers ont contribué à améliorer cet environnement de travail, permettant à R d'acquérir « sa lettre » de noblesse.

Le domaine des analyses spatiales a énormément bénéficié de ce projet collaboratif et on dénombre actuellement plus d'une centaine d'extensions (*packages*) qui peuvent venir compléter l'environnement de base du logiciel R. Certains *packages* définissent des classes particulières d'objets spatiaux (`sp`, `raster`), d'autres permettent de lire et d'écrire des objets spatiaux (`rgdal`, `maptools`), de manipuler le système de coordonnées, ou encore de réaliser des statistiques spatiales (`spatial`, `spdep`, `gstat`, `geoR`). Cependant, R n'est pas un logiciel de SIG : il est très efficace pour analyser les objets spatiaux, mais reste limité quant à leur représentation cartographique. Nous verrons cependant qu'il est tout de même possible de produire des cartes assez esthétiques sous R. Notons enfin qu'il peut facilement communiquer avec des logiciels de SIG tels que GRASS ou Quantum GIS afin de représenter les objets spatiaux de manière plus professionnelle (partie non abordée dans ce document).

Mais, pourquoi utiliser le logiciel R pour manipuler des objets spatiaux quand des outils très puissants ont été développés spécialement à cet effet ? Son utilisation va trouver toute sa justification lorsqu'une chaîne de traitements devra être répétée un grand nombre de fois (automatisation des tâches), mais aussi pour réaliser l'ensemble du flux de travail (importation, modification de la géométrie, analyse statistique, cartographie, exportation) sur un même support logiciel. De nos jours, ce besoin se fait d'autant plus ressentir que les données accessibles aux scientifiques sont nombreuses et complexes.

Cependant, plonger dans le monde fascinant des analyses spatiales sous R peut s'avérer déroutant, voire décourageant pour le néophyte. C'est de ce besoin qu'est née l'idée d'une formation en analyses spatiales sous R. Ce document est une synthèse d'un enseignement

donné aux membres du Groupe de recherche sur les environnements nordiques BORÉAS à l'Université du Québec à Rimouski à l'hiver 2013. Il propose un survol des fonctionnalités de R dans le domaine des analyses spatiales. Dans sa présente version, l'enseignement est divisé en trois chapitres. Le premier chapitre détaille la structure des données spatiales sous R (notamment sous le *package* `sp`). Au travers de ce chapitre, nous apprendrons à importer/exporter une couche spatiale sous forme vectorielle et matricielle, ainsi qu'à modifier son système de coordonnées. Dans le second chapitre, nous verrons comment modifier la géométrie de couches vectorielles (jointure, dissolution, intersection, découpage, etc.). Enfin, le dernier chapitre mettra l'emphase sur la représentation cartographique d'objets spatiaux sous R.

Il ne s'agit en aucun cas d'un cours de programmation, ni même d'un cours d'analyses spatiales. Il existe de très bons ouvrages dédiés à ces thématiques. De même, de nombreux aspects ont volontairement été écartés, car un seul ouvrage ne pourrait faire le tour des possibilités de R dans ce domaine qui nous intéresse aujourd'hui. Une prochaine version de ce document sera bientôt disponible. Celle-ci se verra augmentée d'un chapitre supplémentaire, intitulé « Statistiques spatiales » et traitera plus précisément des notions d'autocorrélation spatiale, d'interpolation spatiale et de modélisation spatiale.

Pour terminer, je tiens à remercier le Groupe de recherche sur les environnements nordiques BORÉAS, et plus particulièrement Geneviève Allard, pour m'avoir offert l'opportunité de donner cette formation. Merci pour le support financier et logistique. Je remercie également Kévin « Jah » Cazelles et Frédéric Lesmerises, tous deux étudiants au doctorat en biologie, pour leurs ateliers de travail sous R, dont les nombreux codes m'ont fourni un bon point de départ pour la rédaction de ce document. Merci aussi à Hedvig Nenzén pour ses conseils sur LaTeX, `Sweave` et `knitr`. Merci enfin aux nombreux programmeurs R du monde entier qui partagent leurs travaux librement sur le Net. La connaissance doit être libre, gratuite et accessible à tous ! R, une philosophie à lui seul ! ! !

Nicolas Casajus,
le 16 juin 2013

Table des matières

Liste des figures

Chapitre 1

Les objets spatiaux sous R

1.1 Les packages incontournables

L'arrivée des analyses spatiales sous R s'est accompagnée du développement d'une centaine de *packages* spécialisés (voir la page web suivante pour une liste assez exhaustive : `http://cran.r-project.org/web/views/Spatial.html`). Manipuler des objets spatiaux sous R nécessite donc l'installation préalable de certains d'entre eux. Le *package* `sp` est le plus populaire d'entre eux puisqu'il regroupe les fonctions et les objets spatiaux de base sur lesquels s'appuient de nombreux autres *packages*. Voici la liste des *packages* qui seront utilisés dans ce document :

- `dismo` : présente une interface avec Google Maps et permet d'accéder à de nombreux jeux de données en ligne ;
- `gstat` : offre des outils indispensables pour l'interpolation spatiale (krigeage) ;
- `maptools` : permet de manipuler les objets spatiaux en créant une classe particulière d'objets spatiaux (*planar point pattern*) ;
- `raster` : offre de nombreuses fonctions permettant de lire et de manipuler les objets de type raster ;
- `rgeos` : permet de manipuler la géométrie des objets spatiaux ;
- `rgdal` : *package* permettant d'importer/exporter de nombreux formats d'objets spatiaux (raster de type grd, GeoTiff, shapefiles ESRI, fichiers KML, etc.) ;
- `sp` : *package* de base définissant des classes d'objets spatiaux et de nombreuses fonctions permettant de les manipuler ;
- `spatstat` : offre de nombreux outils pour réaliser des statistiques spatiales ;
- `spdep` : idéal pour étudier l'autocorrélation spatiale, mais aussi pour ce tout qui touche à la modélisation spatiale.

Sans plus attendre, installons tous ces *packages*, ainsi que tous ceux dont ils dépendent avec la commande `install.packages()`.

```
> # Telechargement des packages
> lib <- c("classInt", "dismo", "fields", "gstat", "maptools",
+     "raster", "pgirmess", "rgdal", "rgeos", "sp", "spatstat", "spdep")
> for (i in 1:length(lib)) install.packages(lib[i], dependencies = T)
```

1.2 Création d'objets spatiaux sous R

Avant d'importer des objets spatiaux déjà existants, nous allons nous intéresser à étudier la structure sous laquelle R gère les objets spatiaux. De nombreuses classes d'objets spatiaux ont été développées. Nous nous limiterons aux classes d'objets spatiaux introduites par le *package* `sp` car elles sont parmi les plus performantes. Ce *package* introduit un type particulier d'objets sous R : les objets sp, qui, selon leur géométrie, prendront différentes dénominations (Table 1.1). Ainsi, un ensemble de points géoréférencés sans table d'attributs sera stocké sous forme d'un `SpatialPoints`.

Objet sp	Géométrie
`SpatialPoints`	Points géoréférencés sans table d'attributs
`SpatialPointsDataFrame`	Points géoréférencés avec table d'attributs
`SpatialLines`	Lignes géoréférencées sans table d'attributs
`SpatialLinesDataFrame`	Lignes géoréférencées avec table d'attributs
`SpatialPolygons`	Polygones géoréférencés sans table d'attributs
`SpatialPolygonsDataFrame`	Polygones géoréférencés avec table d'attributs
`SpatialGrid`	Grille régulière sans table d'attributs
`SpatialGridDataFrame`	Grille régulière avec table d'attributs
`SpatialPixels`	Grille régulière sans table d'attributs
`SpatialPixelsDataFrame`	Grille régulière avec table d'attributs

TABLE 1.1 – Nomenclature des objets sp

Le *package* `sp` permet de créer ou de convertir en objet sp différentes géométries : des points, des lignes, des polygones ou encore des grilles. En général, chaque objet sp est composé de différentes parties : les `slots`. Chaque slot contient une information particulière (coordonnées géographiques, table d'attributs, système de coordonnées, étendue spatiale, etc.). L'accès à un slot d'un objet sp se fera à l'aide de l'opérateur `@` (`objet@slot`). Commençons par charger ce *package* dans notre session de travail.

```
> # Chargement du package 'sp'
> library(sp)
```

Poursuivons en définissant le chemin d'accès à notre répertoire de travail contenant les données dont nous aurons besoin dans ce document.

```
> # Definition du chemin d'acces au repertoire de travail
> root <- "C:/Users/Nicolas/Documents/RFormation/data"
> # Affichage du contenu du repertoire
> dir(root)

## [1] "grd_A.txt"     "grd_B.txt"     "statesAUS.txt" "townsAUS.txt"
```

1.2.1 Création d'objets ponctuels

Intéressons-nous tout d'abord à la structure des objets spatiaux ponctuels, c.-à-d. à un ensemble de points géoréférencés. Ce sont les objets spatiaux dont la structure est la

plus simple à aborder. Nous allons importer sous R le fichier « townsAUS.txt » qui donne la position géographique des capitales régionales de l'Australie.

```
> # Importation des donnees
> (tab <- read.delim(file = paste(root, "/townsAUS.txt", sep = "")))

##           Ville Longitude Latitude Population Etat
## 1      Adelaide     138.6   -34.93    1187466   AM
## 2 Alice Springs     133.9   -23.70      27481   TN
## 3      Brisbane     153.0   -27.47    2043185   QL
## 4      Canberra     149.1   -35.31     323056  TCA
## 5        Hobart     147.3   -42.89     202138   TA
## 6     Melbourne     145.0   -37.81    4009400   VI
## 7         Perth     115.9   -31.95    1658992   AO
## 8        Sydney     151.2   -33.86    4490662  NGS

> # Classe de l'objet
> class(tab)

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

Bien que les données comportent des coordonnées géographiques, pour l'heure elles sont stockées sous forme d'un `data.frame` non spatialisé. Nous allons devoir indiquer à R que ces données possèdent une information spatiale. Dans un premier temps, nous allons convertir ces données en objet spatial ponctuel sans table d'attributs, c.-à-d. un `SpatialPoints` (Table 1.1) en utilisant la fonction du même nom.

```
> # Conversion des donnees en objet spatial ponctuel
> (dat <- SpatialPoints(coords = tab[, 2:3]))

## SpatialPoints:
##      Longitude Latitude
## [1,]     138.6   -34.93
## [2,]     133.9   -23.70
## [3,]     153.0   -27.47
## [4,]     149.1   -35.31
## [5,]     147.3   -42.89
## [6,]     145.0   -37.81
## [7,]     115.9   -31.95
## [8,]     151.2   -33.86
## Coordinate Reference System (CRS) arguments: NA

> # Classe de l'objet cree
> class(dat)

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

La structure des données a changé et elles sont maintenant stockées sous forme d'un `SpatialPoints`. Nous voyons que plusieurs informations se rajoutent lors de l'affichage de l'objet, comme la classe de l'objet et le système de coordonnées (`Coordinate Reference System`, `CRS`). Nous avons mentionné précédemment que les objets sp stockaient ces différentes informations sous forme de slots. Pour connaître le nom des slots, nous utiliserons la fonction `slotNames()`.

```
> # Affichage du nom des slots
> slotNames(dat)

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

Notre objet spatial, de type `SpatialPoints`, contient trois slots différents :
- le slot `@coords` est une matrice à deux colonnes contenant les coordonnées géographiques de nos villes ;
- le slot `@bbox` est également une matrice à deux colonnes contenant l'étendue spatiale délimitant notre objet ;
- le slot `@proj4string` nous indique le système de coordonnées de l'objet spatial.

Pour afficher le contenu de ces trois slots, nous pouvons soit utiliser l'opérateur `@`, soit utiliser les fonctions associées à ces slots. Voyons cela de plus près.

Affichons tout d'abord les coordonnées géographiques. Nous allons procéder de deux manières différentes : en utilisant l'accès au slot `coords` via l'opérateur `@`, et en utilisant la fonction `coordinates()`.

```
> # Affichage des coordonnees geographiques
> dat@coords

##      Longitude Latitude
## [1,]     138.6   -34.93
## [2,]     133.9   -23.70
## [3,]     153.0   -27.47
## [4,]     149.1   -35.31
## [5,]     147.3   -42.89
## [6,]     145.0   -37.81
## [7,]     115.9   -31.95
## [8,]     151.2   -33.86

> coordinates(dat)

##      Longitude Latitude
## [1,]     138.6   -34.93
## [2,]     133.9   -23.70
## [3,]     153.0   -27.47
## [4,]     149.1   -35.31
## [5,]     147.3   -42.89
## [6,]     145.0   -37.81
## [7,]     115.9   -31.95
## [8,]     151.2   -33.86
```

Affichons ensuite l'étendue spatiale définie par notre objet. Dans ce cas, la fonction associée à ce slot est `bbox()`.

```
> # Affichage de l'etendue spatiale
> dat@bbox

##              min   max
## Longitude 115.86 153.0
## Latitude  -42.89 -23.7

> bbox(dat)

##              min   max
## Longitude 115.86 153.0
## Latitude  -42.89 -23.7
```

Pour terminer, affichons le système de coordonnées contenu dans le slot `@proj4string`. La fonction `proj4string()` permet également d'accéder à cette information.

```
> # Affichage du systeme de coordonnees
> dat@proj4string

## CRS arguments: NA

> proj4string(dat)

## [1] NA
```

Ce dernier slot retourne la valeur `NA` : notre objet spatial n'est défini par aucun système de coordonnées. Nous verrons plus loin comment en définir un.

Nous allons maintenant convertir notre `data.frame` initial en un objet spatial ponctuel possédant une table d'attributs : un `SpatialPointsDataFrame` (Table 1.1). Pour cela, nous allons utiliser la fonction du même nom dans laquelle nous allons spécifier la table d'attributs avec l'argument ***data***.

```
> # Conversion en objet spatial ponctuel avec une table d'attributs
> (dat <- SpatialPointsDataFrame(coords = tab[, 2:3], data = tab[,
+     c(1, 4, 5)]))

##           coordinates         Ville Population Etat
## 1     (138.6, -34.93)      Adelaide    1187466   AM
## 2 (133.877, -23.7022) Alice Springs      27481   TN
## 3 (153.033, -27.4667)      Brisbane    2043185   QL
## 4 (149.124, -35.3083)      Canberra     323056  TCA
## 5 (147.331, -42.8858)        Hobart     202138   TA
## 6 (144.968, -37.8142)     Melbourne    4009400   VI
## 7    (115.86, -31.95)         Perth    1658992   AO
## 8 (151.192, -33.8561)        Sydney    4490662  NGS
```

```
> # Classe de l'objet cree
> class(dat)

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

Là encore, la structure de nos données a changé, et on retrouve cette fois-ci toutes les informations contenues dans le `data.frame` initial. Il est important de remarquer que la première colonne de ce nouvel objet se nomme « Ville » (et non « coordinates »). Regardons les slots contenus dans ce `SpatialPointsDataFrame`.

```
> # Affichage des noms des slots
> slotNames(dat)

## [1] "data"        "coords.nrs"  "coords"      "bbox"
## [5] "proj4string"
```

Deux nouveaux `slots` ont été créés :
- le slot `@data` contient la table d'attributs stockée sous forme d'un `data.frame` ;
- le slot `@coords.nrs` contient la position des colonnes des coordonnées géographiques dans les données initiales (peu utile).

Affichons la table d'attributs avec l'opérateur `@`.

```
> # Affichage de la table d'attributs
> dat@data

##           Ville Population Etat
## 1      Adelaide    1187466   AM
## 2 Alice Springs      27481   TN
## 3      Brisbane    2043185   QL
## 4      Canberra     323056  TCA
## 5        Hobart     202138   TA
## 6     Melbourne    4009400   VI
## 7         Perth    1658992   AO
## 8        Sydney    4490662  NGS
```

La structure des objets spatiaux ponctuels sous R n'est pas plus compliquée que ça. Remarquons qu'elle n'est pas sans rappeler l'organisation d'un shapefile ESRI composé de plusieurs fichiers : le fichier « .dbf » (contenant la table d'attributs), le fichier « .prj » (contenant le système de coordonnées), le fichier « .shp » (contenant la géométrie), etc.

1.2.2 Création de polygones

Compliquons un peu les choses et regardons comment créer des polygones géoréférencés. Même si la logique est la même que pour les objets spatiaux ponctuels (séparation des différentes informations en slots), la création de polygones spatiaux est plus complexe et s'effectuera en plusieurs étapes successives :

– Conversion de coordonnées en polygone simple ;
– Création de polygone multiple ;
– Création d'un polygone spatial ;
– Rajout d'une table d'attributs.

Regardons cela dans le détail. L'objectif ici est de créer un objet spatial composé de trois polygones : le premier sera composé d'un seul polygone, le second sera formé d'un polygone avec un trou à l'intérieur, et le dernier sera composé de deux polygones.

Procédons pas à pas et créons tout d'abord cinq séries de coordonnées qui vont définir les sommets de cinq polygones (une série pour le premier polygone, deux séries pour le second, dont une définira le positionnement du trou, et deux séries pour le dernier polygone formé de deux polygones).

```
> # Creation de la premiere serie de coordonnees
> x1 <- c(439518.5, 433091.8, 455774.1, 476566.1, 476944.2, 459554.4,
+     439518.5)
> y1 <- c(8045280, 8031293, 8018439, 8026756, 8044902, 8054731,
+     8045280)
> (c1 <- data.frame(x1, y1))

##       x1      y1
## 1 439519 8045280
## 2 433092 8031293
## 3 455774 8018439
## 4 476566 8026756
## 5 476944 8044902
## 6 459554 8054731
## 7 439519 8045280

> # Creation de la seconde serie de coordonnees
> x2 <- c(444929.2, 417667.9, 501837.1, 499792.5, 444929.2)
> y2 <- c(8121306, 8078029, 8067465, 8109039, 8121306)
> c2 <- data.frame(x2, y2)
> # Creation de la troisieme serie de coordonnees
> x3 <- c(456530.1, 450481.5, 472785.8, 476566.1, 456530.1)
> y3 <- c(8101608, 8089510, 8087620, 8099717, 8101608)
> c3 <- data.frame(x3, y3)
> # Creation de la quatrieme serie de coordonnees
> x4 <- c(530802.2, 505585.5, 520579.3, 549544.4, 611223, 530802.2)
> y4 <- c(8115513, 8096771, 8067806, 8055538, 8084163, 8115513)
> c4 <- data.frame(x4, y4)
> # Creation de la cinquieme serie de coordonnees
> x5 <- c(549885.1, 556359.7, 549544.4, 540684.4, 537958.3, 549885.1)
> y5 <- c(8021121, 8025551, 8043612, 8043271, 8034752, 8021121)
> c5 <- data.frame(x5, y5)
```

Conversion en polygone simple

La première étape consiste à convertir chaque série de coordonnées en un polygone simple. Pour cela, nous allons utiliser la fonction `Polygon()`.

```
> # Conversion en polygones simples
> (p1 <- Polygon(coords = c1, hole = F))

## An object of class "Polygon"
## Slot "labpt":
## [1]  457052 8036322
##
## Slot "area":
## [1] 1.067e+09
##
## Slot "hole":
## [1] FALSE
##
## Slot "ringDir":
## [1] 1
##
## Slot "coords":
##          x1      y1
## [1,] 439519 8045280
## [2,] 459554 8054731
## [3,] 476944 8044902
## [4,] 476566 8026756
## [5,] 455774 8018439
## [6,] 433092 8031293
## [7,] 439519 8045280
##

> p2 <- Polygon(coords = c2, hole = F)
> p3 <- Polygon(coords = c3, hole = T)
> p4 <- Polygon(coords = c4, hole = F)
> p5 <- Polygon(coords = c5, hole = F)
```

L'argument ***hole*** indique si le polygone servira à définir le trou d'un autre polygone (**T**) ou non (**F**). La structure des séries de coordonnées se présente maintenant sous forme d'un `Polygon` possédant plusieurs slots. Affichons le nom de ces slots.

```
> # Affichage du nom des slots de p1
> slotNames(p1)

## [1] "labpt"   "area"    "hole"    "ringDir" "coords"
```

Cinq slots ont été créés. Décrivons ceux qui nous serons utiles par la suite :
- le slot `@labpt` donne les coordonnées du centre du polygone ;
- le slot `@hole` indique si le polygone servira à définir le trou d'un autre ;

– le slot `@coords` contient les coordonnées du polygone.

Affichons les coordonnées du premier polygone.

```
> # Acces aux coordonnees
> p1@coords

##          x1      y1
## [1,] 439519 8045280
## [2,] 459554 8054731
## [3,] 476944 8044902
## [4,] 476566 8026756
## [5,] 455774 8018439
## [6,] 433092 8031293
## [7,] 439519 8045280
```

Création de polygone multiple

La seconde étape consiste à créer des polygones multiples, c.-à-d. à regrouper certains polygones simples dans un même polygone. Cette étape est obligatoire, même dans le cas d'un polygone formé d'un seul polygone. C'est lors de cette conversion qu'un identifiant sera donné à chaque polygone multiple. Procédons à cette transformation en utilisant la fonction `Polygons()` (notons la présence d'un « s »).

```
> # Conversion en polygones multiples
> P1 <- Polygons(srl = list(p1), ID = "PolygA")
> P2 <- Polygons(srl = list(p2, p3), ID = "PolygB")
> P3 <- Polygons(srl = list(p4, p5), ID = "PolygC")
```

Détaillons ces trois lignes de code : le premier polygone multiple *P1* est formé du polygone simple p1, le polygone multiple *P2* regroupe les polygones simples *p2* et *p3* (avec *p3* délimitant un espace vide à l'intérieur de *p2*) et le polygone multiple *P3* est composé des deux polygones simples *p4* et *p5*.

Affichons le nom des slots du polygone *P2*.

```
> # Affichage du nom des slots de P2
> slotNames(P2)

## [1] "Polygons"  "plotOrder" "labpt"     "ID"        "area"
```

Là encore, cinq slots ont été créés. Les deux qui vont nous intéresser sont `@Polygons` et `@ID`. Ce dernier contient l'identifiant (unique) du polygone multiple.

```
> # Affichage l'identifiant d'un objet de type Polygons
> P2@ID

## [1] "PolygB"
```

Intéressons-nous plus précisément au slot `@Polygons`.

```
> # Classe du slot Polygons
> class(P2@Polygons)

## [1] "list"

> # Nombre d'elements contenus dans cette liste
> length(P2@Polygons)

## [1] 2

> # Classe du premier element
> class(P2@Polygons[[1]])

## [1] "Polygon"
## attr(,"package")
## [1] "sp"

> # Nom de ses slots
> slotNames(P2@Polygons[[1]])

## [1] "labpt"   "area"    "hole"    "ringDir" "coords"
```

Le slot `@Polygons` est une liste d'objets de type `Polygon`, c.-à-d. une liste de polygones simples. Dans notre cas, *P2* contient deux polygones simples. En affichant le nom des slots contenu dans le premier élément (le premier polygone simple), on retrouve la structure vue précédemment.

Regardons comment accéder aux coordonnées du premier polygone simple (*p2*) contenu dans le second polygone multiple (*P2*).

```
> # Acces aux coordonnees
> P2@Polygons[[1]]@coords

##          x2      y2
## [1,] 444929 8121306
## [2,] 499793 8109039
## [3,] 501837 8067465
## [4,] 417668 8078029
## [5,] 444929 8121306
```

Création d'un polygone spatial

Maintenant, on souhaite regrouper ces trois polygones multiples en un seul objet spatial. Pour cela, nous allons créer un objet de classe `SpatialPolygons` (Table 1.1).

```
> # Conversion en polygone spatial
> SP <- SpatialPolygons(Srl = list(P1, P2, P3))
```

Regardons le contenu de ce nouvel objet spatial.

```
> # Nom des slots de SP
> slotNames(SP)

## [1] "polygons"    "plotOrder"   "bbox"        "proj4string"
```

On retrouve ici une structure en slots que nous avons vue précédemment avec les objets spatiaux ponctuels. Le système de coordonnées est stocké dans `@proj4string`, l'étendue spatiale dans `@bbox`. Intéressons-nous plus précisément au slot `@polygons`.

```
> # Classe du slot polygons
> class(SP@polygons)

## [1] "list"

> # Nombre d'elements contenus dans cette liste
> length(SP@polygons)

## [1] 3

> # Classe du premier element
> class(SP@polygons[[1]])

## [1] "Polygons"
## attr(,"package")
## [1] "sp"

> # Nom de ses slots
> slotNames(SP@polygons[[1]])

## [1] "Polygons"  "plotOrder" "labpt"     "ID"        "area"
```

Notre objet spatial se structure donc de la manière suivante : le `SpatialPolygons` contient une liste de trois `Polygons` (polygones multiples) contenant chacun une liste de `Polygon` (polygones simples), lesquels contiennent les coordonnées qui les délimitent.

Ainsi, pour accéder aux coordonnées du premier polygone simple contenu dans le second polygone multiple, nous devons écrire :

```
> # Acces aux coordonnees
> SP@polygons[[2]]@Polygons[[1]]@coords

##          x2      y2
## [1,] 444929 8121306
## [2,] 499793 8109039
## [3,] 501837 8067465
## [4,] 417668 8078029
## [5,] 444929 8121306
```

Rajout d'une table d'attributs

Enfin, à ce `SpatialPolygons` nous pouvons ajouter une table d'attributs. Construisons tout d'abord une table d'attributs fictive.

```
> # Creation des champs de la table
> Info <- c("Simple", "Hole", "Double")
> Value <- c(342, 123, 546)
> Frequence <- c(0.34, 0.12, 0.54)
> # Creons maintenant la table
> (mat <- data.frame(Info, Value, Frequence))

##     Info Value Frequence
## 1 Simple   342      0.34
## 2   Hole   123      0.12
## 3 Double   546      0.54
```

Remarque importante : il est nécessaire que les noms des lignes de la table d'attributs correspondent aux identifiants des polygones multiples dans l'objet spatial. De plus, l'ordre des lignes doit également correspondre à l'ordre des polygones multiples. Renommons donc les lignes de la table :

```
> # Attribution d'un nom aux lignes de la table
> rownames(mat) <- c("PolygA", "PolygB", "PolygC")
> mat

##          Info Value Frequence
## PolygA Simple   342      0.34
## PolygB   Hole   123      0.12
## PolygC Double   546      0.54
```

Finalement, ajoutons cette table dans l'objet sp afin de créer, en accord avec la Table 1.1, un `SpatialPolygonsDataFrame`.

```
> # Rajout d'une table d'attributs
> SPDF <- SpatialPolygonsDataFrame(Sr = SP, data = mat)
```

Affichons le nom des slots afin d'y voir plus clair.

```
> # Nom des slots du polygone spatial avec table d'attributs
> slotNames(SPDF)

## [1] "data"        "polygons"    "plotOrder"   "bbox"
## [5] "proj4string"
```

Ainsi, la structure n'a pas changé par rapport au `SpatialPolygons`, mise à part l'addition d'un nouveau slot : `@data` qui contient désormais la table d'attributs.

```
> # Acces a la table d'attributs
> SPDF@data

##          Info Value Frequence
## PolygA Simple   342      0.34
## PolygB   Hole   123      0.12
## PolygC Double   546      0.54
```

Anticipons légèrement sur le chapitre consacré à la cartographie et représentons notre polygone spatial. Nous allons identifier chacun de nos trois polygones multiples par une couleur différente.

```
> # Visualisation du polygone spatial
> par(mar = c(0, 0, 0, 0))
> plot(SPDF, col = c("lightgray", "darkgray", "black"), axes = F)
```

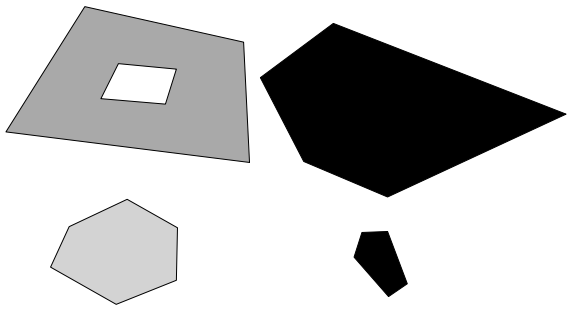

FIGURE 1.1 – Représentation d'un `SpatialPolygons`

1.2.3 Création de lignes

La structure d'objets spatiaux de type *polyligne* suit la même philosophie que celle des polygones. Leur création s'effectuera en plusieurs étapes :

- Conversion des coordonnées en ligne simple ;
- Création de ligne multiple ;
- Création d'une ligne spatiale ;
- Rajout d'une table d'attributs.

Regardons cela dans le détail en construisant deux lignes, dont une sera formée de deux segments. Définissons les coordonnées de ces segments.

```
> # Creation de la premiere serie de coordonnees
> x1 <- c(439518.5, 433091.8, 455774.1, 476566.1)
> y1 <- c(8045280, 8031293, 8018439, 8026756)
> (c1 <- data.frame(x1, y1))

##       x1      y1
## 1 439519 8045280
## 2 433092 8031293
## 3 455774 8018439
## 4 476566 8026756

> # Creation de la seconde serie de coordonnees
> x2 <- c(444929.2, 417667.9, 501837.1, 499792.5)
> y2 <- c(8121306, 8078029, 8067465, 8109039)
> c2 <- data.frame(x2, y2)
> # Creation de la seconde serie de coordonnees
> x3 <- c(450192.5, 430904.9, 493701.7, 493028.8)
> y3 <- c(8120872, 8082746, 8074223, 8109210)
> c3 <- data.frame(x3, y3)
```

Conversion en ligne simple

La première étape consiste à convertir chaque série de coordonnées en une ligne simple. Pour cela, nous allons utiliser la fonction `Line()`.

```
> # Conversion en lignes simples
> (l1 <- Line(c1))

## An object of class "Line"
## Slot "coords":
##          x1      y1
## [1,] 439519 8045280
## [2,] 433092 8031293
## [3,] 455774 8018439
## [4,] 476566 8026756
##

> l2 <- Line(c2)
> l3 <- Line(c3)
```

La structure des séries de coordonnées se présente maintenant sous forme d'un objet de type `Line` qui n'est pas sans rappeler les objets de type `Polygon`. Affichons le nom de ses slots.

```
> # Nom des slots
> slotNames(l1)

## [1] "coords"
```

Création de ligne multiple

La seconde étape consiste à créer des lignes multiples, c.-à-d. à regrouper certaines lignes simples (segments) dans une même ligne. Cette étape est obligatoire, même dans le cas d'une ligne formée d'un seul segment. C'est lors de cette conversion qu'un identifiant sera donné à chaque ligne multiple. Procédons à cette transformation en utilisant la fonction `Lines()` (notons là encore la présence d'un « s »).

```
> # Conversion en lignes multiples
> L1 <- Lines(list(l1), ID = "LineA")
> (L2 <- Lines(list(l2, l3), ID = "LineB"))

## An object of class "Lines"
## Slot "Lines":
## [[1]]
## An object of class "Line"
## Slot "coords":
##          x2      y2
## [1,] 444929 8121306
## [2,] 417668 8078029
## [3,] 501837 8067465
## [4,] 499793 8109039
##
##
## [[2]]
## An object of class "Line"
## Slot "coords":
##          x3      y3
## [1,] 450193 8120872
## [2,] 430905 8082746
## [3,] 493702 8074223
## [4,] 493029 8109210
##
##
##
## Slot "ID":
## [1] "LineB"
##
```

La première ligne multiple *L1* est formée d'un seul segment (*l1*), alors que la seconde, *L2*, est composée de deux segments : *l2* et *l3*.

Affichons le nom des slots de la ligne multiple *L2*.

```
> # Nom des slots
> slotNames(L2)

## [1] "Lines" "ID"
```

Les objets de type `Lines` sont composés de deux slots : `@Lines` et `@ID`. Bien que leur structure ressemble à celle des `Polygons`, leur contenu est simplifié.

```
> # Classe du slot Lines
> class(L2@Lines)

## [1] "list"

> # Nombre d'elements dans la liste
> length(L2@Lines)

## [1] 2

> # Classe du premier element
> class(L2@Lines[[1]])

## [1] "Line"
## attr(,"package")
## [1] "sp"

> # Nom de ses slots
> slotNames(L2@Lines[[1]])

## [1] "coords"
```

Regardons comment accéder aux coordonnées du premier segment de la seconde ligne multiple.

```
> # Acces aux coordonnees
> L2@Lines[[1]]@coords

##          x2      y2
## [1,] 444929 8121306
## [2,] 417668 8078029
## [3,] 501837 8067465
## [4,] 499793 8109039
```

Création d'une ligne spatiale

Nous allons maintenant regrouper ces deux lignes multiples en un seul objet spatial. Pour cela, nous allons créer un objet de classe `SpatialLines` (Table 1.1) avec la fonction du même nom.

```
> # Conversion en polygone spatial
> SL <- SpatialLines(LinesList = list(L1, L2))
```

Regardons le contenu de cet objet spatial.

```
> # Nom des slots de SL
> slotNames(SL)

## [1] "lines"       "bbox"        "proj4string"
```

On retrouve ici une structure en slots qui commence à nous être familière. L'étendue spatiale de notre objet est stockée dans le slot `@bbox` et le système de coordonnées dans lequel il est défini se trouve dans `@proj4string`. Regardons de plus près le slot `@lines`.

```
> # Classe du slot lines
> class(SL@lines)

## [1] "list"

> # Nombre d'elements contenus dans cette liste
> length(SL@lines)

## [1] 2

> # Classe du premier element
> class(SL@lines[[1]])

## [1] "Lines"
## attr(,"package")
## [1] "sp"

> # Nom de ses slots
> slotNames(SL@lines[[1]])

## [1] "Lines" "ID"
```

Ainsi, pour accéder aux coordonnées du premier segment de la seconde ligne multiple, nous procèderons de la manière suivante :

```
> SL@lines[[2]]@Lines[[1]]@coords

##          x2      y2
## [1,] 444929 8121306
## [2,] 417668 8078029
## [3,] 501837 8067465
## [4,] 499793 8109039
```

Rajout d'une table d'attributs

Pour terminer, nous pouvons rajouter une table d'attributs à ce `SpatialLines`. Commençons par en créer une avec les mêmes contraintes que pour les `SpatialPolygons` (l'ordre et le nom des lignes de la table doivent être identiques à l'ordre et aux identifiants des lignes multiples).

```
> # Creation des champs de la table
> Info <- c("Enil1", "Enil2")
> Value <- c(342, 123)
> Frequence <- c(0.74, 0.26)
> # Creons maintenant la table
```

```
> (mat <- data.frame(Info, Value, Frequence))

##    Info Value Frequence
## 1 Enil1   342      0.74
## 2 Enil2   123      0.26

> rownames(mat) <- c("LineA", "LineB")
```

Rajoutons cette table d'attributs à notre `SpatialLines`, afin de le convertir en un `SpatialLinesDataFrame` avec la fonction du même nom.

```
> # Rajout d'une table d'attributs
> SLDF <- SpatialLinesDataFrame(sl = SL, data = mat)
```

Affichons le nom des `slots` afin d'y voir plus clair.

```
> # Nom des slots de l'objet spatial avec table d'attributs
> slotNames(SLDF)

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

La structure n'a pas changé par rapport au `SpatialLines`, mise à part l'addition d'un nouveau slot : `@data` qui contient maintenant la table d'attributs. Affichons cette table :

```
> # Acces a la table d'attributs
> SLDF@data

##        Info Value Frequence
## LineA Enil1   342      0.74
## LineB Enil2   123      0.26
```

1.2.4 Création de grilles

Les trois classes d'objet sp que nous venons de voir (points, polylignes et polygones) sont des objets vectoriels, c.-à-d. des objets composés d'éléments géométriques individuels. Chacun de ces éléments possède des caractéristiques propres qui doivent être stockées. Nous allons voir maintenant comment créer des objets matriciels (plus connus sous le nom de *raster*), c.-à-d. des grilles composées de cellules régulièrement espacées. Deux approches peuvent être employées pour créer ce genre d'objets : par pixellisation d'objets ponctuels ou en générant une grille à partir d'une information de base (origine, résolution et nombre de cellules).

Pixellisation d'objets ponctuels

Imaginons que nous ayons une série de points géoréférencés régulièrement répartis dans l'espace et définissant le centre de cellules. L'idée est de convertir ces points en un ensemble de pixels formant une grille matricielle. Pour ce faire, nous allons utiliser la fonction `SpatialPixels()` du *package* `sp` (Table 1.1).

Regardons cela de plus près, et importons tout d'abord un fichier donnant les coordonnées d'un ensemble de points.

```
> # Importation des donnees
> pts <- read.delim(paste(root, "/grd_A.txt", sep = ""))
> # Affichons les premieres lignes
> head(pts)

##       x     y
## 1 112.5 -43.5
## 2 113.5 -43.5
## 3 114.5 -43.5
## 4 115.5 -43.5
## 5 116.5 -43.5
## 6 117.5 -43.5
```

Avant d'utiliser la fonction citée ci-dessus, nous devons convertir ces points en un objet spatial ponctuel.

```
> # Conversion en SpatialPoints
> pts <- SpatialPoints(coords = pts)
> # Conversion en SpatialPixels
> SPx <- SpatialPixels(points = pts)
> # Classe de l'objet cree
> class(SPx)

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

Regardons comment se structure cet objet.

```
> # Nom des slots
> slotNames(SPx)

## [1] "grid"        "grid.index"  "coords"      "bbox"
## [5] "proj4string"
```

Ici, on retrouve des slots que nous avons déjà vus : `@coords`, `@bbox` et `@proj4string`. Cependant, on remarque l'apparition de deux nouveaux éléments, spécifiques aux objets sp de type `SpatialPixels` et `SpatialGrid` (que nous verrons plus tard).

Le slot `@grid.index` donne l'identifiant des différentes cellules créées. Intéressons-nous maintenant au dernier slot : `@grid`.

```
> # Affichage du slot 'grid'
> SPx@grid

##                       x     y
## cellcentre.offset 112.5 -43.5
## cellsize            1.0   1.0
## cells.dim          43.0  38.0
```

```
> # Classe de l'element
> class(SPx@grid)

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

Ce slot est un des plus importants car c'est lui qui permet de caractériser la grille. Il se présente sous la forme d'un objet de type `GridTopology` qui contient trois attributs (stockés sous forme de slots) :

- le slot `@cellcentre.offset` correspond aux coordonnées du centre de la cellule en bas à gauche (sud-ouest) ;
- le slot `@cellsize` correspond à la résolution en longitude et en latitude de la grille ;
- le slot `@cells.dim` correspond au nombre de cellules selon les deux directions géographiques.

Simplement avec ces trois informations, nous serions capables de produire une grille rectangulaire complète sans aucun besoin de points. Nous verrons cela plus loin dans cette section. En attendant, visualisons cette grille.

```
> # Visualisation de la grille
> par(mar = c(0, 0, 0, 0))
> plot(SPx)
```

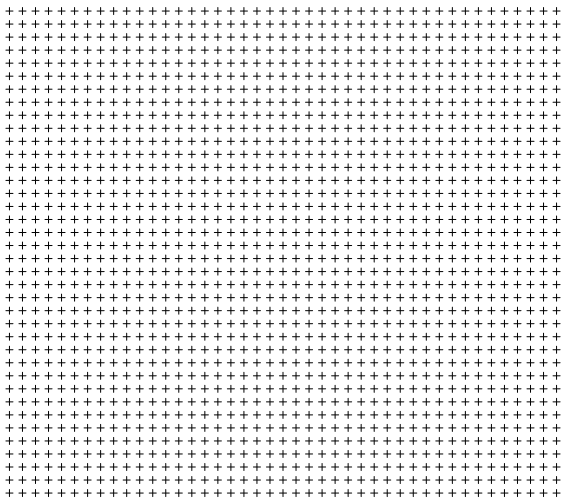

FIGURE 1.2 – Représentation d'un `SpatialPixels` complet

Il est également possible de rajouter une table d'attributs à ce genre d'objet, notamment en utilisant la fonction `SpatialPixelsDataFrame()`.

Nous venons de créer une grille rectangulaire complète à partir de points géoréférencés. Cependant, la principale utilité de la fonction `SpatialPixels()` est de permettre la construction d'une grille à partir de points ne formant pas une grille rectangulaire complète (mais tout de même espacés régulièrement dans l'espace).

Pour illustrer cet aspect, importons un nouveau jeu de données, lequel contient un échantillon des points précédents et pixellisons-les.

```
> # Importation de nouvelles donnees
> pts <- read.delim(paste(root, "/grd_B.txt", sep = ""))
> # Conversion en SpatialPoints
> pts <- SpatialPoints(coords = pts)
> # Conversion en SpatialPixels
> SP2 <- SpatialPixels(points = pts)
```

Représentons cette grille incomplète.

```
> par(mar = c(0, 0, 0, 0))
> # Representation de la grille incomplete
> plot(SP2)
```

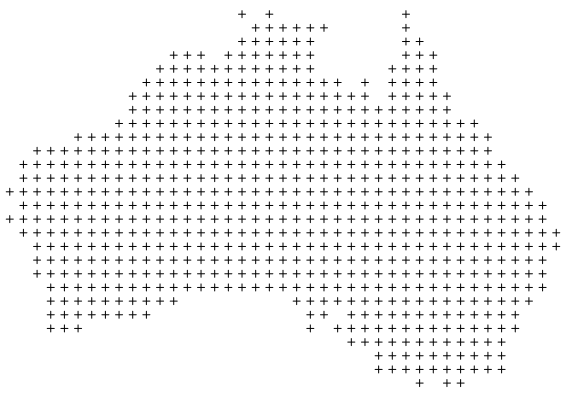

FIGURE 1.3 – Représentation d'un `SpatialPixels` incomplet

Génération automatique d'une grille

Nous venons de voir comment créer un objet matriciel en pixellisant des points géoréférencés régulièrement répartis dans l'espace et définissant une grille rectangulaire complète ou incomplète. Voyons maintenant comment définir une grille avec une information initiale minimale (c.-à-d. sans points à pixelliser).

Précédemment, nous avons vu que l'objet de classe `GridTopology` contenait l'information de base de notre grille régulière rectangulaire complète. Nous allons donc nous servir de cette information pour générer une grille complète sans utiliser de quelconques coordonnées.

Reproduisons la grille complète précédente au moyen de la fonction `GridTopology()`.

```
> # Coordonnees de l'origine
> cc <- c(112.5, -43.5)
> # Resolution des cellules
> cs <- c(1, 1)
> # Nombre de cellules
> cd <- c(43, 38)
> # Creation du GridTopology
> grd <- GridTopology(cellcentre.offset = cc, cellsize = cs, cells.dim =
cd)
```

Créons maintenant une grille régulière complète avec la fonction `SpatialGrid()`.

```
> # Creation de la grille
> grd <- SpatialGrid(grd)
> # Nom des slots
> slotNames(grd)

## [1] "grid"        "bbox"        "proj4string"
```

On remarque que la structure de l'objet est la même que précédemment, sauf que les slots relatifs aux coordonnées des cellules ont disparu : c'est un avantage de travailler avec les `SpatialGrid` : l'information stockée est minimale, ce qui peut être intéressant dans le cas de fichiers volumineux. De la même manière, il est possible de rajouter une table d'attributs à cet objet matriciel ; pour ce faire, on utilisera la fonction `SpatialGridDataFrame()`.

```
> par(mar = c(0, 0, 0, 0))
> # Representation de la grille
> plot(grd)
```

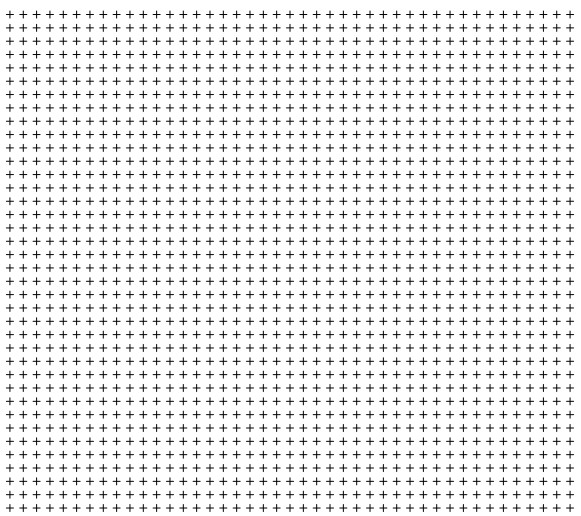

FIGURE 1.4 – Représentation d'un `SpatialGrid`

1.3 Import/export d'objets spatiaux

Nous venons de voir comment créer des objets spatiaux de type vectoriel et matriciel sous R à l'aide du *package* `sp`. Ceci nous a permis d'étudier leur structure et la façon d'accéder aux différentes informations qu'ils contiennent (coordonnées, table d'attributs, système de coordonnées, etc.). Cependant, dans la plupart des cas, nous nous contenterons de manipuler des objets spatiaux déjà existants.

Plusieurs *packages* permettent de lire et d'écrire des objets vectoriels et/ou matriciels. Le *package* `maptools` p. ex., permet de lire des fichiers de type vectoriel (avec la fonction `readShapeSpatial()`) ainsi que des rasters (fonction `readAsciiGrid()`). Cependant, il présente l'inconvénient majeur de ne pas pouvoir lire le fichier décrivant le système de coordonnées (fichier « .prj » d'un shapefile ESRI). De plus, il ne supporte que peu de formats matriciels (p. ex. ESRI Arc ASCII Grid).

Heureusement, une solution a été apportée : le *package* `rgdal`, développé entre autres par Roger Bivand, le créateur du *package* `sp`, et grand contributeur aux analyses spatiales sous R. Le *package* `rgdal` communique avec la bibliothèque libre GDAL (*Geospatial Data Abstraction Library*) qui permet de lire et de manipuler un très grand nombre de formats matriciels (GeoTIFF, ASCII Grid, etc.) au moyen de différents langages de programmation, dont le langage R. Sa sous-bibliothèque OGR (*OpenGIS Simple Features Reference Implementation*) permet quant à elle la manipulation de nombreux formats vectoriels (ESRI Shapefile, MapInfo, KML, etc.). Les fonctions `gdalDrivers()` et `ogrDrivers()` permettent de connaître respectivement les formats de données matricielles et vectorielles supportés. Sans plus attendre, explorons les possibilités de ce *package*.

1.3.1 Couches vectorielles

Nous allons tout d'abord voir comment manipuler des données vectorielles. Pour cela, nous allons télécharger des données fournissant les limites administratives de l'Australie sous un format « ESRI Shapefile ». Le site *Global Administrative Area* (`http://gadm.org`) met à la disposition du public des données SIG en libre téléchargement.

Téléchargeons ces données vectorielles à l'aide de R. Créons tout d'abord un répertoire « Australia » dans notre répertoire de travail actuel qui recueillera les données téléchargées.

```
> # Creation d'un repertoire
> dir.create(paste(root, "/Australia", sep = ""), showWarnings = F)
> # Verification
> dir(root)

## [1] "Australia"     "grd_A.txt"     "grd_B.txt"     "statesAUS.txt"
## [5] "townsAUS.txt"
```

Procédons maintenant au téléchargement.

```
> # Lien du telechargement
> link <- "http://gadm.org/data/shp/AUS_adm.zip"
> # Destination du fichier sur le disque
> dest <- paste(root, "/Australia/AUS_adm.zip", sep = "")
> # Telechargement
> download.file(url = link, destfile = dest, method = "internal")
```

```
> # Verification
> dir(paste(root, "/Australia", sep = ""))

## [1] "AUS_adm.zip"
```

Les données téléchargées sont stockées sous forme d'une archive ZIP. La commande R `unzip()` permet de décompresser les fichiers contenus une archive ZIP. Voyons cela.

```
> # Decompression de l'archive telechargee
> unzip(zipfile = dest, exdir = paste(root, "/Australia", sep = ""))
> # Verification
> dir(paste(root, "/Australia", sep = ""))

##  [1] "AUS_adm.zip"    "AUS_adm0.dbf"   "AUS_adm0.prj"
##  [4] "AUS_adm0.sbn"   "AUS_adm0.sbx"   "AUS_adm0.shp"
##  [7] "AUS_adm0.shx"   "AUS_adm1.dbf"   "AUS_adm1.prj"
## [10] "AUS_adm1.sbn"   "AUS_adm1.sbx"   "AUS_adm1.shp"
## [13] "AUS_adm1.shx"   "AUS_adm2.dbf"   "AUS_adm2.prj"
## [16] "AUS_adm2.sbn"   "AUS_adm2.sbx"   "AUS_adm2.shp"
## [19] "AUS_adm2.shx"   "AUS_readme.txt"
```

Il ne nous reste plus qu'à charger le *package* `rgdal` dans notre session de travail.

```
> # Chargement du package 'rgdal'
> library(rgdal)
```

Importation d'une couche vectorielle

Nous allons voir comment importer des données vectorielles en ouvrant le fichier « AUS_adm1 », qui donne les limites administratives des états/territoires australiens. Pour cela, nous allons utiliser la fonction `readOGR()` qui prend deux arguments principaux :

- ***dsn*** qui correspond, dans le cas de données type « ESRI Shapefile » au répertoire contenant les fichiers ;
- ***layer*** qui correspond au nom du fichier sans extension.

Regardons cela de plus près.

```
> # Repertoire contenant les donnees
> dest <- paste(root, "/Australia", sep = "")
> # Importation de la couche
> aus1 <- readOGR(dsn = dest, layer = "AUS_adm1", verbose = F)
```

Regardons comment sont stockées ces données sous R.

```
> # Classe de l'objet aus1
> class(aus1)
```

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

> # Nom des slots de aus1
> slotNames(aus1)

## [1] "data"        "polygons"    "plotOrder"   "bbox"
## [5] "proj4string"
```

Ce shapefile est donc structuré sous forme de polygones spatiaux possédant une table d'attributs (Section 1.2.2). Représentons graphiquement l'objet vectoriel « AUS_adm1 ».

```
> # Representation cartographique
> par(mar = c(0, 0, 0, 0))
> plot(aus1, col = "gray")
```

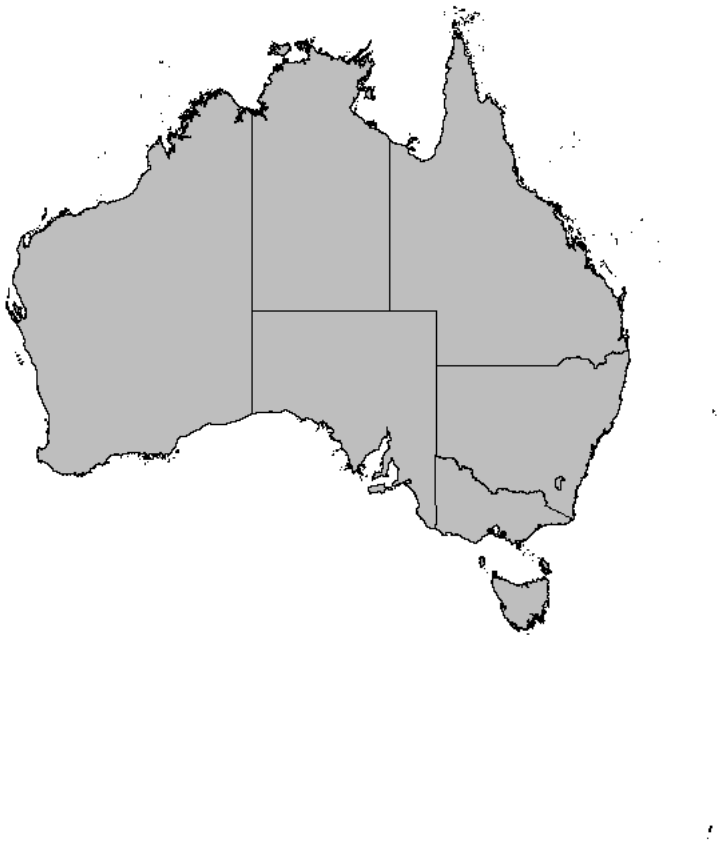

FIGURE 1.5 – Représentation d'un objet vectoriel

Exportation d'une couche vectorielle

Avant de voir comment exporter un objet vectoriel, nous allons sélectionner une partie de ces polygones. Pour ce faire, nous allons anticiper légèrement sur le chapitre suivant

et extraire tous les polygones correspondant à l'état australien du Queensland et créer un nouvel objet spatial (extraction par attributs).

```
> # Identification de la position du Queensland dans la table
> pos <- which(aus1@data[, "NAME_1"] == "Queensland")
> # Extraction des polygones correspondants
> Ql <- aus1[pos, ]
```

Superposons nos deux objets en identifiant notre nouvel objet par une couleur distincte.

```
> # Representation cartographique
> par(mar = c(0, 0, 0, 0))
> plot(aus1)
> plot(Ql, col = "gray", add = T)
```

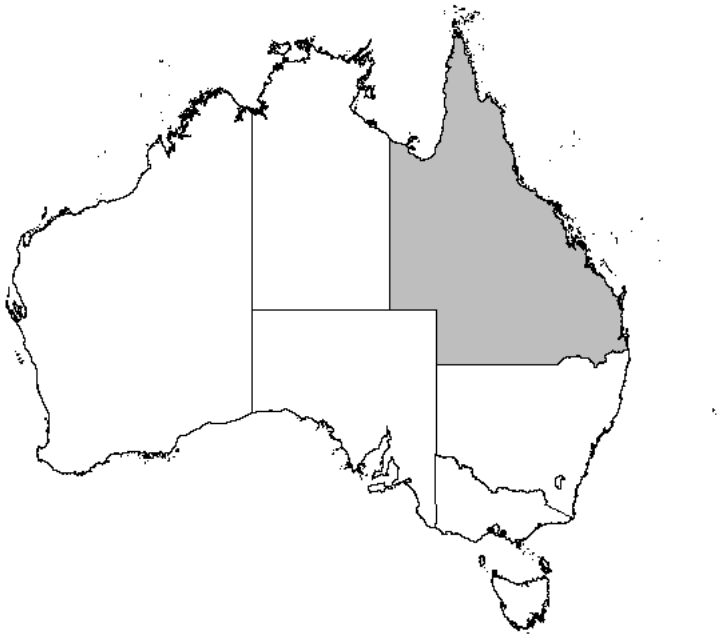

FIGURE 1.6 – Représentation cartographique du Queensland

Pour exporter un objet vectoriel, nous pouvons utiliser la fonction `writeOGR()` qui accepte plusieurs arguments. Voici ceux qui sont obligatoires :

- ***obj*** correspond au nom de l'objet vectoriel sous R ;
- ***dsn*** correspond, dans le cas de données type « ESRI Shapefile » au répertoire dans lequel seront écrits nos fichiers ;
- ***layer*** correspond au nom d'exportation du fichier (sans extension) ;

– ***driver*** correspond au format d'exportation de l'objet vectoriel (on peut utiliser la commande `ogrDrivers()` pour connaitre les formats disponibles).

```
> # Exportation d'un objet vectoriel
> writeOGR(obj = Ql, dsn = dest, layer = "Queensland", driver = "ESRI
Shapefile")
```

Affichons le contenu du répertoire de destination et vérifions que ce shapefile a été correctement exporté.

```
> # Contenu du répertoire d'exportation
> dir(dest)

##  [1] "AUS_adm.zip"    "AUS_adm0.dbf"   "AUS_adm0.prj"
##  [4] "AUS_adm0.sbn"   "AUS_adm0.sbx"   "AUS_adm0.shp"
##  [7] "AUS_adm0.shx"   "AUS_adm1.dbf"   "AUS_adm1.prj"
## [10] "AUS_adm1.sbn"   "AUS_adm1.sbx"   "AUS_adm1.shp"
## [13] "AUS_adm1.shx"   "AUS_adm2.dbf"   "AUS_adm2.prj"
## [16] "AUS_adm2.sbn"   "AUS_adm2.sbx"   "AUS_adm2.shp"
## [19] "AUS_adm2.shx"   "AUS_readme.txt" "Queensland.dbf"
## [22] "Queensland.prj" "Queensland.shp" "Queensland.shx"
```

Notre objet a été exporté sous le format « ESRI shapefile » en quatre fichiers (« .shp », « .dbf », « .prj » et « .shx »). Ce fichier sera directement lisible sous n'importe quel logiciel de SIG.

1.3.2 Couches matricielles

Pour explorer l'importation/exportation d'objets matriciels, nous allons également télécharger des données depuis Internet. Nous allons utiliser un raster décrivant l'altitude de l'Australie, disponible gratuitement sur le site DIVA-GIS (`http://diva-gis.org`).

Procédons maintenant au téléchargement.

```
> # Lien du telechargement
> link <- "http://www.diva-gis.org/data/alt/AUS_alt.zip"
> # Destination du fichier sur le disque
> dest <- paste(root, "/Australia/AUS_alt.zip", sep = "")
> # Telechargement
> download.file(url = link, destfile = dest, method = "internal")
```

Décompressons les fichiers contenus dans cette archive ZIP.

```
> # Decompression de l'archive telechargee
> unzip(zipfile = dest, exdir = paste(root, "/Australia", sep = ""))
```

Importation d'une couche matricielle

Nous allons procéder de la même manière que précédemment, sauf que nous allons recourir à la bibliothèque GDAL qui permet la lecture de nombreux formats de rasters. La fonction R correspondante est `readGDAL()`. Importons notre raster au format « .vrt ».

```
> # Importation d'un raster
> ras <- readGDAL(paste(root, "/Australia/AUS_alt.vrt", sep = ""),
+     silent = T)
```

Regardons comment il se structure sous R.

```
> # Classe de l'objet importe
> class(ras)

## [1] "SpatialGridDataFrame"
## attr(,"package")
## [1] "sp"

> # Nom de ses elements constitutifs
> slotNames(ras)

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

Nous n'en dirons pas plus sur les rasters, car la dernière section de ce chapitre leur sera consacrée : nous verrons notamment une alternative à l'utilisation des *packages* `sp` et `rgdal`.

Exportation d'une couche matricielle

Pour terminer, regardons comment exporter ce raster au format « GeoTIFF » avec le *package* `rgdal`. Nous allons utiliser la fonction `writeGDAL()` qui reçoit les arguments suivants :

- ***dataset*** correspond au nom du raster sous R ;
- ***fname*** correspond au nom d'exportation avec extension ;
- ***drivername*** correspond au format d'exportation de l'objet matriciel (on peut utiliser la commande `gdalDrivers()` pour connaitre les formats disponibles).

```
> # Fichier de destination
> dest <- paste(root, "/Australia/ras_alt.tif", sep = "")
> # Exportation du raster
> writeGDAL(dataset = ras, fname = dest, drivername = "GTiff")
```

Ce fichier est directement accessible par n'importe quel logiciel de SIG supportant le format « GeoTIFF ».

1.4 Le système de coordonnées

Le propre des objets géographiques est de pouvoir être représentés dans l'espace. Pour cela, ils doivent être définis dans un référentiel spatial, c.-à-d. un système de coordonnées. On distinguera la représentation en coordonnées géographiques (longitude et latitude) de la représentation en coordonnées projetées (représentation cartographique plane).

Nous avons déjà mentionné que la lecture de données géographiques avec le *package* `rgdal` permettait à la fois d'importer la géométrie de ces objets, mais aussi leur référentiel spatial, c.-à-d. le système de coordonnées dans lequel ils sont définis. Pour interpréter le référentiel géographique, ce *package* se base sur la bibliothèque *PROJ.4*. C'est une bibliothèque libre qui permet la conversion de nombreux systèmes de coordonnées, le tout en définissant un standard d'écriture.

Dans cette section, nous allons voir comment définir le système de coordonnées d'objets vectoriels et matriciels (structurés dans le format `sp`), ainsi que la manière de le projeter sous différentes représentations en coordonnées projetées.

1.4.1 Définition du système de coordonnées

Les données spatiales, formatées sous le *package* `sp`, contiennent le système de coordonnées dans le slot `@proj4string`. Regardons ce que contient cet élément pour le shapefile « aus1 ».

```
> # Systeme de projection de 'aus1'
> aus1@proj4string

## CRS arguments:
##  +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84
## +towgs84=0,0,0
```

Le système de coordonnées se présente sous forme d'un objet `CRS` (*Coordinate Reference System*) dont les arguments sont écrits au format *PROJ.4*. Que nous dit cet élément ? Que cet objet spatial est défini dans un référentiel de coordonnées géographiques, c.-à-d. non projeté (argument *+proj*) sous le datum et l'ellipsoïde WGS84.

Dans ce cas là, le système de coordonnées est déjà identifié. Comment définir le référentiel spatial d'un objet ? Illustrons cela en définissant le système de coordonnées des capitales australiennes que nous avons importées au tout début.

```
> # Rappel de la structure des donnees
> class(dat)

## [1] "SpatialPointsDataFrame"
## attr(,"package")
## [1] "sp"

> # Systeme de coordonnees
> dat@proj4string

## CRS arguments: NA
```

```
> # Definition du systeme (WGS84)
> prj <- CRS("+proj=longlat +datum=WGS84 +ellps=WGS84")
> # Attribution de ce systeme aux donnees
> dat@proj4string <- prj
> # Verification
> dat@proj4string

## CRS arguments:
##  +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

Maintenant, la question qui se pose tout naturellement, c'est comment écrire un système de coordonnées dans le standard *PROJ.4* ? Dans le monde de la géographie, il existe plusieurs standards d'écriture d'un système de coordonnées. Voici les trois principaux :

- le *PROJ.4*, dont nous avons déjà parlé ;
- le *WKT* (*Well-Known Text*) est un format standardisé utilisé par de nombreux logiciels servant à définir le système de coordonnées d'objets vectoriels (fichier « .prj » d'un ESRI Shapefile) ;
- le *EPSG* (*European Petroleum Survey Group*) est un système de codage qui attribue un code à chaque système de coordonnées.

La fonction `make_EPSG()` du *package* `rgdal` permet de lister un peu plus de 3700 systèmes de coordonnées sous le format *PROJ.4* et le code EPSG correspondant.

```
> # Liste des systemes de coordonnees sous PROJ.4
> prjs <- make_EPSG()
> # Affichons les lignes 2 et 5
> prjs[c(2, 5), ]

##   code       note                                        prj4
## 2 3821    # TWD67 +proj=longlat +ellps=aust_SA +no_defs
## 5 3906 # MGI 1901  +proj=longlat +ellps=bessel +no_defs
```

Nous allons introduire une fonction qui va nous être utile pour parcourir cette liste : la fonction `grep()`. Celle-ci permet de chercher une expression dans un vecteur. Par exemple, imaginons que nous cherchons à écrire un système de coordonnées pour lequel nous connaissons le code EPSG. Nous pouvons utiliser cette fonction pour chercher la ligne de la table « prjs » à laquelle se rencontre ce code.

Cherchons l'écriture du système de coordonnées géographiques sous l'ellipsoïde WGS84 (celui définissant le référentiel spatial de nos précédents objets) à partir de son code EPSG : 4326.

```
> # Recherche dans la colonne 'code' (EPSG)
> (pos <- grep(4326, prjs[, "code"]))

## [1] 249

> # Extraction de l'ecriture PROJ.4
> CRS(prjs[pos, "prj4"])

## CRS arguments:
##  +proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs
## +towgs84=0,0,0
```

Pour connaître le code EPSG d'un système de coordonnées particulier, le site suivant `http://prj2epsg.org` permet de convertir l'écriture *WKT* en code EPSG. Notons également que le site `http://spatialreference.org` donne la correspondance entre plusieurs formats d'écriture de systèmes de coordonnées.

Finalement, au lieu de recopier l'écriture en *PROJ.4* pour définir le système de coordonnées d'un objet sous R, on pourra directement utiliser le code EPSG. Voici comment procéder :

```
> # Suppression du systeme de coord. de dat
> dat@proj4string <- CRS(as.character(NA))
> # Definition du systeme de coordonnees de dat
> (dat@proj4string <- CRS("+init=epsg:4326"))

## CRS arguments:
##  +init=epsg:4326 +proj=longlat +ellps=WGS84 +datum=WGS84
## +no_defs +towgs84=0,0,0
```

1.4.2 Projection du système de coordonnées

Nous allons voir maintenant comment projeter un système de coordonnées géographiques en une représentation cartographique plane. Nous allons projeter le système de coordonnées du shapefile de l'Australie et des capitales australiennes (système de coordonnées géographiques en WGS 1984) dans le système de projection « Lambert Azimuthal Equal Area ». Ce système de projection correspond au code EPSG 3035. La fonction R utilisée sera `spTransform()`.

```
> # Projection de l'objet 'aus1'
> aus1p <- spTransform(x = aus1, CRSobj = CRS("+init=epsg:3035"))
> # Verification
> aus1p@proj4string

## CRS arguments:
##  +init=epsg:3035 +proj=laea +lat_0=52 +lon_0=10 +x_0=4321000
## +y_0=3210000 +ellps=GRS80 +units=m +no_defs

> # Projection de l'objet 'dat'
> datp <- spTransform(x = dat, CRSobj = CRS("+init=epsg:3035"))
```

Pour terminer cette section, représentons l'Australie dans le système de coordonnées projetées.

```
> # Representation cartographique
> par(mar = c(0, 0, 0, 0))
> plot(aus1p)
> plot(datp, pch = 19, add = T)
```

FIGURE 1.7 – Projection Lambert Azimuthal Equal Area

1.5 Introduction au package raster

Précédemment, nous avons vu que le *package* `rgdal` permettait d'importer et d'exporter des données matricielles sous différents formats. Une fois importées, ces données sont stockées dans R sous le format `SpatialGridDataFrame` (*package* `sp`). Nous allons maintenant introduire un autre *package* qui offre de nombreux outils pour manipuler les couches matricielles : le *package* `raster`. Voici les principales possibilités offertes par ce *package* :

- Création, importation et exportation de couches matricielles ;
- Conversion en objets sp ;
- Modification de l'étendue et de la résolution spatiale ;
- Interpolation et prédiction ;
- Calcul de différentes métriques ;
- Représentation cartographique.

Ce *package* présente la particularité de ne pas stocker dans la mémoire de R les valeurs d'un raster lors de son importation : elles ne seront lues que lorsque cela sera nécessaire (analyse, résumé statistique, cartographie). Ceci présente l'avantage de pouvoir manipuler des objets matriciels de très grandes dimensions sans toutefois saturer la mémoire du logiciel.

Approfondissons certaines fonctionnalités de ce *package*. Tout d'abord, chargeons-le dans la session de travail.

```
> # Chargement du package
> library(raster)
```

1.5.1 Importation d'un raster

Nous allons importer les données matricielles contenues dans le raster « AUS_alt » grâce la fonction `raster()`. Remarque : cette fonction nous servira également à créer un objet matriciel sous R (voir ci-après).

```
> # Importation d'un raster
> ras <- raster(paste(root, "/Australia/AUS_alt.grd", sep = ""))
> # Classe de l'objet cree
> class(ras)

## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"

> # Nom des elements qu'il contient
> slotNames(ras)

##  [1] "file"       "data"       "legend"     "history"    "title"
##  [6] "extent"     "rotated"    "rotation"   "ncols"      "nrows"
## [11] "crs"        "layernames" "z"          "zname"      "zvalue"
## [16] "unit"
```

Une fois importé, l'objet matriciel est stocké sous forme d'un `RasterLayer`, c.-à-d. un raster à une couche de données. Le *package* `raster` permet de créer des rasters à plusieurs couches d'information (collection de rasters de mêmes dimensions et résolution) : les objets créés seront stockés sous forme de `RasterStack` ou `RasterBrick`.

Cet objet contient plusieurs informations réparties en différents slots. Pour y accéder, on aura recours à l'opérateur `@` ou aux fonctions associées à ces slots. Regardons comment accéder à certaines d'entre elles.

```
> # Dimensions du raster
> ncol(ras)

## [1] 5568

> nrow(ras)

## [1] 5496

> # Nombre de cellules
> ncell(ras)

## [1] 30601728
```

```
> # Resolution du raster
> xres(ras)

## [1] 0.008333

> yres(ras)

## [1] 0.008333

> res(ras)

## [1] 0.008333 0.008333

> # Etendue spatiale
> extent(ras)

## class       : Extent
## xmin        : 112.8
## xmax        : 159.2
## ymin        : -54.9
## ymax        : -9.1
```

Maintenant, nous allons voir comment accéder aux valeurs contenues dans cet objet matriciel. Pour ce faire, nous devons établir une connexion avec le fichier source et utiliser la commande `getValues()`.

```
> # Acces aux donnees du raster
> val <- getValues(ras)
> # Quelques statistiques
> mean(na.omit(val))

## [1] 278

> min(na.omit(val))

## [1] -43

> max(na.omit(val))

## [1] 3492

> sd(na.omit(val))

## [1] 198.4
```

Pour terminer, représentons une carte de l'altitude de l'Australie.

```
> # Representation cartographique
> par(mar = c(0, 0, 0, 0))
> plot(ras)
```

FIGURE 1.8 – Raster de l'altitude de l'Australie

1.5.2 Système de coordonnées

Le *package* `raster` se base aussi sur la librairie libre *PROJ.4* pour manipuler le système de coordonnées des objets matriciels. La seule différence par rapport aux objets sp réside simplement dans les fonctions utilisées. Regardons comment définir le système de coordonnées d'un raster au format `RasterLayer`.

```
> # Acces au systeme de coordonnees de 'ras'
> projection(ras)

## [1] "+proj=longlat +ellps=WGS84"

> # Reinitialisation du systeme
> projection(ras) <- CRS(as.character(NA))
> # Definition du systeme
> prj <- CRS("+proj=longlat +datum=WGS84 +ellps=WGS84")
> # Attribution du systeme au raster
> projection(ras) <- prj
> # Verification
> projection(ras)

## [1] "+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
```

Pour projeter le système de coordonnées d'un raster, nous utiliserons la fonction `projectRaster()`. Projetons ce raster dans le système « Lambert Azimuthal Equal Area ».

```
> # Projection du raster
> rasp <- projectRaster(x = ras, CRSobj = CRS("+init=epsg:3035"))
```

1.5.3 Création d'un raster

La fonction `raster()` avec laquelle nous avons importé les données matricielles décrivant l'altitude de l'Australie va également nous servir à créer des rasters sous R dans le format `RasterLayer`. Tout comme dans la section 1.2.4, nous allons voir deux approches pour créer des rasters : une construction de grille à partir d'information minimale (*from scratch*) ou à partir de l'étendue spatiale d'un autre raster.

Génération automatique d'un raster

Tout comme la construction d'un objet `SpatialGrid`, nous allons créer un raster en spécifiant les dimensions de la grille et les coordonnées des cellules extrêmes. Regardons cela en attribuant le système de coordonnées géographiques du raster « ras ».

```
> # Creation d'un raster
> ras0 <- raster(nrows = 180, ncols = 360, xmn = -180, xmx = 180,
+     ymn = -90, ymx = 90, crs = projection(ras))
```

Affichons les caractéristiques principales de notre nouveau raster.

```
> # Caracteristiques du raster
> projection(ras0)

## [1] "+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"

> ncell(ras0)

## [1] 64800

> extent(ras0)

## class       : Extent
## xmin        : -180
## xmax        : 180
## ymin        : -90
## ymax        : 90

> res(ras0)

## [1] 1 1
```

On remarque que la résolution spatiale de notre nouveau raster est déterminée automatiquement à partir des dimensions de la grille et des coordonnées extrêmes. Rajoutons

maintenant des valeurs à notre objet matriciel : nous allons échantillonner aléatoirement 64800 valeurs (nombre de cellules de la grille) parmi des nombres allant de 1 à 100.

```
> # Tirage de valeurs aleatoires
> val <- sample(x = 1:100, size = ncell(ras0), replace = T)
> # Attribution au raster
> values(ras0) <- val
```

Visualisons notre objet auquel nous allons superposer le shapefile de l'Australie.

```
> # Representation cartographique
> par(mar = c(0, 0, 0, 0))
> plot(ras0)
> plot(aus1, add = T)
```

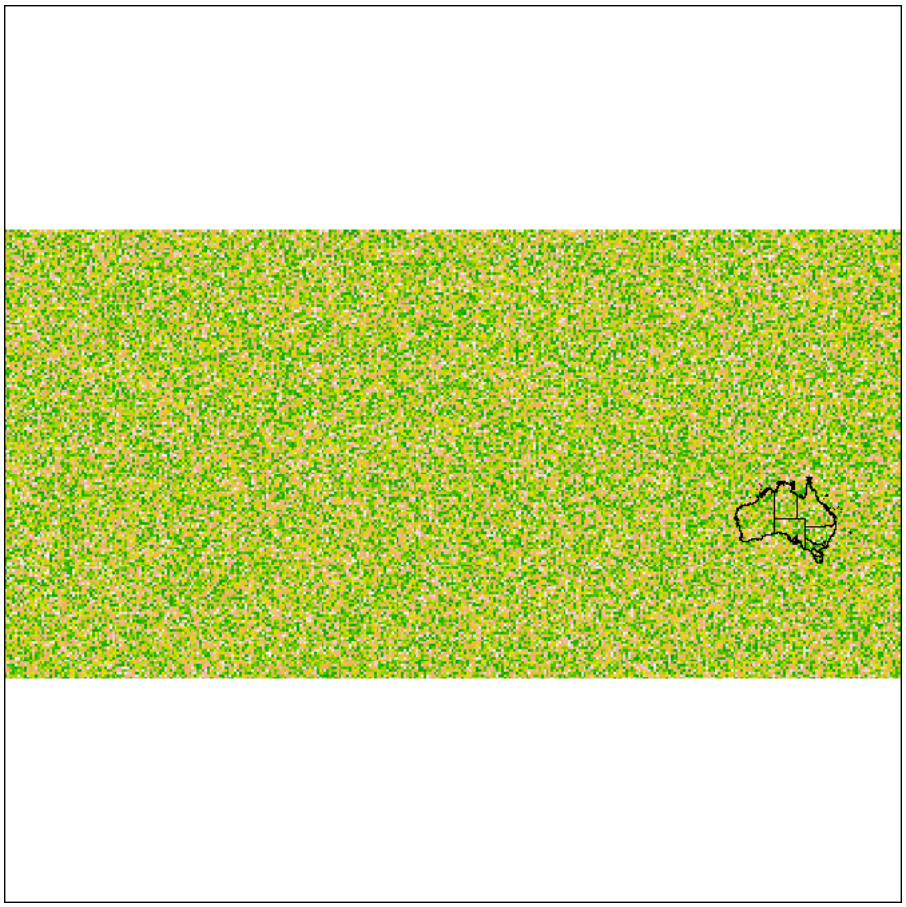

FIGURE 1.9 – Raster avec valeurs aléatoires

Création à partir d'un autre raster

Créons maintenant le même raster en spécifiant simplement son étendue spatiale à partir de celle d'un autre. L'écriture de la fonction `raster()` est beaucoup plus simple.

```
> # Etendue du raster 'ras0'
> ext <- extent(ras0)
> # Creation d'u nouveau raster
> ras1 <- raster(ext)
> # Definition des valeurs
> values(ras1) <- sample(x = 1:100, size = ncell(ras1), replace = T)
> # Definition du systeme de coordonnees
> projection(ras1) <- projection(ras0)
```

1.5.4 Exportation d'un raster

Le *package* `raster` s'appuie sur son homologue, le *package* `rgdal` pour lire et écrire des données matricielles. Ainsi, il permet d'exporter des objets matriciels sous différents formats. La fonction `writeFormats()` permet de connaitre ces différents formats.

```
> # Liste des formats de raster disponibles en ecriture
> head(writeFormats())

##      name     long_name
## [1,] "raster" "R-raster"
## [2,] "SAGA"   "SAGA GIS"
## [3,] "IDRISI" "IDRISI"
## [4,] "BIL"    "Band by Line"
## [5,] "BSQ"    "Band Sequential"
## [6,] "BIP"    "Band by Pixel"
```

Pour exporter un raster, nous utiliserons la fonction `writeRaster()`. Exportons le raster « ras0 » au format GeoTIFF.

```
> # Nom du fichier exporte
> dest <- paste(root, "/Australia/raster_world.tif", sep = "")
> # Exportation d'un raster au format GeoTIFF
> writeRaster(x = ras0, filename = dest, format = "GTiff", overwrite = T)
```

Chapitre 2

Modification des géométries

2.1 Manipulation de la table d'attributs

Dans le premier chapitre, nous avons vu que les données géographiques étaient stockées sous forme d'objets sp, lesquels répartissent les différentes informations (coordonnées géographiques, système de coordonnées, étendue spatiale, table d'attributs, etc.) dans des slots distincts. La table d'attributs d'objets vectoriels est contenue dans le slot `@data` et se présente sous la forme d'un `data.frame`. Ainsi, il sera très facile de manipuler cette table d'attributs à des fins diverses : accès à certaines valeurs de la table, modification de valeurs, rajout et suppression de champs, ou encore réorganisation de la table.

Suppression de champs

Regardons tout d'abord comment supprimer certaines colonnes de la table d'attributs de l'objet vectoriel « aus1 ».

```
> # Suppression de certains champs
> (aus1@data <- aus1@data[, -c(1, 2, 6, 7, 9:16)])

##       NAME_0 ID_1                       NAME_1 HASC_1
## 0  Australia  151  Ashmore and Cartier Islands  AU.AS
## 1  Australia  152 Australian Capital Territory  AU.CT
## 2  Australia  153            Coral Sea Islands  AU.CR
## 3  Australia  154              New South Wales  AU.NS
## 4  Australia  155           Northern Territory  AU.NT
## 5  Australia  156                   Queensland  AU.QL
## 6  Australia  157              South Australia  AU.SA
## 7  Australia  158                     Tasmania  AU.TS
## 8  Australia  159                     Unknown1  AU.TS
## 9  Australia  160                     Victoria  AU.VI
## 10 Australia  161            Western Australia  AU.WA
```

L'opérateur `-` permet de supprimer certaines colonnes d'un `data.frame`. Nous aurions également pu sélectionner les colonnes que nous souhaitions conserver. Nous allons maintenant renommer le nom des colonnes conservées.

```
> # Renommer les colonnes
> colnames(aus1@data) <- c("Pays", "ID", "Etat", "Code")
> head(aus1@data)

##        Pays  ID                          Etat  Code
## 0 Australia 151  Ashmore and Cartier Islands AU.AS
## 1 Australia 152 Australian Capital Territory AU.CT
## 2 Australia 153            Coral Sea Islands AU.CR
## 3 Australia 154              New South Wales AU.NS
## 4 Australia 155           Northern Territory AU.NT
## 5 Australia 156                   Queensland AU.QL
```

Modification de valeurs

Nous allons maintenant voir comment modifier certaines valeurs de la table. Modifions le contenu de la dernière colonne afin de ne conserver que l'abréviation des états (sans l'abréviation du pays).

```
> # Modification de la dernière colonne
> aus1@data[, "Code"] <- substr(aus1@data[, "Code"], 4, 5)
> aus1@data

##         Pays  ID                          Etat Code
## 0  Australia 151  Ashmore and Cartier Islands   AS
## 1  Australia 152 Australian Capital Territory   CT
## 2  Australia 153            Coral Sea Islands   CR
## 3  Australia 154              New South Wales   NS
## 4  Australia 155           Northern Territory   NT
## 5  Australia 156                   Queensland   QL
## 6  Australia 157              South Australia   SA
## 7  Australia 158                     Tasmania   TS
## 8  Australia 159                     Unknown1   TS
## 9  Australia 160                     Victoria   VI
## 10 Australia 161            Western Australia   WA
```

Cet objet vectoriel comporte une série de polygones dont les informations contenues dans la table d'attributs sont inconnues : il s'agit du polygone multiple 8. Nous allons donc remplacer les valeurs des deux dernières colonnes par la valeur `NA`. Mais avant, nous devons convertir le type de la colonne « Etat » en une chaîne de caractères (actuellement, cette colonne est stockée sous forme d'un facteur).

```
> # Conversion du type
> aus1@data[, "Etat"] <- as.character(aus1@data[, "Etat"])
> # Modification des valeurs
> aus1@data[9, "Etat"] <- NA
> aus1@data[9, "Code"] <- NA
> # Verification
> aus1@data
```

```
##         Pays  ID                          Etat Code
## 0  Australia 151  Ashmore and Cartier Islands   AS
## 1  Australia 152 Australian Capital Territory   CT
## 2  Australia 153            Coral Sea Islands   CR
## 3  Australia 154              New South Wales   NS
## 4  Australia 155           Northern Territory   NT
## 5  Australia 156                   Queensland   QL
## 6  Australia 157              South Australia   SA
## 7  Australia 158                     Tasmania   TS
## 8  Australia 159                         <NA> <NA>
## 9  Australia 160                     Victoria   VI
## 10 Australia 161            Western Australia   WA
```

Ajout de champs

Regardons comment rajouter une colonne dans cette table d'attributs. Les `data.frame` sont des objets R qui présentent l'avantage de pouvoir créer, ajouter et calculer une colonne en une seule étape. Pour ce faire, nous aurons recours à l'opérateur `$`. Rajoutons donc une colonne qui donne la population de chaque état/territoire australien.

```
> # Rajout d'une colonne
> aus1@data$Population <- c(0, 370729, 4, 7247669, 232365, 4513009,
+     1645040, 511718, NA, 5574455, 2387232)
> # Verification
> head(aus1@data)

##        Pays  ID                          Etat Code Population
## 0 Australia 151  Ashmore and Cartier Islands   AS          0
## 1 Australia 152 Australian Capital Territory   CT     370729
## 2 Australia 153            Coral Sea Islands   CR          4
## 3 Australia 154              New South Wales   NS    7247669
## 4 Australia 155           Northern Territory   NT     232365
## 5 Australia 156                   Queensland   QL    4513009
```

De nombreux calculs peuvent être effectués sur cette table d'attributs. Par exemple, répondons à cette question : quelle est la population totale de l'Australie ?

```
> # Population australienne
> sum(na.omit(aus1@data[, "Population"]))

## [1] 22482221
```

Tri de la table d'attributs

Pour terminer cette section, nous allons voir une commande très pratique qui va nous permettre de réorganiser cette table d'attributs. Nous allons réorganiser cette table d'attributs de manière à trier les lignes par population décroissante. Utilisons pour cela la fonction `order()` qui retourne les positions des lignes dans l'ordre souhaité.

Mais tout d'abord, affichons l'identifiant de tous les polygones.

```
> # Identifiants des polygones
> sapply(aus1@polygons, function(x) slot(x, "ID"))

##  [1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

> # Nom des lignes de la table
> rownames(aus1@data)

##  [1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

L'ordre des lignes et des polygones est identique. Nous souhaitons maintenant modifier l'ordre de ces lignes ainsi que celui des polygones, de manière à ce que les attributs de la première ligne se réfèrent bien au premier polygone.

```
> # Tri conditionnel
> (pos <- order(aus1@data$Population, decreasing = T))

##  [1]  4 10  6 11  7  8  2  5  3  1  9

> # Reorganisation de l'objet
> aus1 <- aus1[pos, ]
> # Identifiants des polygones
> sapply(aus1@polygons, function(x) slot(x, "ID"))

##  [1] "3"  "9"  "5"  "10" "6"  "7"  "1"  "4"  "2"  "0"  "8"

> # Table d'attributs
> aus1@data

##         Pays  ID                          Etat Code Population
## 3  Australia 154               New South Wales   NS    7247669
## 9  Australia 160                      Victoria   VI    5574455
## 5  Australia 156                    Queensland   QL    4513009
## 10 Australia 161             Western Australia   WA    2387232
## 6  Australia 157               South Australia   SA    1645040
## 7  Australia 158                      Tasmania   TS     511718
## 1  Australia 152 Australian Capital Territory   CT     370729
## 4  Australia 155            Northern Territory   NT     232365
## 2  Australia 153             Coral Sea Islands   CR          4
## 0  Australia 151  Ashmore and Cartier Islands   AS          0
## 8  Australia 159                          <NA> <NA>         NA
```

Ainsi, la réorganisation de notre objet vectoriel a bien fonctionné, puisque l'ordre des polygones a été modifié tant dans la table d'attributs que dans la géométrie (slot `@polygons`). C'est là la grande force des objets sp.

2.2 Opérations sur une couche vectorielle

Nous venons de voir que la table d'attributs pouvait être manipulée très facilement. Nous allons maintenant voir des opérations plus poussées nécessitant un référencement à certains attributs de cette table.

2.2.1 Jointure par identifiant

Précédemment, nous avons rajouté une colonne « Population » à la table d'attributs. Nous ne l'avons pas mentionné explicitement, mais l'ordre des lignes devait correspondre entre la table et cette nouvelle colonne. Nous allons maintenant voir une manière extrêmement plus fiable de rajouter de l'information à une table d'attributs. Nous allons joindre une nouvelle table à cette table d'attributs au moyen d'une « clé ». C'est ce qu'on appelle dans le langage des bases de données relationnelles une jointure par identifiant (ou par clé primaire).

Tout d'abord, importons cette table que l'on souhaite joindre à notre objet vectoriel. Ces données, non géoréférencées, fournissent la capitale de chaque état/territoire australien.

```
> # Importations des donnees
> (mat <- read.delim(paste(root, "/statesAUS.txt", sep = "")))

##    Id State   Capitale
## 1   0    AS   Canberra
## 2   1    CT   Canberra
## 3   2    CR Ile Willis
## 4   3    NS     Sydney
## 5   4    NT     Darwin
## 6   5    QL   Brisbane
## 7   6    SA   Adelaide
## 8   7    TS     Hobart
## 9   8  <NA>       <NA>
## 10  9    VI  Melbourne
## 11 10    WA      Perth
```

Pour procéder au joint par identifiant, deux conditions doivent être remplies :

- les lignes des deux tables doivent comporter les mêmes identifiants (la clé sera le nom des lignes) ;
- l'ordre des lignes doit être le même dans les deux tables.

Commençons par trier de nouveau notre objet spatial « aus1 » par l'identifiant des polygones.

```
> # Tri conditionnel
> (pos <- order(as.numeric(rownames(aus1@data)), decreasing = F))

##  [1] 10  7  9  1  8  3  5  6 11  2  4

> # Reorganisation de l'objet
> aus1 <- aus1[pos, ]
```

```
> # Verification
> rownames(aus1@data)

##  [1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Maintenant, définissons le nom des lignes de la table « mat ».

```
> rownames(mat) <- mat[, "Id"]
> mat

##    Id State   Capitale
## 0   0    AS   Canberra
## 1   1    CT   Canberra
## 2   2    CR Ile Willis
## 3   3    NS     Sydney
## 4   4    NT     Darwin
## 5   5    QL   Brisbane
## 6   6    SA   Adelaide
## 7   7    TS     Hobart
## 8   8  <NA>       <NA>
## 9   9    VI  Melbourne
## 10 10    WA      Perth
```

Supprimons quelques champs dans les deux tables.

```
> # Suppression de champs
> aus1@data <- aus1@data[, c(1, 3:5)]
> head(aus1@data)

##        Pays                          Etat Code Population
## 0 Australia  Ashmore and Cartier Islands   AS          0
## 1 Australia Australian Capital Territory   CT     370729
## 2 Australia            Coral Sea Islands   CR          4
## 3 Australia              New South Wales   NS    7247669
## 4 Australia           Northern Territory   NT     232365
## 5 Australia                   Queensland   QL    4513009

> mat <- mat[, -1]
> head(mat)

##   State   Capitale
## 0    AS   Canberra
## 1    CT   Canberra
## 2    CR Ile Willis
## 3    NS     Sydney
## 4    NT     Darwin
## 5    QL   Brisbane
```

Assurons-nous que l'ordre des lignes dans les deux tables soit identique.

```
> rownames(aus1@data) == rownames(mat)

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

Procédons maintenant à la jointure par identifiant (la clé sera le nom des lignes) grâce à la fonction `spCbind()` du *package* `maptools`.

```
> # Chargement du package
> library(maptools)
> # Jointure par identifiant
> aus1 <- spCbind(aus1, mat)
> # Verification
> head(aus1@data)

##        Pays                          Etat Code Population State
## 0 Australia  Ashmore and Cartier Islands   AS          0    AS
## 1 Australia Australian Capital Territory   CT     370729    CT
## 2 Australia            Coral Sea Islands   CR          4    CR
## 3 Australia              New South Wales   NS    7247669    NS
## 4 Australia           Northern Territory   NT     232365    NT
## 5 Australia                    Queensland   QL    4513009    QL
##     Capitale
## 0   Canberra
## 1   Canberra
## 2 Ile Willis
## 3     Sydney
## 4     Darwin
## 5   Brisbane
```

Les colonnes « Etat » et « State » sont identiques, preuve que la jointure a bien fonctionné. Supprimons la colonne « State » de manière à alléger la lecture de la table.

```
> aus1@data <- aus1@data[, -5]
> aus1@data

##         Pays                          Etat Code Population   Capitale
## 0  Australia  Ashmore and Cartier Islands   AS          0   Canberra
## 1  Australia Australian Capital Territory   CT     370729   Canberra
## 2  Australia            Coral Sea Islands   CR          4 Ile Willis
## 3  Australia              New South Wales   NS    7247669     Sydney
## 4  Australia           Northern Territory   NT     232365     Darwin
## 5  Australia                    Queensland   QL    4513009   Brisbane
## 6  Australia              South Australia   SA    1645040   Adelaide
## 7  Australia                     Tasmania   TS     511718     Hobart
## 8  Australia                         <NA> <NA>         NA       <NA>
## 9  Australia                     Victoria   VI    5574455  Melbourne
## 10 Australia            Western Australia   WA    2387232      Perth
```

2.2.2 Suppression de données

Notre table d'attributs comporte une ligne (c.-à-d. un polygone multiple) dont les attributs sont manquants (NA). Nous allons donc voir comment supprimer des entités de notre objet spatial. Supprimons tous les polygones dont les attributs sont manquants.

```
> # Identification de la ligne avec NA
> pos <- which(is.na(aus1@data[, "Code"]))
> # Suppression des polygones correspondants
> aus <- aus1[-pos, ]
> # Verification
> aus@data

##         Pays                          Etat Code Population   Capitale
## 0  Australia  Ashmore and Cartier Islands   AS          0   Canberra
## 1  Australia Australian Capital Territory   CT     370729   Canberra
## 2  Australia            Coral Sea Islands   CR          4 Ile Willis
## 3  Australia              New South Wales   NS    7247669     Sydney
## 4  Australia           Northern Territory   NT     232365     Darwin
## 5  Australia                   Queensland   QL    4513009   Brisbane
## 6  Australia              South Australia   SA    1645040   Adelaide
## 7  Australia                     Tasmania   TS     511718     Hobart
## 9  Australia                     Victoria   VI    5574455  Melbourne
## 10 Australia            Western Australia   WA    2387232      Perth
```

Non seulement ces éléments ont été supprimés de la table d'attributs, mais ils ont également été éliminés de la géométrie. Vérifions que l'identifiant **8** a disparu de la liste des identifiants des polygones.

```
> # Identifiants des polygones
> sapply(aus@polygons, function(x) slot(x, "ID"))

##  [1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "9"  "10"
```

Ici, il est important de retenir que tout changement apporté aux différents éléments (polygones) de l'objet vectoriel se répercute à la fois sur la table d'attributs et sur la géométrie.

Représentons une carte avec ces objets spatiaux. En rouge, nous allons afficher le polygone spatial avec tous les polygones initiaux, tandis qu'en noir nous représenterons l'objet spatial auquel on aura ôté les polygones sans attributs. Puisque les objets sont quasiment identiques, les éléments représentés en rouge sur la carte seront les éléments sans attributs que nous avons supprimés (localisés à l'extrême sud-est de l'Australie).

```
> # Representation cartographique
> par(mar = c(0, 0, 0, 0))
> plot(aus1, border = "red")
> plot(aus, add = T)
```

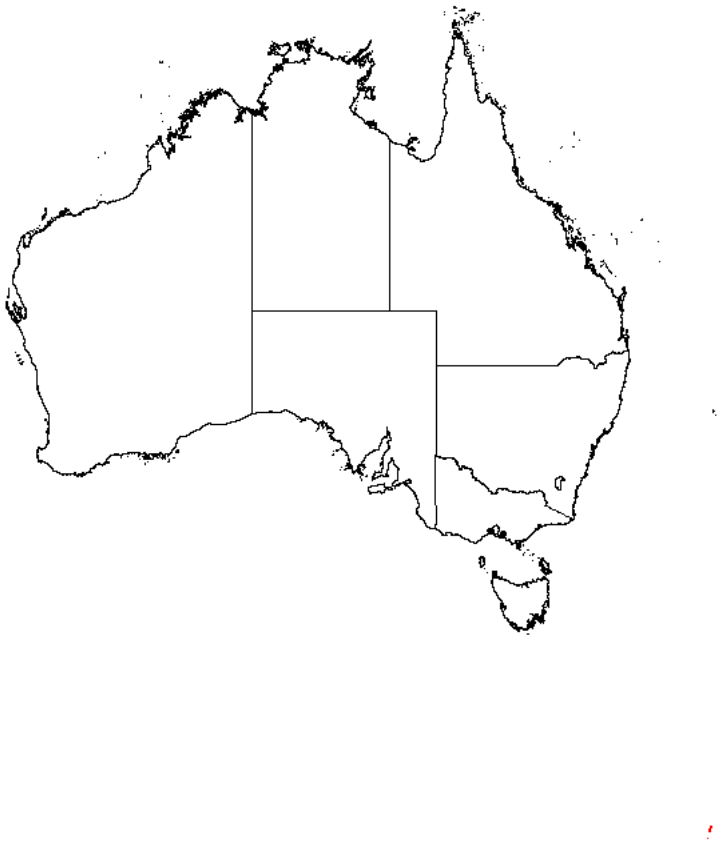

FIGURE 2.1 – Carte de l'Australie avec des données supprimées

2.2.3 Extraction par attributs

Dans la section 1.3.1, nous avons déjà extrait certains polygones d'un objet spatial. Regardons de nouveau comment procéder en construisant deux nouveaux polygones spatiaux : « aus_w » qui comprendra tous les états de l'Ouest australien et « aus_e » contenant les autres états.

Identifions dans la table d'attributs les lignes correspondant aux états suivants : Australie occidentale, Territoire du Nord et Australie méridionale.

```
> # Identification des etats
> (pos <- which(aus@data[, "Etat"] == "Western Australia" | aus@data[,
+     "Etat"] == "Northern Territory" | aus@data[, "Etat"] == "South
Australia"))

## [1]  5  7 10
```

Créons maintenant nos deux shapefiles.

```
> # Australie de l'Ouest
> aus_w <- aus[pos, ]
> # Australie de l'Est
> aus_e <- aus[-pos, ]
```

Représentons nos deux shapefiles par une couleur distincte.

```
> # Representation cartographique
> par(mar = c(0, 0, 0, 0))
> plot(aus)
> plot(aus_w, col = "black", border = "white", add = T)
> plot(aus_e, col = "gray", border = "white", add = T)
```

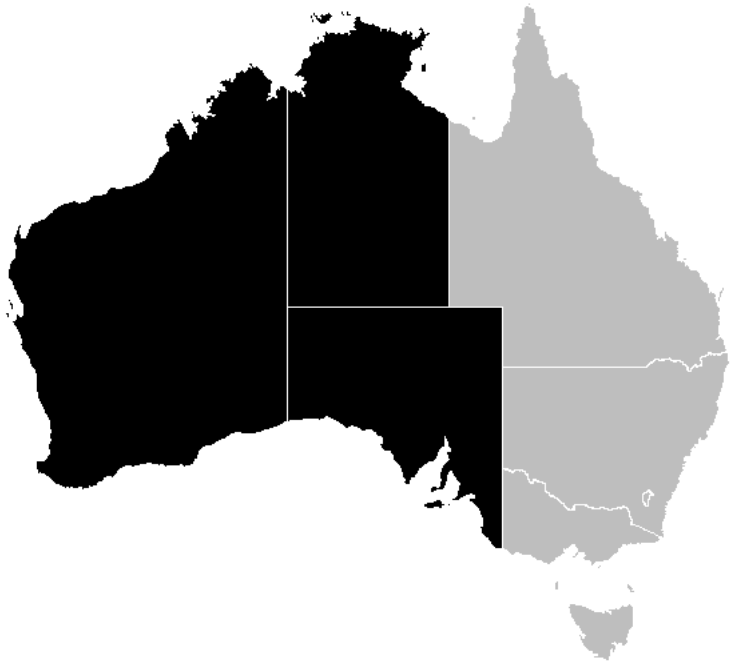

FIGURE 2.2 – Shapefiles issus d'une extraction par attributs

Affichons la table d'attributs de nos shapefiles.

```
> # Table d'attributs de aus_w
> aus_w@data

##         Pays               Etat Code Population Capitale
## 4  Australia Northern Territory   NT     232365   Darwin
## 6  Australia    South Australia   SA    1645040 Adelaide
## 10 Australia  Western Australia   WA    2387232    Perth

> # Table d'attributs de aus_e
> aus_e@data
```

```
##        Pays                          Etat Code Population   Capitale
## 0 Australia  Ashmore and Cartier Islands   AS          0   Canberra
## 1 Australia Australian Capital Territory   CT     370729   Canberra
## 2 Australia            Coral Sea Islands   CR          4 Ile Willis
## 3 Australia              New South Wales   NS    7247669     Sydney
## 5 Australia                   Queensland   QL    4513009   Brisbane
## 7 Australia                     Tasmania   TS     511718     Hobart
## 9 Australia                     Victoria   VI    5574455  Melbourne
```

2.2.4 Dissolution de polygones

Le *package* `rgeos` met à notre disposition de nombreuses fonctions permettant de modifier la géométrie d'objets spatiaux. Ces fonctions vont trouver leurs équivalents dans de nombreux logiciels de SIG. C'est notamment le cas de la fonction `gUnionCascaded()` qui permet de dissoudre plusieurs polygones entre eux (sous ArcMap, c'est la fonction *Dissolve*). Importons ce *package* dans notre session de travail.

```
> # Chargement du package
> library(rgeos)
```

Nous allons dissoudre tous les polygones du shapefile « aus_w », puis tous ceux de « aus_w ».

```
> # Dissolution de polygones
> aus_wd <- gUnionCascaded(aus_w)
> aus_ed <- gUnionCascaded(aus_e)
```

Affichons le nom des slots contenus dans ces nouveaux objets.

```
> # Noms des slots
> slotNames(aus_wd)

## [1] "polygons"    "plotOrder"   "bbox"        "proj4string"
```

La table d'attributs a disparu (absence du slot `@data`), ce qui est normal étant entendu que nous avons fusionné tous les états entre eux.

Représentons ces deux polygones dissouts.

```
> # Representation cartographique
> par(mar = c(0, 0, 0, 0))
> plot(aus)
> plot(aus_wd, col = "gray", add = T)
> plot(aus_ed, col = "black", add = T)
```

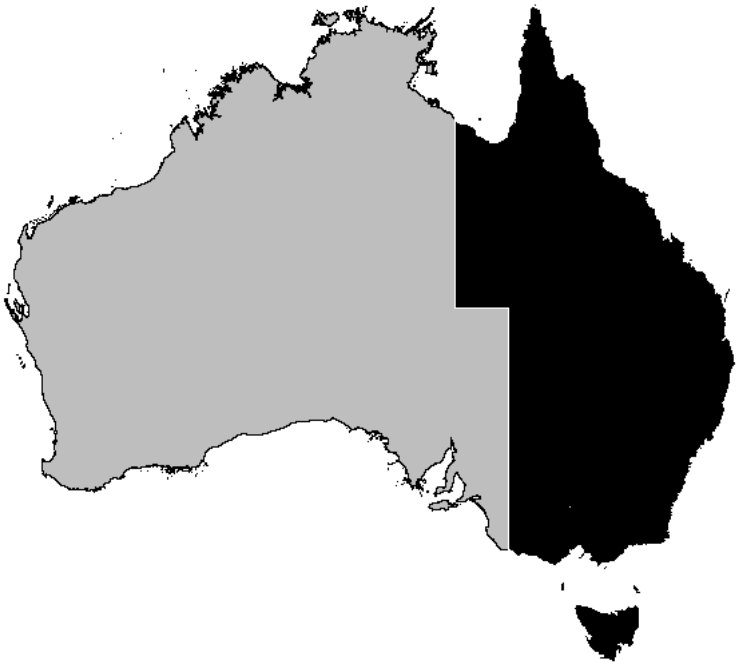

FIGURE 2.3 – Dissolution de polygones

2.3 Opérations à deux couches vectorielles

Poursuivons l'exploration d'objets vectoriels, et regardons comment manipuler plusieurs couches entre elles. La plupart des fonctions qui seront abordées dans cette section se trouvent dans le *package* `rgeos`. Nous allons voir plus particulièrement comment regrouper plusieurs shapefiles entre eux, comment procéder à une jointure spatiale (points-polygones et polygones-polygones) et comment intersecter des polygones. Pour terminer, nous écrirons une fonction qui permettra de découper une couche de polygones par une autre (équivalent du *Clip* sous ArcMap).

2.3.1 Combinaison de couches

Nous allons tout d'abord voir comment regrouper deux couches vectorielles en un seul objet spatial. Nous allons combiner les deux polygones spatiaux créés précédemment (« aus_wd » et « aus_ed ») entre eux. Avant cela, nous allons leur rajouter une table d'attributs.

```
> # Calcul de la population de l'Australie de l'Ouest
> Pop <- sum(aus_w@data[, "Population"])
> # Table d'attributs
> mat <- data.frame(Pays = "Australia", Etat = "West", Code = "WA",
+     Population = Pop)
```

```
> # Nom des lignes
> rownames(mat) <- slot(aus_wd@polygons[[1]], "ID")
> # Placement de la table dans le shapefile
> aus_wd <- SpatialPolygonsDataFrame(aus_wd, data = mat)
> # Verification
> class(aus_wd)

## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"

> aus_wd@data

##        Pays Etat Code Population
## 1 Australia West   WA    4264637
```

Procédons de même pour la seconde couche vectorielle.

```
> # Calcul de la population de l'Australie de l'Est
> Pop <- sum(aus_e@data[, "Population"])
> # Table d'attributs
> mat <- data.frame(Pays = "Australia", Etat = "East", Code = "EA",
+     Population = Pop)
> # Nom des lignes
> rownames(mat) <- slot(aus_ed@polygons[[1]], "ID")
> # Placement de la table dans le shapefile
> aus_ed <- SpatialPolygonsDataFrame(aus_ed, data = mat)
> # Verification
> class(aus_ed)

## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"

> aus_ed@data

##        Pays Etat Code Population
## 1 Australia East   EA   18217584
```

La combinaison de polygones spatiaux ne pourra se faire qu'à une seule condition : les identifiants des polygones (c.-à-d. le nom des lignes des tables d'attributs) doivent être uniques dans les deux couches vectorielles. Vérifions si c'est bien le cas.

```
> # Nom des identifiants des polygones dans 'aus_wd'
> sapply(aus_wd@polygons, function(x) slot(x, "ID"))

## [1] "1"

> # Nom des identifiants des polygones dans 'aus_ed'
> sapply(aus_ed@polygons, function(x) slot(x, "ID"))

## [1] "1"
```

Il s'avère que nos deux objets spatiaux sont formés d'un seul polygone multiple chacun, polygones qui portent le même identifiant dans les deux objets (ceci est une conséquence des dissolutions précédentes). Ainsi, nous devons renommer ces identifiants de manière à ce qu'ils soient uniques. Nous allons devoir utiliser une fonction spéciale qui permet de changer cet identifiant à la fois dans la table d'attributs et dans la géométrie : la fonction `spChFIDs()` du *package* `sp`.

Changeons l'identifiant des polygones de nos couches vectorielles.

```
> # Modification de l'identifiant du polygone de 'aus_wd'
> aus_wd <- spChFIDs(obj = aus_wd, x = "W")
> # Verification dans la geometrie
> sapply(aus_wd@polygons, function(x) slot(x, "ID"))

## [1] "W"

> # Verification dans la table d'attributs
> rownames(aus_wd@data)

## [1] "W"

> # Modification de l'identifiant du polygone de 'aus_ed'
> aus_ed <- spChFIDs(obj = aus_ed, x = "E")
> # Verification dans la geometrie
> sapply(aus_ed@polygons, function(x) slot(x, "ID"))

## [1] "E"

> # Verification dans la table d'attributs
> rownames(aus_ed@data)

## [1] "E"
```

Nous pouvons maintenant procéder à la combinaison des deux couches. Pour cela, nous allons utiliser la fonction `spRbind()` du *package* `maptools` qui va à la fois regrouper les géométries ainsi que les tables d'attributs.

```
> # Combinaison des couches
> aus2 <- spRbind(aus_wd, aus_ed)
> # Table d'attributs
> aus2@data

##        Pays Etat Code Population
## W Australia West   WA    4264637
## E Australia East   EA   18217584

> # Nombre de polygones multiples
> length(aus2@polygons)

## [1] 2
```

Représentons maintenant une carte de notre nouvelle couche vectorielle.

```
> # Representation cartographique
> par(mar = c(0, 0, 0, 0))
> plot(aus2, col = "gray")
```

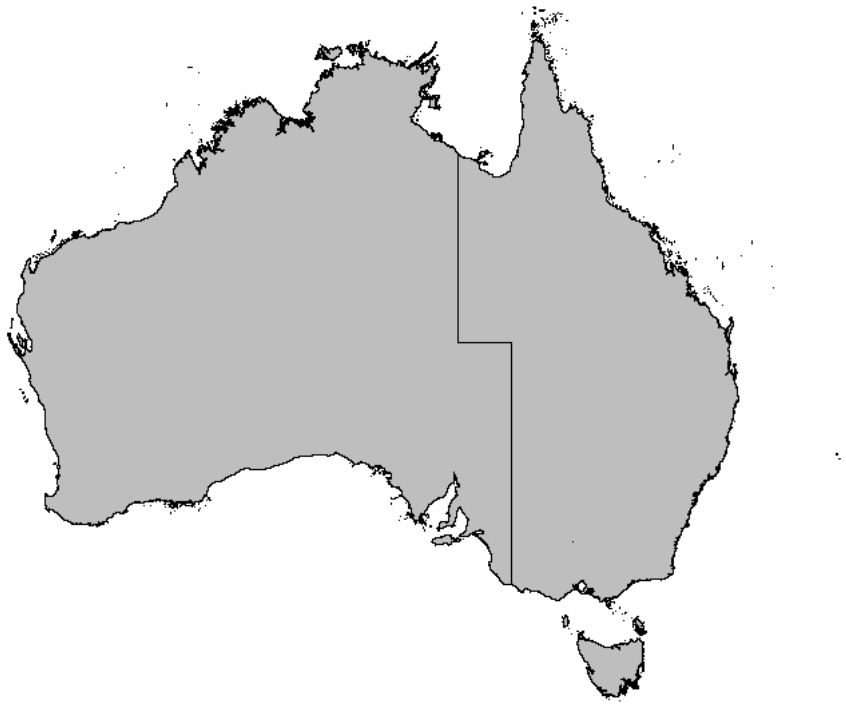

FIGURE 2.4 – Combinaison de polygones

2.3.2 Jointure spatiale

Nous allons nous intéresser maintenant aux jointures spatiales, c.-à-d. à la proximité spatiale entre éléments de couches différentes. En d'autres termes, il s'agit d'identifier les éléments d'une couche qui se superposent en partie (ou totalement) aux éléments d'une autre couche. Sous ArcMap, c'est la fonction *Spatial Join*. Nous allons voir deux cas de figure : un joint spatial entre des points et des polygones, et un joint spatial entre des polygones et des polygones.

Points-polygones

Commençons par créer une couche spatiale de points. Nous allons tirer aléatoirement 50 points à l'intérieur de l'étendue spatiale définie par le shapefile de l'Australie (« aus »). Pour cela, il nous faut convertir l'étendue spatiale de notre shapefile (actuellement sous forme d'une `matrix`) en un `SpatialPolygons` (Section 1.2.2).

```
> # Coordonnees de l'etendue spatiale de 'aus'
> x <- c(bbox(aus)[1, 1], bbox(aus)[1, 2], bbox(aus)[1, 2], bbox(aus)[1,
+     1])
> y <- c(bbox(aus)[2, 2], bbox(aus)[2, 2], bbox(aus)[2, 1], bbox(aus)[2,
+     1])
> xy <- data.frame(x, y)
> xy <- rbind(xy, xy[1, ])
> # Conversion en SpatialPolygons
> pol <- SpatialPolygons(list(Polygons(list(Polygon(xy)), "Poly")))
> # Definition du systeme de coordonnees
> proj4string(pol) <- proj4string(aus)
```

Représentons notre polygone auquel on va superposer le shapefile de l'Australie.

```
> par(mar = c(0, 0, 0, 0))
> plot(pol, col = "gray")
> plot(aus, col = "black", border = "white", add = T)
```

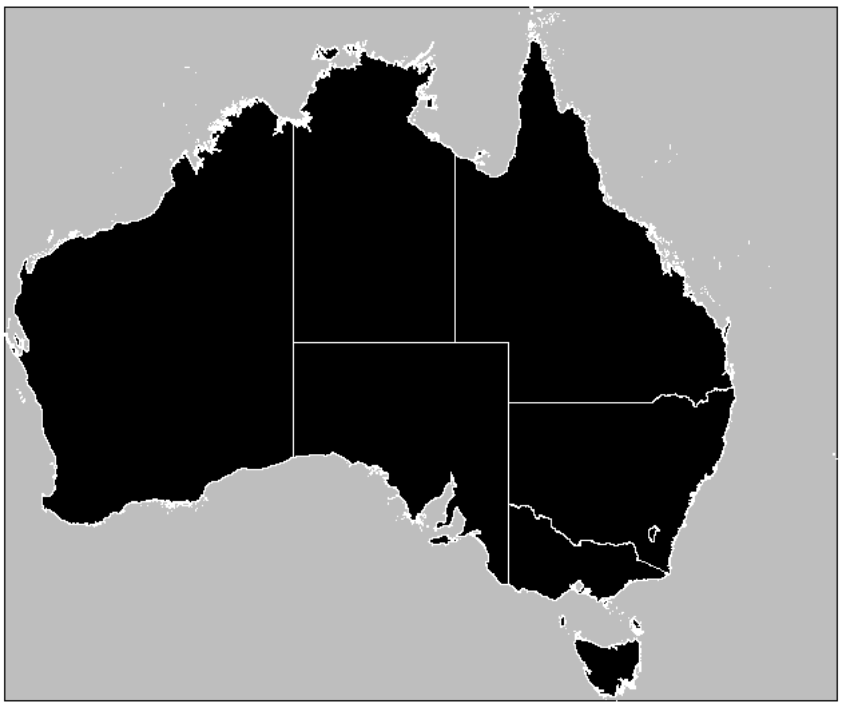

FIGURE 2.5 – Étendue spatiale de l'Australie

Maintenant, nous allons sélectionner aléatoirement 50 points à l'intérieur de cette étendue spatiale. Le *package* `sp` met à notre disposition la fonction `spsample()` pour réaliser cette opération.

```
> # Selection aleatoire de 50 points dans 'pol'
> pts <- spsample(x = pol, n = 50, type = "random")
```

Regardons de plus près la structure de l'objet créé.

```
> # Classe de l'objet cree
> class(pts)

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

Superposons cette série de points aléatoires sur la carte précédente.

```
> par(mar = c(0, 0, 0, 0))
> plot(pol, col = "gray")
> plot(aus, col = "black", border = "white", add = T)
> plot(pts, pch = 19, col = "red", add = T)
```

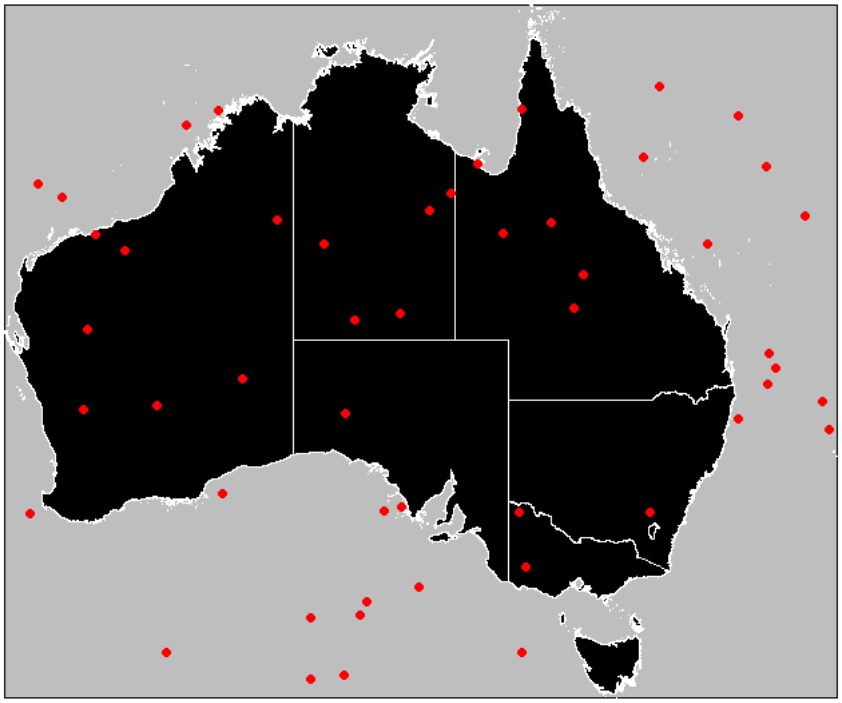

FIGURE 2.6 – Sélection de points aléatoires

Rajoutons une table d'attributs à ce `SpatialPoints` qui contiendra un identifiant unique pour chaque point.

```
> # Definition de la table d'attributs
> mat <- data.frame(ID = paste("Random", 1:50, sep = ""))
> # Rajout de la table d'attributs
> pts <- SpatialPointsDataFrame(pts, data = mat)
> # Verification
> pts[1:5, ]

##           coordinates      ID
## 1 (149.521, -22.7962) Random1
## 2 (122.866, -33.1808) Random2
## 3 (133.848, -40.6259) Random3
## 4  (154.03, -40.4912) Random4
## 5 (135.268, -38.1105) Random5
```

Maintenant, nous allons pouvoir procéder à la jointure spatiale entre ces points et la couche vectorielle de l'Australie. Pour ce faire, nous allons utiliser la fonction `over()` du *package* `sp`.

```
> # Jointure spatiale
> join <- over(pts, aus)
> # Classe de l'objet cree
> class(join)

## [1] "data.frame"

> # Dimensions de l'objet cree
> dim(join)

## [1] 50  5

> # Affichage du resultat
> head(join)

##        Pays              Etat Code Population Capitale
## 1 Australia        Queensland   QL    4513009 Brisbane
## 2 Australia Western Australia   WA    2387232    Perth
## 3      <NA>              <NA> <NA>         NA     <NA>
## 4      <NA>              <NA> <NA>         NA     <NA>
## 5      <NA>              <NA> <NA>         NA     <NA>
## 6      <NA>              <NA> <NA>         NA     <NA>
```

Le résultat de la jointure spatiale entre des points et des polygones se présente sous forme d'une table dont le nombre de lignes correspond au nombre de points aléatoires et le nombre de colonnes au nombre de champs dans la table d'attributs du `SpatialPolygons`. Les lignes identifiées comme `NA` correspondent aux points qui ne sont pas contenus dans les polygones de la couche vectorielle. Par contre, les points aléatoires tombant à l'intérieur d'un polygone se voit affecter les attributs de ce polygone.

Rajoutons les identifiants des points aléatoires à cette table.

```
> # Rajout des identifiants
> join <- data.frame(ID = pts@data[, "ID"], join)
> # Verification
> head(join)

##        ID      Pays              Etat Code Population Capitale
## 1 Random1 Australia        Queensland   QL    4513009 Brisbane
## 2 Random2 Australia Western Australia   WA    2387232    Perth
## 3 Random3      <NA>              <NA> <NA>         NA     <NA>
## 4 Random4      <NA>              <NA> <NA>         NA     <NA>
## 5 Random5      <NA>              <NA> <NA>         NA     <NA>
## 6 Random6      <NA>              <NA> <NA>         NA     <NA>
```

Maintenant, nous n'allons conserver que les points superposant les polygones, c.-à-d. que nous allons supprimer dans la table « join » toutes lignes prenant la valeur `NA`.

```
> # Identification des lignes avec NA
> pos <- which(is.na(join[, "Pays"]))
> # Suppression de ces lignes
> join2 <- join[-pos, ]
> # Reinitialisation des noms de lignes
> rownames(join2) <- NULL
> # Verification
> head(join2)

##         ID      Pays               Etat Code Population Capitale
## 1  Random1 Australia         Queensland   QL    4513009 Brisbane
## 2  Random2 Australia  Western Australia   WA    2387232    Perth
## 3  Random7 Australia  Western Australia   WA    2387232    Perth
## 4 Random14 Australia  Western Australia   WA    2387232    Perth
## 5 Random21 Australia Northern Territory   NT     232365   Darwin
## 6 Random22 Australia    New South Wales   NS    7247669   Sydney
```

Nous pouvons convertir ce nouveau `data.frame` en un objet sp.

```
> join2 <- SpatialPointsDataFrame(coords = pts@coords[-pos, ],
+     data = join2)
> # Verification
> head(join2@data)

##         ID      Pays               Etat Code Population Capitale
## 1  Random1 Australia         Queensland   QL    4513009 Brisbane
## 2  Random2 Australia  Western Australia   WA    2387232    Perth
## 3  Random7 Australia  Western Australia   WA    2387232    Perth
## 4 Random14 Australia  Western Australia   WA    2387232    Perth
## 5 Random21 Australia Northern Territory   NT     232365   Darwin
## 6 Random22 Australia    New South Wales   NS    7247669   Sydney
```

Affichons maintenant ces points superposant les polygones de la couche vectorielle.

```
> par(mar = c(0, 0, 0, 0))
> plot(aus, col = "gray")
> plot(join2, pch = 19, add = T)
```

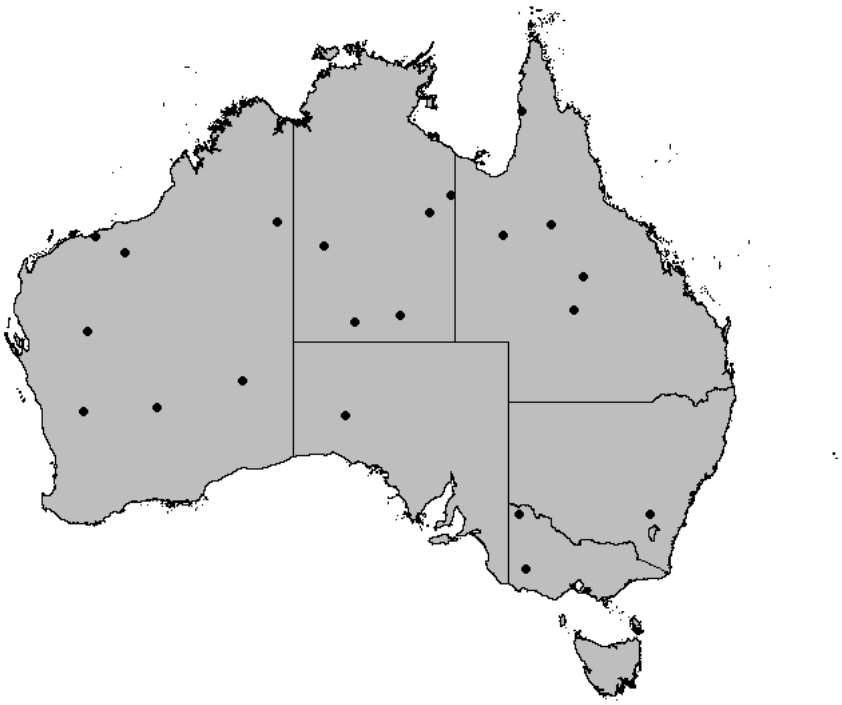

FIGURE 2.7 – Points aléatoires après jointure spatiale

Pour terminer, nous pouvons calculer le nombre de points superposant chaque état australien (seulement pour les états contenant au moins un point aléatoire).

```
> # Nombre de points par etat australien
> summary(as.factor(join2@data[, "Code"]))

## NS NT QL SA VI WA
##  2  3  6  1  1  5
```

Polygones-polygones

Voyons maintenant comment réaliser une jointure spatiale entre deux couches vectorielles formées de polygones. Le principe est exactement le même que ce que nous venons de voir. Nous allons créer un polygone spatiale de manière à le joindre spatialement à la couche vectorielle « aus ».

```
> # Creation d'un polygone
> x <- c(135.4732, 141.0291, 143.3934, 143.5116, 141.7384, 139.256)
> y <- c(-23.72366, -23.02321, -24.32404, -31.62868, -35.33104,
+     -34.73066)
> xy <- data.frame(x, y)
> xy <- rbind(xy, xy[1, ])
> # Conversion en SpatialPolygons
> pol <- SpatialPolygons(list(Polygons(list(Polygon(xy)), "Poly")))
> # Definition du systeme de coordonnees
> proj4string(pol) <- proj4string(aus)
```

Rajoutons ce polygone à la carte de l'Australie.

```
> par(mar = c(0, 0, 0, 0))
> plot(aus, col = "gray")
> plot(pol, border = "red", lwd = 2, add = T)
```

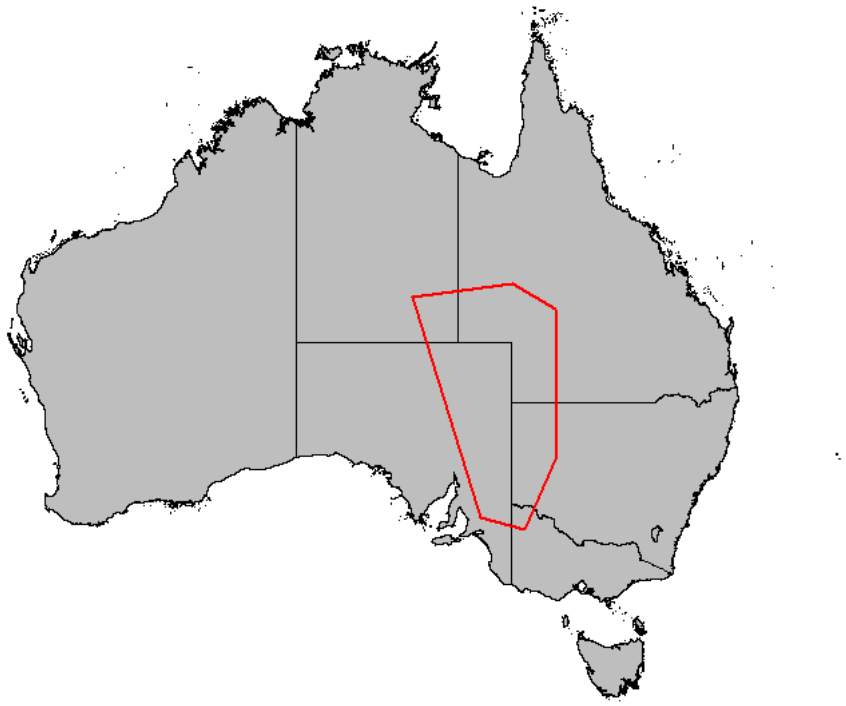

FIGURE 2.8 – Carte de l'Australie superposée d'un polygone

Nous pouvons maintenant procéder à la jointure spatiale entre ce polygone et les états australiens en utilisant la même fonction que précédemment.

```
> # Jointure spatiale
> (join <- over(aus, pol))

##  0  1  2  3  4  5  6  7  9 10
## NA NA NA  1  1  1  1 NA  1 NA
```

Il est important de remarquer que la jointure spatiale entre deux polygones spatiaux retournent un vecteur dont la longueur correspond au nombre de polygones multiples contenus dans le premier objet spatial (dans notre cas, il s'agit du nombre d'états australiens contenus dans « aus »). Les états qui ne superposent pas le polygone sont associés à la valeur `NA`, tandis que ceux qui touchent le polygone se voient affecter la valeur `1`. Identifions les états superposés par ce polygone.

```
> # Identification des etats superposes
> pos <- which(!is.na(join))
```

```
> par(mar = c(0, 0, 0, 0))
> plot(aus, col = "gray")
> plot(aus[pos, ], add = T, col = "black", border = "white")
> plot(pol, border = "red", lwd = 2, add = T)
```

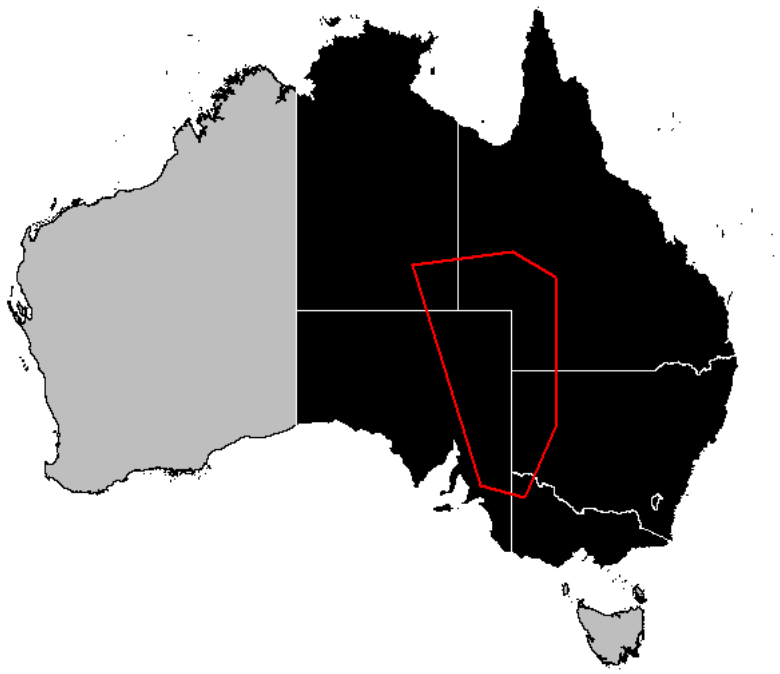

FIGURE 2.9 – Jointure spatiale entre polygones spatiaux

2.3.3 Intersection de couches

Dans certains cas, il peut être utile de ne conserver que la région intersectée entre deux couches vectorielles, c.-à-d. leur partie commune. Cela sera possible avec la fonction `gIntersection()` du *package* `rgeos`. Regardons tout d'abord comment réaliser une intersection entre deux objets vectoriels de type polylignes.

Intersection de lignes

Nous allons créer deux objets de type `SpatialLines` comme dans la Section 1.2.3. Définissons pour commencer les coordonnées de deux lignes qui vont s'entrecouper en plusieurs points.

```
> # Coordonnees de la premiere ligne
> x1 <- c(134.8964, 138.8433, 140.3327, 141.1518, 141.5986, 140.3327,
+     139.5135, 140.2582, 142.3433, 144.1306, 145.1731)
> y1 <- c(-23.80983, -23.34348, -23.94307, -25.67522, -28.40668,
+     -31.80435, -33.80299, -35.80162, -36.40121, -35.40189, -33.60312)
> c1 <- data.frame(x1, y1)
> # Coordonnees de la seconde ligne
> x2 <- c(138.173, 136.8326, 137.8007, 138.9922, 142.2689, 143.088,
+     142.7157, 139.8858, 138.8433, 141.0029, 144.354)
> y2 <- c(-21.34485, -25.40873, -27.34074, -27.87371, -27.00764,
+     -28.07358, -29.93897, -30.33869, -32.67043, -34.13609, -36.80093)
> c2 <- data.frame(x2, y2)
> # Conversion en SpatialLines
> L1 <- SpatialLines(list(Lines(list(Line(c1)), "Line1")))
> L2 <- SpatialLines(list(Lines(list(Line(c2)), "Line2")))
```

Nous allons maintenant définir le système de coordonnées de ces deux lignes dans le même système de coordonnées que la couche vectorielle des états australiens.

```
> # Definition du systeme de coordonnees
> L1@proj4string <- aus@proj4string
> L2@proj4string <- aus@proj4string
```

Nous pouvons maintenant procéder à l'intersection de nos deux lignes.

```
> # Intersection de lignes
> z <- gIntersection(L1, L2)
```

Regardons comment se structure le résultat de cette intersection.

```
> # Classe de l'objet
> class(z)

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

```
> # Affichage des coordonnees
> z@coords

##       x      y
## 1 137.5 -23.51
## 1 139.7 -33.27
## 1 140.9 -30.19
## 1 141.4 -27.24
## 1 143.2 -35.91
```

Vérifions le résultat en représentant les lignes et leurs intersections sur une carte.

```
> par(mar = c(0, 0, 0, 0))
> plot(aus, col = "gray")
> plot(L1, lwd = 2, col = "red", add = T)
> plot(L2, lwd = 2, col = "blue", add = T)
> plot(z, pch = 19, add = T)
```

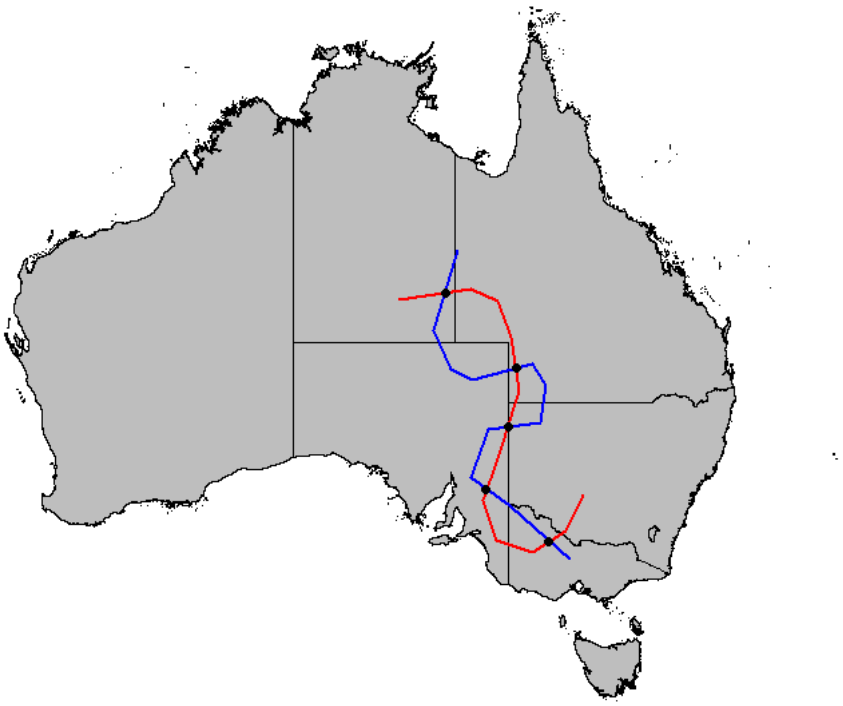

FIGURE 2.10 – Intersection de polylignes

Ainsi, l'intersection de deux lignes conduit à la création d'un objet spatial ponctuel qui fournit les coordonnées des points d'intersection. Nous allons voir maintenant que l'intersection de deux polygones donne un résultat sensiblement différent.

Intersection de polygones

Intersectons maintenant deux polygones entre eux. Nous allons utiliser la même fonction que pour intersecter deux lignes, la fonction `gIntersection()`. Notons dès à présent que cette fonction ne pourra s'effectuer qu'entre deux couches de polygones, chacune formée d'un seul polygone. Ce point sera important à considérer pour la dernière section de ce chapitre.

```
> # Creation du premier polygone
> x1 <- c(142.6764, 140.8663, 141.2284, 143.9434, 146.7489, 146.7489)
> y1 <- c(-21.89401, -25.53735, -31.44766, -34.60522, -31.1238,
+     -25.37542)
> c1 <- data.frame(x1, y1)
> c1 <- rbind(c1, c1[1, ])
> # Creation du second polygone
> x2 <- c(135.4732, 141.0291, 143.3934, 143.5116, 141.7384, 139.256)
> y2 <- c(-23.72366, -23.02321, -24.32404, -31.62868, -35.33104,
+     -34.73066)
> c2 <- data.frame(x2, y2)
> c2 <- rbind(c2, c2[1, ])
> # Conversion en SpatialPolygons
> pol1 <- SpatialPolygons(list(Polygons(list(Polygon(c1)), "Poly1")))
> pol2 <- SpatialPolygons(list(Polygons(list(Polygon(c2)), "Poly2")))
> # Definition du systeme de coordonnees
> proj4string(pol1) <- proj4string(aus)
> proj4string(pol2) <- proj4string(aus)
```

Réalisons l'intersection entre ces deux polygones de manière à identifier leur partie commune.

```
> # Intersection de polygones
> w <- gIntersection(pol1, pol2)
```

Regardons la structure de cette intersection.

```
> # Classe de l'objet
> class(w)

## [1] "SpatialPolygons"
## attr(,"package")
## [1] "sp"

> # Nom de ses elements
> slotNames(w)

## [1] "polygons"    "plotOrder"   "bbox"        "proj4string"
```

Ainsi, le résultat de l'intersection de deux polygones est également un polygone. Représentons le résultat de cette intersection sur la carte de l'Australie.

```
> par(mar = c(0, 0, 0, 0))
> plot(aus, col = "gray")
> plot(pol1, col = "black", border = NA, add = T)
> plot(pol2, col = "black", border = NA, add = T)
> plot(w, col = "#78A419", border = NA, add = T)
```

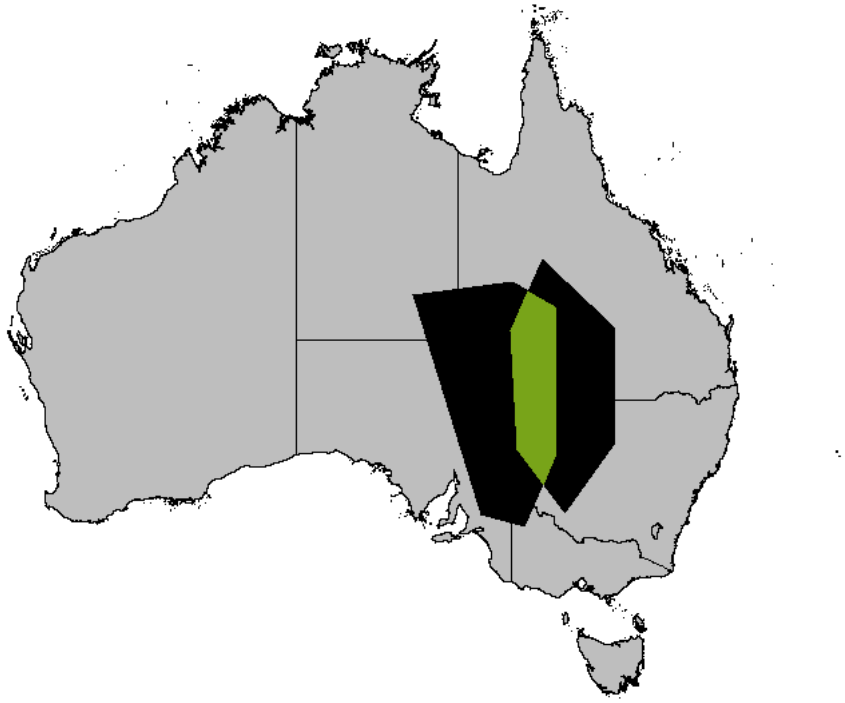

FIGURE 2.11 – Intersection de polygones

2.3.4 Découpage d'une couche

Terminons ce chapitre en voyant comment découper une couche vectorielle par une autre. Malheureusement, il n'existe pas de fonction prédéfinie sous R pour faire un tel *clip* d'objets spatiaux. Mais, nous allons voir qu'en combinant quelques fonctions vues précédemment, cette opération est tout à fait réalisable.

Découpage d'une couche vectorielle par un polygone

Nous allons découper les états australiens par un des deux polygones précédents, « pol2 ». Voici les étapes que nous suivrons :

- Jointure spatiale, afin d'identifier les états australiens superposés par le polygone ;
- Extraction par attributs, afin de placer chaque état intersecté par le polygone dans un objet spatial distinct ;

– Intersection de chaque état par le polygone, afin de ne conserver que la partie des états australiens située à l'intérieur du polygone ;
– Combinaison des différentes intersections pour créer une seule couche vectorielle découpée.

Allons-y pour la première étape : procédons à la jointure spatiale entre le shapefile des états australiens « aus » et le polygone « pol2 ».

```
> # Jointure spatiale
> (join <- over(aus, pol2))

##  0  1  2  3  4  5  6  7  9 10
## NA NA NA  1  1  1  1 NA  1 NA
```

La jointure spatiale permettra d'effectuer la suite des calculs uniquement sur les états d'intérêts. Ceci peut s'avérer extrêmement utile, notamment dans le cas d'un découpage très fin (donc avec de nombreux polygones). Conservons donc les états superposés par le polygone « pol2 ».

```
> # Identification des etats superposes
> pos <- which(!is.na(join))
> names(pos) <- NULL
> pos

## [1] 4 5 6 7 9
```

La seconde étape consiste en la création d'un objet spatial par état sélectionné. Nous allons donc procéder à une extraction par attribut de manière itérative et placer tous les nouveaux objets extraits dans une même `list`. Cette étape est nécessaire, car comme nous l'avons mentionné plus haut, la fonction `gIntersection()` éprouve de la difficulté à déterminer l'intersection entre des couches composées de plusieurs polygones.

```
> # Initialisation d'une liste vide
> states <- list()
> # Extraction par attribut
> for (i in 1:length(pos)) {
+     x <- aus[pos[i], ]
+     states[[i]] <- x
+ }
```

Maintenant, nous allons parcourir chaque `SpatialPolygons` contenu dans cette liste et l'intersecter avec le polygone « pol2 ». Le résultat de ces intersections successives sera contenu dans une nouvelle liste.

```
> # Initialisation d'une liste vide
> inters <- list()
> # Intersection de polygones
> for (i in 1:length(pos)) {
+     x <- gIntersection(states[[i]], pol2)
+     inters[[i]] <- x
+ }
```

La dernière étape va nous amener à combiner ces derniers `SpatialPolygons` en un seul et même objet spatial. Nous allons donc utiliser la fonction `spRbind()`. Cependant, la condition *sine qua non* d'une telle opération est que chaque polygone que nous allons combiner doit avoir un identifiant unique. Vérifions cela.

```
> # Affichage des identifiants des polygones a combiner
> for (i in 1:length(pos)) {
+     print(sapply(inters[[i]]@polygons, function(x) slot(x, "ID")))
+ }

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

Les cinq `SpatialPolygons` (états australiens découpés) possèdent le même identifiant (résultat de l'intersection). Nous devons donc leur attribuer un identifiant unique.

```
> # Attribution d'un nouvel identifiant aux polygones
> for (i in 1:length(pos)) {
+     inters[[i]] <- spChFIDs(inters[[i]], as.character(i))
+ }
> # Verification
> for (i in 1:length(pos)) {
+     print(sapply(inters[[i]]@polygons, function(x) slot(x, "ID")))
+ }

## [1] "1"
## [1] "2"
## [1] "3"
## [1] "4"
## [1] "5"
```

Pour terminer, regroupons tous ces polygones dans un objet spatial que nous appellerons « clip ».

```
> # Initialisation de l'objet 'clip'
> clip <- inters[[1]]
> # Combinaison iterative des polygones
> for (i in 2:length(pos)) {
+     clip <- spRbind(clip, inters[[i]])
+ }
```

Visualisons le résultat de ce découpage.

```
> par(mar = c(0, 0, 0, 0))
> plot(aus, col = "gray", border = "white")
> plot(clip, border = "black", col = "#78A419", add = T)
```

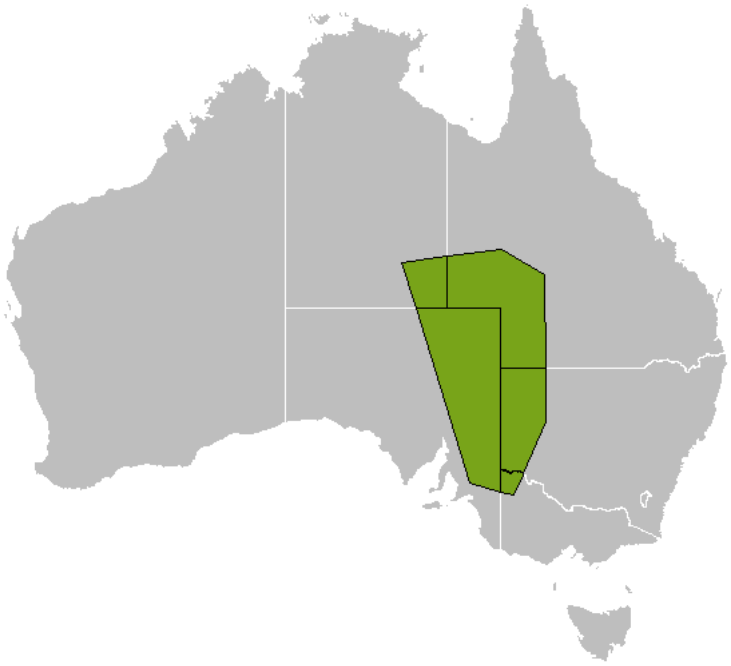

FIGURE 2.12 – Découpage d'une couche vectorielle

Lors de l'étape d'intersection, la table d'attributs de chaque état a été perdu. Mais, comme l'ordre de traitement des différents états est conservé à chaque étape, nous pouvons facilement recréer cette table d'attributs et l'ajouter à l'objet « clip ».

```
> # Table d'attributs
> mat <- aus@data[pos, ]
> # Nom des lignes
> rownames(mat) <- 1:5
> # Rajout de la table a l'objet
> clip <- SpatialPolygonsDataFrame(clip, mat)
> # Verification
> clip@data

##        Pays               Etat Code Population  Capitale
## 1 Australia    New South Wales   NS    7247669    Sydney
## 2 Australia Northern Territory   NT     232365    Darwin
## 3 Australia         Queensland   QL    4513009  Brisbane
## 4 Australia    South Australia   SA    1645040  Adelaide
## 5 Australia           Victoria   VI    5574455 Melbourne
```

Implémentation d'une fonction générique

Nous venons de voir comment clipper une couche vectorielle par un polygone. Nous allons maintenant modifier le code précédent de manière à ce qu'il soit beaucoup plus général et permette le découpage entre couches vectorielles, chacune composée d'une multitude de polygones. Nous allons pour cela regrouper les différentes lignes de code en un bloc d'instructions définissant une fonction que nous appellerons `clip.poly()`. Cette fonction recevra trois arguments :

- `spdf`, la couche vectorielle à découper ;
- `clip`, la couche vectorielle qui découpera la seconde ;
- `Id`, le nom de la colonne dans la table d'attributs de `spdf` qui contient l'identifiant de chaque polygone multiple contenu dans cet objet. Si on prend l'exemple de la couche vectorielle donnant la délimitation des états/territoires australiens (« aus »), il pourrait s'agir du code de chaque état (colonne « Code »).

Cette fonction retournera un `SpatialPolygonsDataFrame`.

Définissons cette fonction `clip.poly()`.

```
> clip.poly <- function(spdf, clip, Id) {
+     # ================================
+     nb <- length(clip@polygons)
+     k <- 0
+     COMB <- list()
+     # Decoupage par polygone du clip
+     for (i in 1:nb) {
+         # ieme polygone du clip
+         xx <- clip[i, ]
+         # Jointure spatiale
+         pos <- which(!is.na(over(spdf, xx)))
+         # Extraction par polygone de spdf superpose
+         states <- list()
+         for (j in 1:length(pos)) states[[j]] <- spdf[pos[j], ]
+         # Intersection
+         inters <- list()
+         for (j in 1:length(pos)) {
+             inters[[j]] <- gIntersection(states[[j]], xx)
+         }
+         # Identifiants uniques
+         for (j in 1:length(pos)) {
+             inters[[j]] <- spChFIDs(inters[[j]], as.character(k))
+             k <- k + 1
+         }
+         # Combinaison des polygones de spdf decoupes
+         comb <- inters[[1]]
+         for (j in 2:length(pos)) comb <- spRbind(comb, inters[[j]])
+         # Rajout de la table d'attributs
+         mat <- spdf@data[pos, ]
+         z <- sapply(comb@polygons, function(x) slot(x, "ID"))
+         rownames(mat) <- z
```

```
+         comb <- SpatialPolygonsDataFrame(comb, mat)
+         # Placement temporaire dans la liste
+         COMB[[i]] <- comb
+     }
+     # Combinaison de tous les decoupages
+     CLIP <- COMB[[1]]
+     if (nb > 1) {
+         for (j in 2:length(COMB)) CLIP <- spRbind(CLIP, COMB[[j]])
+     }
+     # Dissolution par id
+     temp <- gUnionCascaded(CLIP, id = as.character(CLIP@data[, Id]))
+     # Rajout d'une table d'attributs
+     id <- sapply(temp@polygons, function(x) slot(x, "ID"))
+     mat <- data.frame()
+     for (k in 1:length(id)) {
+         pos <- which(CLIP$Code == id[k])
+         mat <- rbind(mat, CLIP@data[pos[1], ])
+     }
+     rownames(mat) <- id
+     CLIP <- SpatialPolygonsDataFrame(temp, mat)
+     # Retour de la fonction
+     return(CLIP)
+ }
```

Nous pouvons maintenant utiliser cette fonction. Pour cela, nous allons créer une couche vectorielle composée de six polygones contigus qui viendra clipper la couche vectorielle des états australiens. Créons cet objet spatial.

```
> # Coordonnees du premier polygone
> x0 <- c(136, 139, 139, 136, 136)
> y0 <- c(-20.5, -20.5, -22.5, -22.5, -20.5)
> # Coordonnees du second polygone
> x1 <- c(136, 139, 139, 136, 136)
> y1 <- c(-22.5, -22.5, -24.5, -24.5, -22.5)
> # Coordonnees du troisieme polygone
> x2 <- c(136, 139, 139, 136, 136)
> y2 <- c(-24.5, -24.5, -26.5, -26.5, -24.5)
> # Coordonnees du quatrieme polygone
> x3 <- c(139, 142, 142, 139, 139)
> y3 <- c(-24.5, -24.5, -26.5, -26.5, -24.5)
> # Coordonnees du cinquieme polygone
> x4 <- c(139, 142, 142, 139, 139)
> y4 <- c(-26.5, -26.5, -28.5, -28.5, -26.5)
> # Coordonnees du sixieme polygone
> x5 <- c(139, 142, 142, 139, 139)
> y5 <- c(-28.5, -28.5, -30.5, -30.5, -28.5)
> # Conversion en polygones multiples
> P0 <- Polygons(list(Polygon(data.frame(x0, y0))), "Pol0")
```

```
> P1 <- Polygons(list(Polygon(data.frame(x1, y1))), "Pol1")
> P2 <- Polygons(list(Polygon(data.frame(x2, y2))), "Pol2")
> P3 <- Polygons(list(Polygon(data.frame(x3, y3))), "Pol3")
> P4 <- Polygons(list(Polygon(data.frame(x4, y4))), "Pol4")
> P5 <- Polygons(list(Polygon(data.frame(x5, y5))), "Pol5")
> # Conversion en polygone spatial
> pol3 <- SpatialPolygons(list(P0, P1, P2, P3, P4, P5))
> proj4string(pol3) <- proj4string(aus)
```

Superposons cette couche vectorielle à la carte des états australiens.

```
> par(mar = c(0, 0, 0, 0))
> plot(aus, border = "gray")
> plot(pol3, add = T, col = NA)
```

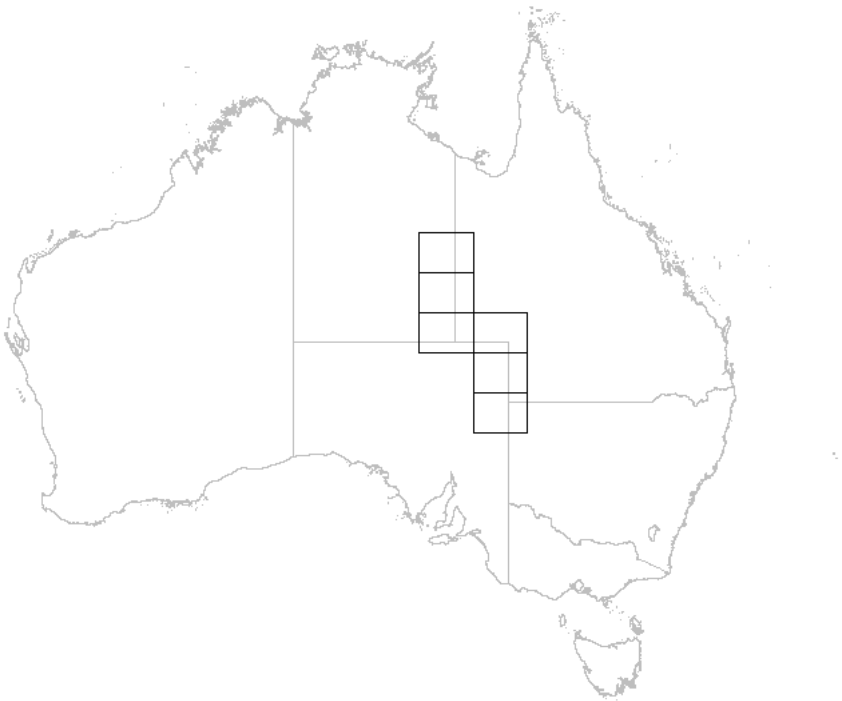

FIGURE 2.13 – Couche vectorielle multipolygonale

Procédons maintenant au découpage de l'objet « aus » par le polygone « pol3 ».

```
> aus_clip <- clip.poly(spdf = aus, clip = pol3, Id = "Code")
```

Regardons la table d'attributs de l'objet clippé.

```
> # Table d'attributs
> aus_clip@data

##         Pays                Etat Code Population Capitale
## NS Australia    New South Wales   NS    7247669   Sydney
## NT Australia Northern Territory   NT     232365   Darwin
## QL Australia         Queensland   QL    4513009 Brisbane
## SA Australia    South Australia   SA    1645040 Adelaide
```

Pour terminer, cartographions le résultat du découpage de deux couches vectorielles composées chacune de plusieurs polygones.

```
> par(mar = c(0, 0, 0, 0))
> plot(aus, col = "gray", border = "white")
> plot(clip, border = "black", col = "#78A419", add = T)
```

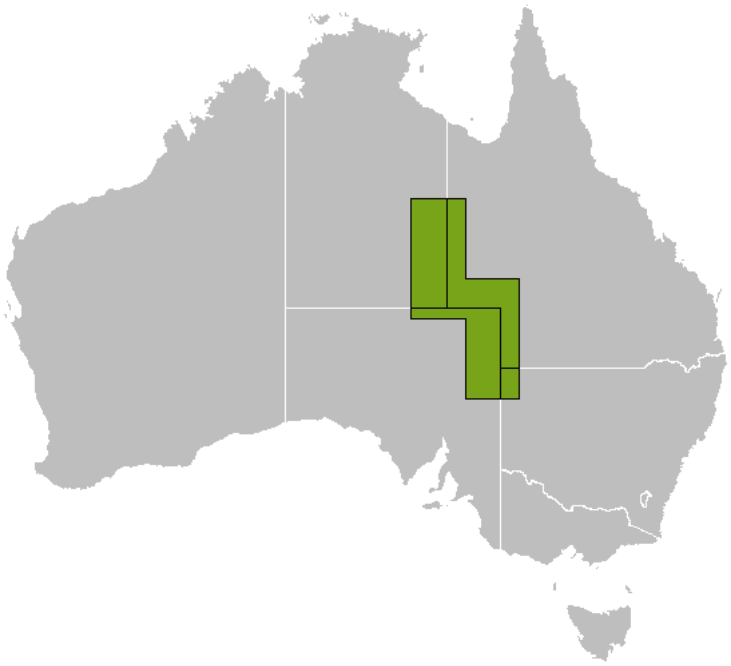

FIGURE 2.14 – Clip par une couche vectorielle multipolygonale

Chapitre 3

Cartographie sous R

3.1 Cartographie d'objets vectoriels

R n'est pas un logiciel de SIG. Mais, comme nous l'avons vu dans les chapitres précédents, il peut se comporter comme tel. Nous avons étudié la structure des objets spatiaux sous R, et exploré différents outils permettant d'importer, exporter et manipuler la géométrie de couches vectorielles et matricielles. Nous allons maintenant nous intéresser aux aspects cartographiques. Nous avons déjà produit quelques cartes rudimentaires en affichant simplement des objets spatiaux au moyen de la fonction de base `plot()`. Nous allons voir dans ce chapitre comment rendre ces cartes plus professionnelles et plus esthétiques.

R est un langage de programmation complet : ceci implique que toutes les caractéristiques d'une carte seront paramétrables et ajustables. Les classes d'objets spatiaux introduites par les *packages* `sp` et `raster` ont été construites de telle manière que la plupart des fonctions graphiques de base peuvent être utilisées pour représenter ces objets. Ainsi, les fonctions `plot()`, `points()`, `rect()`, `text()` ou encore `legend()` pourront servir à représenter des couches géographiques. De même, la plupart des paramètres graphiques seront modifiables grâce à la fonction `par()`.

Dans cette première section, nous allons produire une carte administrative des états australiens en respectant les normes et standards cartographiques. Ainsi, nous allons voir comment ajouter à la carte de base un titre, une légende, une échelle, les graticules, ainsi qu'une rose des vents. Regardons cela plus en détails.

3.1.1 Carte basique

Commençons par importer les couches vectorielles dont nous allons avoir besoin dans cette section. Nous allons importer deux couches vectorielles : « AUS_adm0 », qui donne les limites nationales de l'Australie, et « AUS_adm1 », qui donne les délimitations administratives des états/territoires australiens. Nous allons également supprimer les polygones dans la seconde couche vectorielle qui ne possèdent pas d'attributs (voir section 2.2.2).

```
> # Repertoire contenant les donnees
> dest <- paste(root, "/Australia", sep = "")
> # Importation des limites nationales
> aus0 <- readOGR(dsn = dest, layer = "AUS_adm0", verbose = F)
> # Importation des limites régionales
> aus1 <- readOGR(dsn = dest, layer = "AUS_adm1", verbose = F)
```

```
> # Suppression de polygones
> aus1 <- aus1[-9, ]
```

Nous allons également avoir besoin des positions géographiques des capitales régionales australiennes. Nous allons donc importer la table de données « townsAUS.txt » et convertir les données en `SpatialPointsDataFrame`.

```
> # Importation des capitales australiennes
> tab <- read.delim(paste(root, "/townsAUS.txt", sep = ""))
> # Conversion en objet spatial avec table d'attributs
> dat <- SpatialPointsDataFrame(coords = tab[, 2:3], data = tab[,
+     c(1, 4, 5)])
> # Definition du systeme de coordonnees
> proj4string(dat) <- proj4string(aus0)
```

Tout d'abord, paramétrons une nouvelle fenêtre graphique. Nous allons utiliser pour cela la fonction R `par()`. Celle-ci permet de configurer un peu plus de 70 paramètres graphiques. Pour plus d'information sur les options graphiques, on pourra consulter l'aide de cette fonction (`?par`). Pour cet exercice, nous n'allons modifier que les paramètres suivants :

- ***mar*** permet de contrôler les marges de la zone graphique ;
- ***xaxs*** permet de contrôler pour le style de l'axe des abscisses ;
- ***yaxs*** permet de contrôler pour le style de l'axe des ordonnées.

Voici comment modifier les paramètres d'une fenêtre graphique.

```
> # Configuration de la zone graphique
> par(xaxs = "i", yaxs = "i", mar = c(2, 3, 2, 3))
```

Nous allons maintenant tracer la couche vectorielle des états australiens (soustraite des polygones inconnus). Pour l'instant, nous n'allons pas personnaliser cet objet : il nous sert simplement à définir l'étendue spatiale de notre carte. Pour représenter cet objet spatial, nous allons utiliser la fonction R `plot()` en spécifiant deux arguments :

- ***xlim*** permet de définir les limites minimale et maximale de l'axe des abscisses ;
- ***ylim*** permet de définir les limites minimale et maximale de l'axe des ordonnées.

```
> # Definition de l'etendue spatiale de la carte
> plot(aus1, xlim = c(110, 160), ylim = c(-50, -10))
```

Maintenant, nous allons rajouter un rectangle bleu pour définir les zones marines bordant l'Australie. Nous allons utiliser pour cela la fonction `polygon()`, qui ne doit pas être confondue avec la fonction `Polygon()` du *package* `sp` qui s'écrit avec une majuscule. Cette fonction accepte de nombreux arguments, dont :

- ***x***, les coordonnées en abscisse des différents sommets du polygone ;
- ***y***, les coordonnées en ordonnée des différents sommets du polygone ;
- ***col***, la couleur intérieure du polygone (ici, nous allons tracer un polygone rempli d'un bleu océan) ;
- ***border***, la couleur de la bordure du polygone.

```
> # Ajout des mers-oceans
> polygon(x = c(110, 160, 160, 110, 110), y = c(-10, -10, -50,
+     -50, -10), col = "#bbe6fb", border = "transparent")
```

Nous allons superposer à ce polygone la couche vectorielle des états/territoires australiens grâce à la fonction de base `plot()` dans laquelle nous allons spécifier l'argument ***add*** $=$ ***T***. Cet argument va indiquer à R qu'il doit rajouter cet objet à la fenêtre graphique actuellement active. En effet, par défaut, la fonction `plot()` ouvre une nouvelle fenêtre graphique (en fait, elle écrase l'ancienne fenêtre graphique). C'est la grande différence avec les fonctions `polygon()`, `points()`, `rect()`, etc. qui elles permettent d'ajouter directement des éléments à une fenêtre graphique ouverte. D'ailleurs, ces dernières retourneront un message d'erreur si aucune fenêtre n'est ouverte.

Regardons cela.

```
> # Ajout de l'Australie
> plot(aus1, col = "#fff79c", border = "gray", add = T)
```

FIGURE 3.1 – Limites administratives de l'Australie

Pour terminer, nous allons colorier le contour maritime de l'Australie avec un bleu plus foncé. Mais, nous voulons que les limites terrestres entre les états/territoires restent dans leur couleur actuelle. Nous allons donc superposer la couche vectorielle qui fournit les limites nationales de l'Australie en transparence.

```
> # Rajout du contour de l'Australie
> plot(aus0, col = NULL, border = "#34bdf2", add = T)
```

FIGURE 3.2 – Australie avec bordures maritimes

3.1.2 Ajout de graticules

Nous allons voir maintenant comment rajouter des graticules à cette carte, c.-à-d. les méridiens et les parallèles. Le *package* `sp` met à notre disposition une fonction pour cela : la fonction `gridlines()`. Cependant, nous n'allons pas utiliser directement cette fonction, car elle n'offre que peu de paramètres pour la personnaliser. Nous allons donc écrire une fonction, qui s'inspire en grande partie de celle-ci, mais qui siéra mieux à nos besoins. Écrivons cette fonction avant de l'expliquer.

```
> graticules <- function(sp, WE, NS) {
+     # ============================
+     WE.l <- NS.l <- list()
+     # Definition des longitudes
+     for (i in 1:length(WE)) {
+         x <- rep(WE[i], 20)
+         y <- seq(NS[1], NS[length(NS)], length.out = 20)
+         xy <- data.frame(x, y)
```

```
+         WE.l[[i]] <- Line(xy)
+     }
+     # Definition des latitudes
+     for (i in 1:length(NS)) {
+         x <- seq(WE[1], WE[length(WE)], length.out = 20)
+         y <- rep(NS[i], 20)
+         xy <- data.frame(x, y)
+         NS.l[[i]] <- Line(xy)
+     }
+     # Conversion en objet spatial
+     SL <- SpatialLines(list(Lines(NS.l, "NS"), Lines(WE.l, "WE")),
CRS(proj4string(sp)))
+     return(SL)
+ }
```

Cette fonction va retourner un objet de la classe `SpatialLines` contenant deux séries de lignes spatiales : les méridiens, notés « NS », et les parallèles, notés « WE ». L'utilisateur devra indiquer la valeur de trois arguments :

- ***sp***, un des objets actuellement représenté sur la carte. Celui ne servira qu'à définir le système de coordonnées des graticules ;
- ***WE***, les coordonnées des latitudes pour lesquelles on souhaite tracer les parallèles ;
- ***NS***, les coordonnées des longitudes pour lesquelles on souhaite tracer les méridiens.

L'avantage de cette fonction devant celle du *package* `sp` est que les graticules ne sont pas limités à l'étendue spatiale de l'objet représenté, mais ils peuvent s'ajuster à l'étendue spatiale de la carte (étendues spatiales qui ne correspondent pas nécessairement).

Utilisons cette fonction et calculons le positionnement des parallèles et des méridiens tous les 10 degrés.

```
> # Calcul des graticules
> grat <- graticules(sp = aus1, WE = seq(from = 110, to = 160,
+     by = 10), NS = seq(from = -50, to = -10, by = 10))
```

Et rajoutons les graticules à notre carte.

```
> # Rajout des graticules
> plot(grat, lty = 3, col = "#34bdf2", add = T)
```

Cette fonction ne fait que tracer les méridiens et les parallèles sans indiquer leurs valeurs sur la carte. Nous allons donc rajouter cette information grâce à la fonction `axis()`. Celle-ci accepte plusieurs arguments :

- ***side*** qui indique l'axe devant être tracé. Les valeurs possibles sont : 1 (axe du bas), 2 (axe de gauche), 3 (axe du haut) ou 4 (axe de droite) ;
- ***at*** qui donne la graduation de l'axe ;
- ***labels*** qui spécifie le nom des étiquettes sur la graduation définie avec l'argument précédent ;
- ***pos*** la coordonnée sur l'axe perpendiculaire à laquelle doit être tracé cet axe.

Affichons les quatre axes.

```
> # Definition des etiquettes de l'axe inferieur
> labs <- paste(seq(110, 160, by = 10), "E", sep = "")
> # Rajout de l'axe inferieur
> axis(side = 1, pos = -50, at = seq(110, 160, by = 10), labels = labs)
> # Definition des etiquettes de l'axe de gauche
> labs <- paste(seq(50, 10, by = -10), "S", sep = "")
> # Rajout de l'axe de gauche
> axis(side = 2, pos = 110, at = seq(-50, -10, by = 10), labels = labs)
> # Definition des etiquettes de l'axe superieur
> labs <- paste(seq(110, 160, by = 10), "E", sep = "")
> # Rajout de l'axe superieur
> axis(side = 3, pos = -10, at = seq(110, 160, by = 10), labels = labs)
> # Definition des etiquettes de l'axe de droite
> labs <- paste(seq(50, 10, by = -10), "S", sep = "")
> # Rajout de l'axe de droite
> axis(side = 4, pos = 160, at = seq(-50, -10, by = 10), labels = labs)
```

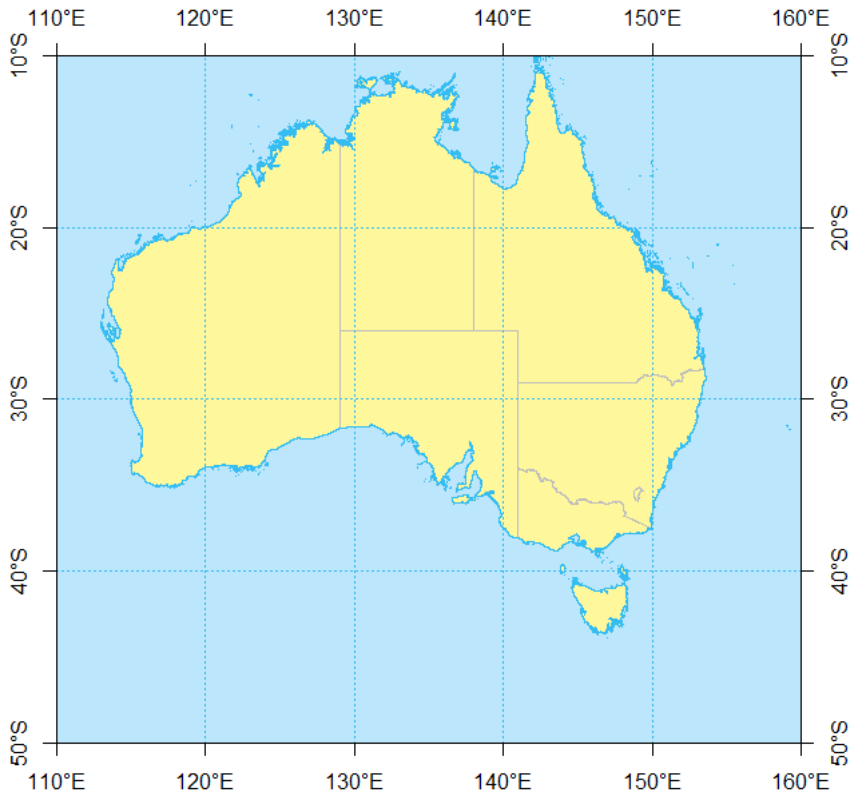

FIGURE 3.3 – Carte avec graticules

3.1.3 Ajout de texte

Regardons maintenant comment rajouter des informations textuelles sur la carte. Commençons par ajouter les capitales des différents états/territoires australiens.

```
> # Rajout des capitales
> plot(dat, pch = 19, add = T)
```

Nous allons ensuite ajouter le nom de ces villes sur la carte. Pour ce faire, nous allons utiliser la fonction `text()`, qui reçoit plusieurs arguments :

- ***x***, ***y*** correspondent aux coordonnées auxquelles seront rajouter les noms des villes ;
- ***labels*** spécifie les expressions textuelles que nous souhaitons ajouter à la carte (c.-à-d. le nom des villes) ;
- ***pos*** indique la position par rapport aux coordonnées précédentes où l'expression textuelle doit être tracée. Cet argument peut prendre les valeurs 1 (en bas du point), 2 (à gauche), 3 (en haut) ou 4 (à droite).

```
> # Extraction des coordonnees
> lon <- dat@coords[, "Longitude"]
> lat <- dat@coords[, "Latitude"]
> # Extraction du nom des villes
> labs <- as.character(dat@data[, "Ville"])
> # Rajout des noms des capitales sur la carte
> text(x = lon, y = lat, labels = labs, pos = 4, font = 2, cex = 0.75)
```

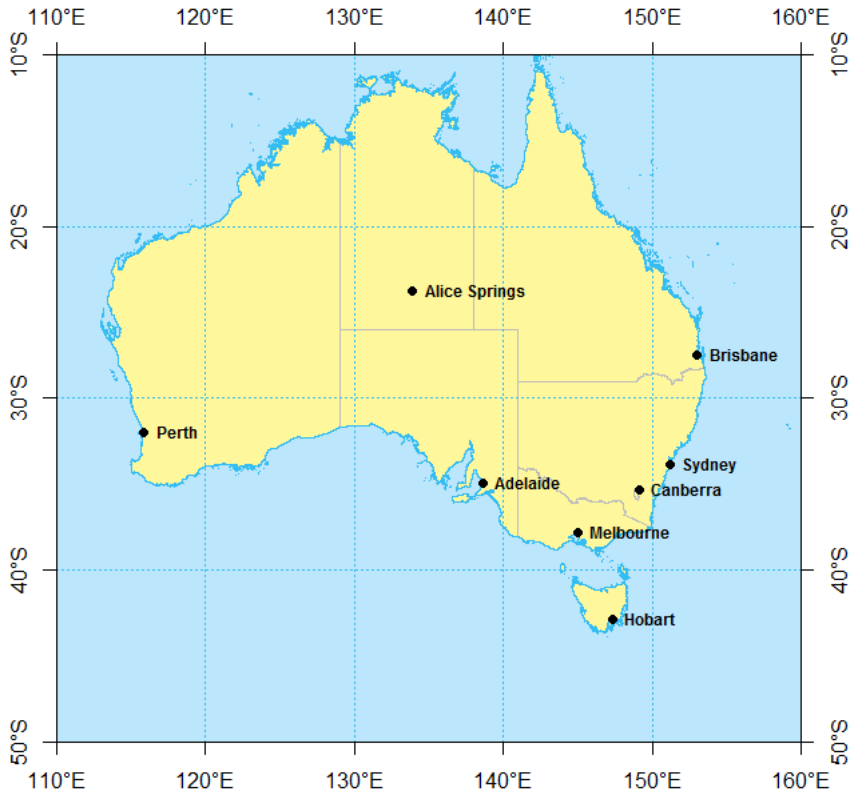

FIGURE 3.4 – Australie avec ses principales villes

Afin de rendre cette carte un peu plus professionnelle, nous allons également rajouter le nom des océans et des principales mers bordant l'Australie. Pour écrire une expression

textuelle sur plusieurs lignes, on aura recours au caractère échappé formé d'une barre oblique inverse suivie de la lettre « n », qui indiquera à R qu'il devra effectuer un retour à ligne. Regardons cela.

```
> # Rajout de l'Ocean Indien
> text(x = 130, y = -42.5, labels = "OCÉAN INDIEN", font = 3,
+     col = "#34bdf2")
> # Rajout de l'Ocean Pacifique
> text(x = 155, y = -20, labels = "OCÉAN\nPACIFIQUE", font = 3,
+     col = "#34bdf2")
```

Avant d'afficher cette carte, nous allons rajouter d'autres informations relatives aux zones marines, en plaçant sur la carte le nom de quatre mers bordant le continent australien. Nous allons tout d'abord créer une table avec le nom de ces mers ainsi que leurs positions géographiques.

```
> # Creation d'une table avec coordonnees et noms des mers
> mer <- as.data.frame(matrix(ncol = 3, nrow = 4))
> colnames(mer) <- c("Nom", "Longitude", "Latitude")
> mer[, "Nom"] <- c("Grande Baie\naustralienne", "Mer de Timor",
+     "Mer de\nCorail", "Mer de\nTasman")
> mer[, "Longitude"] <- c(130, 115, 150, 155)
> mer[, "Latitude"] <- c(-35, -17.5, -15, -42.5)
> # Affichage de la table
> mer

##                          Nom Longitude Latitude
## 1 Grande Baie\naustralienne       130    -35.0
## 2              Mer de Timor       115    -17.5
## 3            Mer de\nCorail       150    -15.0
## 4            Mer de\nTasman       155    -42.5
```

Procédons de la même manière que pour les océans (appel à la fonction `text()`) et ajoutons sur la carte le nom de ces quatre mers.

```
> # Extraction des coordonnees
> lon <- mer[, "Longitude"]
> lat <- mer[, "Latitude"]
> # Extraction des etiquettes
> labs <- as.character(mer[, "Nom"])
> # Placement des noms des mers sur la carte
> text(x = lon, y = lat, labels = labs, cex = 0.75, col = "#34bdf2",
+     font = 3)
```

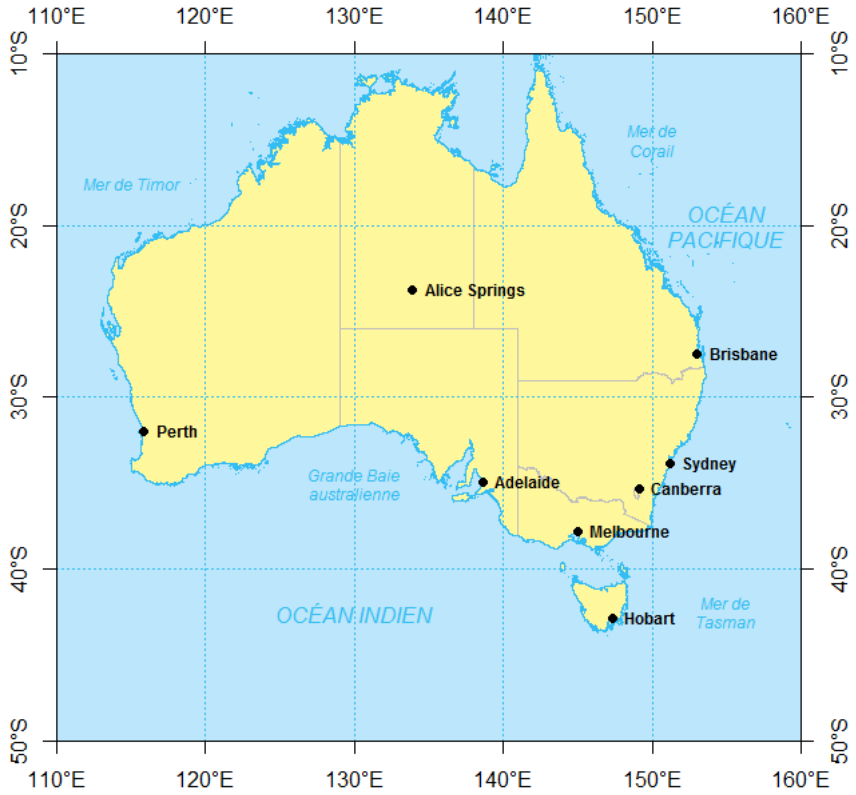

FIGURE 3.5 – Principales mers bordant l'Australie

3.1.4 Ajout d'une échelle

Rendons cette carte un peu plus professionnelle, et ajoutons une échelle cartographique. Le *package* `raster` met à notre disposition la fonction `scalebar()`. Celle-ci accepte plusieurs arguments :

- ***d***, la distance couverte par l'échelle ;
- ***xy***, la position géographique à laquelle l'échelle doit être tracée (bord gauche) ;
- ***type***, indique le type d'échelle : ligne (***line***) ou barre (***bar***) ;
- ***below***, le texte à rajouter sous l'échelle ;
- ***divs***, le nombre de divisions de l'échelle ;
- ***lonlat***, indique si la carte est en représentation plane (***F***) ou dans un système non projeté (***T***).

```
> # Rajout d'une echelle
> scalebar(d = 500, xy = c(152.5, -47.5), type = "bar", below = "km",
+     lwd = 4, divs = 2, col = "black", cex = 0.75, lonlat = T)
```

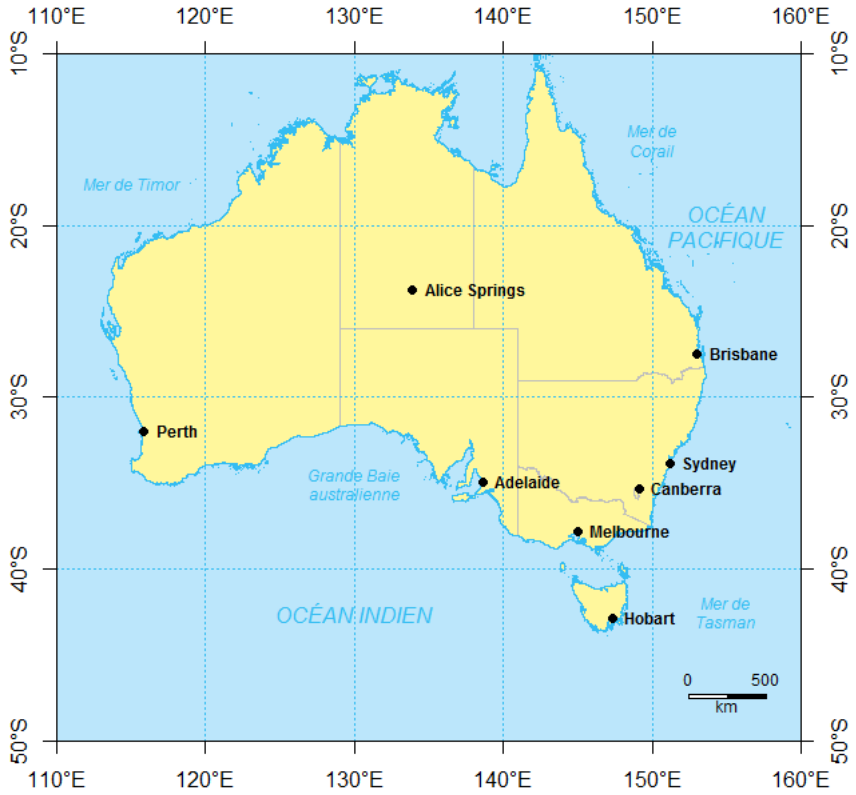

FIGURE 3.6 – Carte avec échelle cartographique

3.1.5 Ajout d'une rose des vents

Rajoutons maintenant une rose des vents qui nous indiquera la direction du Nord géographique. Dans le *Journal of Statistical Software* du mois d'Avril 2007, Tanimura *et al.*[1] proposent plusieurs fonctions cartographiques auxiliaires, dont la fonction `northarrow()`. Celle-ci n'étant pas implémentée dans un *package*, recopions-là ici afin de la définir dans notre session de travail.

```
> northarrow <- function(loc, size, bearing = 0, cex = 1) {
+     # ========================================
+     cols <- rep(c("white", "black"), 8)
+     # Coordonnees des polygones de la rose des vents
+     radii <- rep(size/c(1, 4, 2, 4), 4)
+     x <- radii[(0:15) + 1] * cos((0:15) * pi/8 + bearing) + loc[1]
+     y <- radii[(0:15) + 1] * sin((0:15) * pi/8 + bearing) + loc[2]
+     # Trace des polygones
+     for (i in 1:15) {
+         x1 <- c(x[i], x[i + 1], loc[1])
+         y1 <- c(y[i], y[i + 1], loc[2])
```

1. Tanimura, S. *et al.* (2007) Auxilary Cartographic Functions in R : North Arrow, Scale Bar, and Label with Leader Arrow. *Journal of Statistical Software*, Vol.19, Code Snippet 1, 1-8.

```
+         polygon(x1, y1, col = cols[i])
+     }
+     # Trace du dernier polygone
+     x1 <- c(x[16], x[1], loc[1])
+     y1 <- c(y[16], y[1], loc[2])
+     polygon(x1, y1, col = cols[16])
+     # Rajout des lettres
+     b <- c("E", "N", "O", "S")
+     for (i in 0:3) {
+         text((size + par("cxy")[1]) * cos(bearing + i * pi/2) + loc[1],
+             (size + par("cxy")[2]) * sin(bearing + i * pi/2) + loc[2],
+             b[i + 1], cex = cex)
+     }
+ }
```

Exécutons maintenant cette fonction pour rajouter une rose des vents à notre carte.

```
> # Ajout du Nord geographique
> northarrow(loc = c(115, -45), size = 3, cex = 0.75)
```

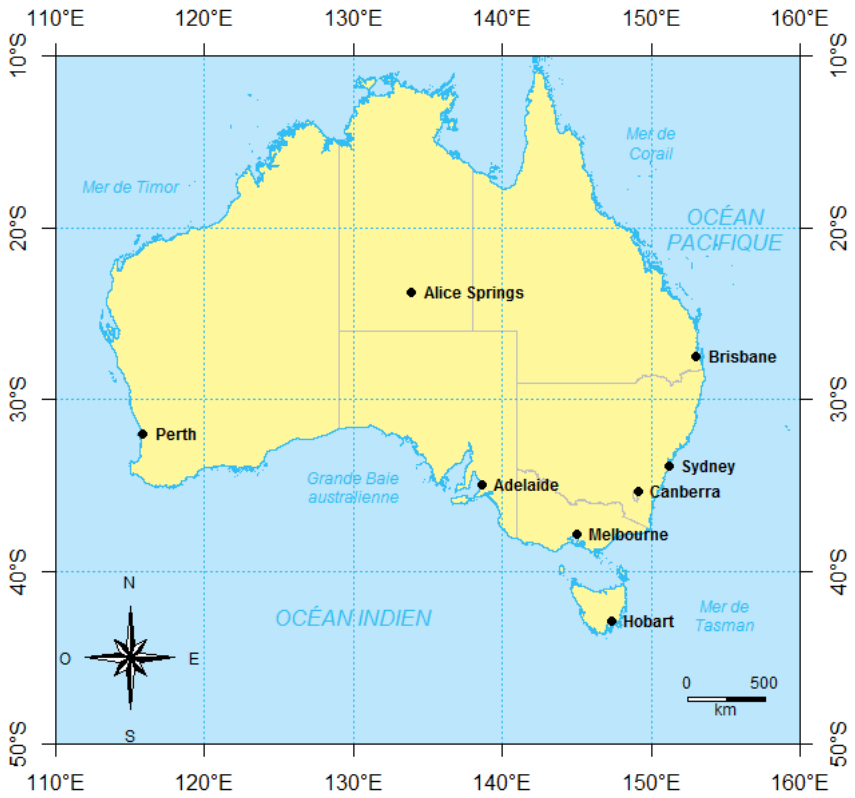

FIGURE 3.7 – Carte avec rose des vents

3.1.6 Ajout d'une légende

Bien que cela ne soit pas nécessaire dans notre cas, nous allons rajouter une légende à notre carte. Pour ce faire, nous allons utiliser la fonction `legend()`, laquelle accepte plusieurs arguments, dont voici les principaux :

- ***x***, la coordonnée en abscisse à laquelle doit être placée la légende ;
- ***y***, la coordonnée en ordonnée à laquelle doit être placée la légende ;
- ***legend***, le nom des différents items de la légende ;
- ***pch***, le symbole des différents items de la légende ;
- ***bg***, la couleur du fond du cadre de la légende ;
- ***title***, le titre de la légende.

Rajoutons donc une légende à notre carte avec les villes comme unique item.

```
> legend(111, -10.75, legend = "Capitale régionale", pch = 19,
+     bg = "white", cex = 0.75, title = "LÉGENDE")
```

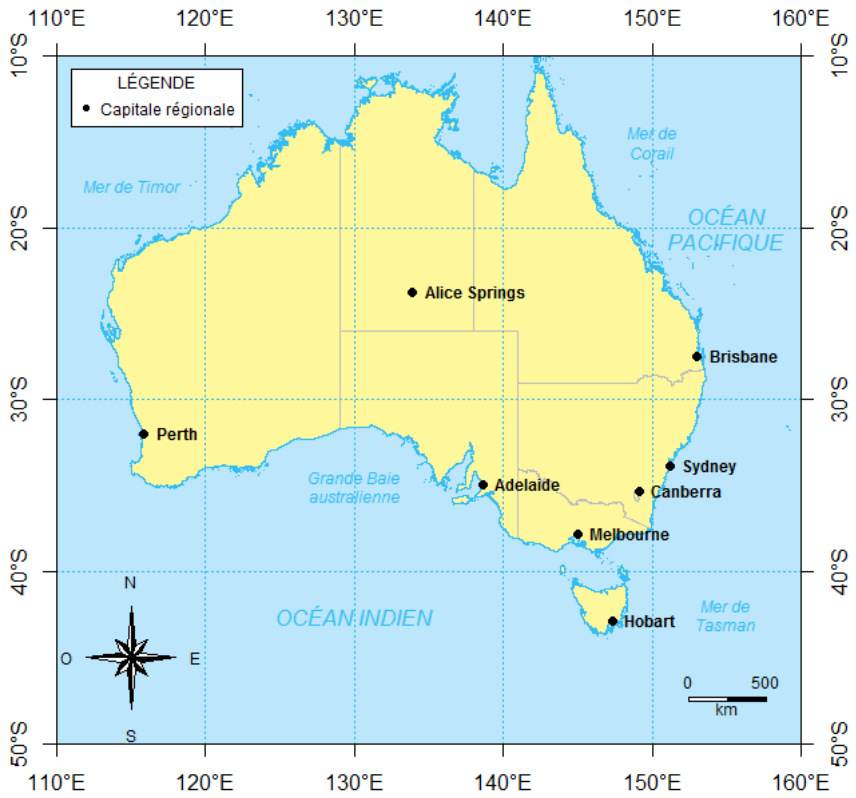

FIGURE 3.8 – Carte avec légende

3.1.7 Ajout d'un titre

Que serait une carte sans un titre ? Pour terminer cette section, nous allons rajouter un titre à notre représentation cartographique des limites administratives de l'Australie.

Nous pourrions utiliser la fonction `title()`, mais nous allons préférer la fonction que nous avons utilisé précédemment : `text()`. Avant de rajouter cette zone de texte, nous allons tracer un rectangle qui servira de cadre à notre titre. Pour ce faire, nous allons utiliser la fonction `rect()`, qui requière les coordonnées extrêmes du rectangle (nous pourrions également utiliser la fonction `polygon()`).

```
> # Coordonnees du titre
> x <- 135
> y <- -47.5
> # Rajout d'un cadre blanc
> rect(xleft = x - 14, ybottom = y - 2, xright = x + 14, ytop = y +
+     2, col = "white", border = "black")
> # Rajout du titre
> text(x = x, y = y, labels = "CARTE DE L'AUSTRALIE", font = 2,
+     cex = 1.5)
```

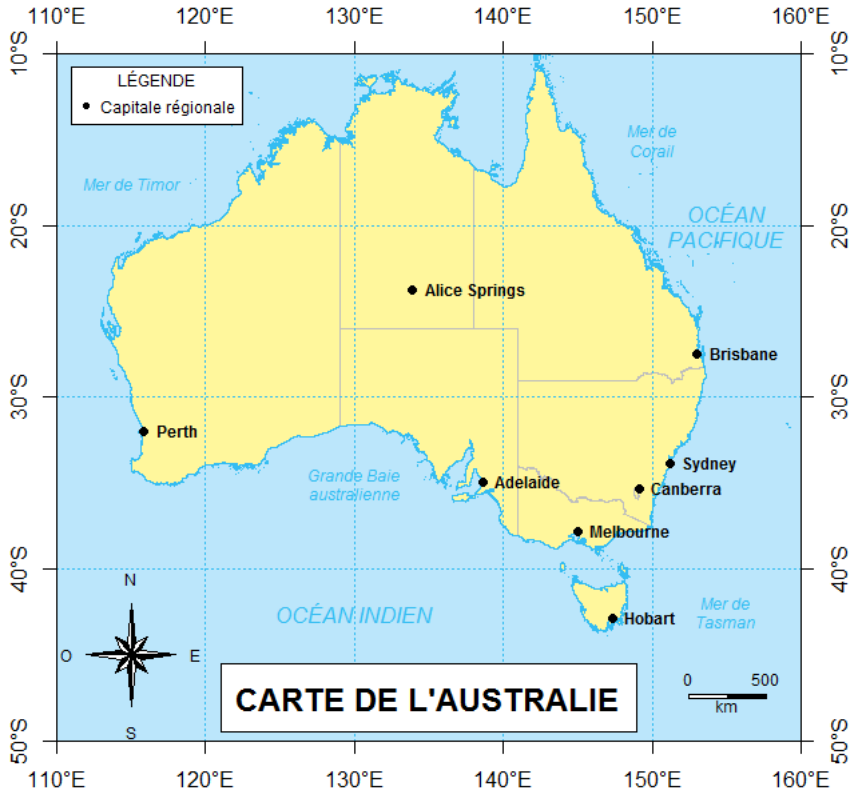

FIGURE 3.9 – Carte avec titre

Pour terminer, regardons comment exporter cette carte au format « tiff ».

```
> # Exportation de la carte
> dest <- "C:/Users/Nicolas/Documents/RFormation"
> savePlot(filename = paste(dest, "/mapAUS", sep = ""), type = "tiff")
```

3.2 Cartographie avec GoogleMap

La carte que nous venons de réaliser est complète et très esthétique, mais le code nécessaire à sa production est assez long à générer. Ainsi, procéder de la sorte pour une simple visualisation de couches géographiques peut s'avérer assez contraignant. Heureusement, d'autres alternatives existent pour générer une carte plus rapidement. De nombreux *packages* ont été développés pour communiquer à la base de données de GoogleMap. Parmi ceux-ci, le *package* `dismo`. Il s'inspire en partie du *package* `RgoogleMaps` et permet de télécharger des données de GoogleMap et de les stocker sous R sous la forme d'une `RasterLayer` (ou d'un `RasterBrick` dans le cas d'un raster tribande). Chargeons ce *package* dans la session de travail.

```
> # Chargement du package 'dismo'
> library(dismo)
```

3.2.1 Accès aux données de GoogleMap

Regardons comment télécharger des données depuis GoogleMap. Pour ce faire, nous allons utiliser la fonction `gmap()` du *package* `dismo`. Celle-ci accepte plusieurs arguments :

- ***x*** correspond au nom de la localité ou aux coordonnées de l'étendue spatiale des données souhaitées ;
- ***exp***, permet de dézoomer la carte sur la zone désirée ;
- ***type***, permet de sélectionner le type de données téléchargées. Il peut prendre les valeurs « roadmap », « satellite », « hybrid » ou « terrain » ;
- ***scale***, permet d'augmenter la résolution de l'image si l'argument prend la valeur 2.

Téléchargeons une carte satellitaire de l'Australie.

```
> # Acces aux donnees satellite de l'Australie
> g1 <- gmap(x = "Australia", type = "satellite")
```

Affichons les caractéristiques de cet objet.

```
> # Classe de l'objet
> class(g1)

## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"

> # Dimensions
> ncol(g1)

## [1] 464

> nrow(g1)

## [1] 640
```

```
> ncell(g1)

## [1] 296960

> # Resolution spatiale
> res(g1)

## [1] 19568 19568

> # Etendue spatiale
> extent(g1)

## class       : Extent
## xmin        : 10610985
## xmax        : 19690481
## ymin        : -10034704
## ymax        : 2488738
```

Ainsi, les données GoogleMap téléchargées se présentent sous un `RasterLayer`, classe d'objet spécifique au *package* `raster`. Ce raster est projeté dans le système Mercator comme en témoignent les unités de l'étendue spatiale.

Affichons la carte satellitaire de l'Australie au moyen de la fonction `plot()`.

```
> # Visualisation du raster
> plot(g1)
```

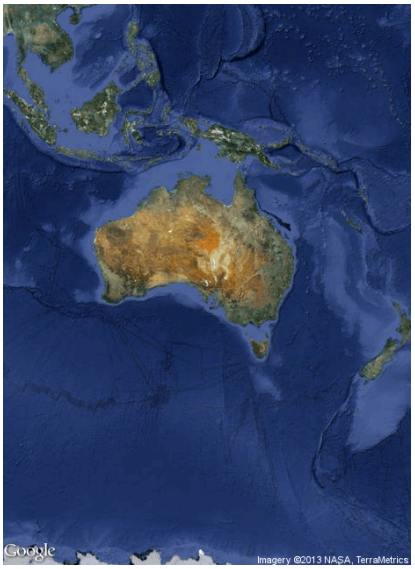

FIGURE 3.10 – Carte satellitaire de l'Australie

Regardons maintenant comment télécharger des données dans le cas où on ignore le nom de la localité. Nous utiliserons la même fonction (`gmap()`), mais nous allons spécifier l'étendue spatiale de la zone au lieu du nom.

```
> # Etendue spatiale
> ext <- extent(c(112, 160, -45, -9))
> # Acces aux donnees GoogleMap
> g2 <- gmap(x = ext, type = "satellite")
> # Etendue spatiale
> extent(g2)

## class       : Extent
## xmin        : 12013482
## xmax        : 18275203
## ymin        : -6053762
## ymax        : -202966
```

Ici, nous avons déterminé l'étendue spatiale de la zone à télécharger dans un système de coordonnées géographiques (longitude et latitude en degrés), mais l'objet retourné est défini dans le système Mercator : la conversion se fait automatiquement. Affichons la carte de la nouvelle zone.

```
> # Visualisation du raster
> plot(g2)
```

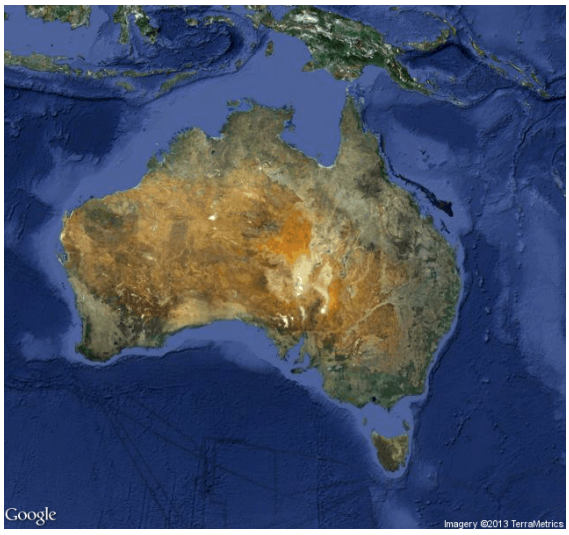

FIGURE 3.11 – Carte satellitaire GoogleMap

3.2.2 Zoom de la carte

Le *package* `raster` met à notre disposition la fonction `drawExtent()` qui permet de sélectionner une zone de zoom en cliquant directement sur la carte. Il suffit de cliquer sur la carte en deux points diagonalement opposés, points qui délimiteront une nouvelle étendue spatiale.

Cette fonction est basée sur la fonction `locator()`, disponible dans les *packages* de base de R. La fonction `locator()` permet de récupérer les coordonnées d'un (ou de plusieurs)

point issu d'un clic sur une fenêtre graphique. La fonction `drawExtent()` procède de même (avec deux points), mais convertit les coordonnées des points sélectionnés par clic en une étendue spatiale. Regardons cela.

```
> # Definition d'une zone de zoom
> e <- drawExtent()
```

Téléchargeons les données correspondant à cette nouvelle région et traçons la carte associée.

```
> # Acces aux donnees
> tas <- gmap(x = e, type = "satellite")
> # Visalisation
> plot(tas)
```

3.2.3 Ajout de couches supplémentaires

La fonction `gmap()` permet d'accéder très rapidement à un fond de carte GoogleMap. Il est tout à fait possible d'y superposer d'autres couches géographiques. Seule restriction : les données supplémentaires doivent être définies dans le système de projection Mercator.

Nous allons rajouter les états/territoires australiens (« aus1 ») sur ce fond de carte, ainsi que les principales villes (« dat »). Projetons tout d'abord leur système de coordonnées (WGS 1984) en Mercator (code EPGS : 3857).

```
> # Projection du systeme de coordonnees
> dat <- spTransform(x = dat, CRSobj = CRS("+init=epsg:3857"))
> aus1 <- spTransform(x = aus1, CRSobj = CRS("+init=epsg:3857"))
```

Rajoutons tout d'abord les états australiens en leur attribuant des couleurs transparentes différentes.

```
> # Carte satellitaire de l'Australie
> plot(g2)
> # Rajout des etats australiens
> for (i in 1:nrow(aus1)) {
+     plot(aus1[i, ], col = rainbow(10, alpha = 0.35)[i], border = NA,
add = T)
+ }
```

Pour terminer, rajoutons les principales villes australiennes.

```
> # Rajout des localisations des villes
> plot(dat, pch = 19, col = "white", add = T)
> # Extraction des coordonnees
> lon <- dat@coords[, "Longitude"]
> lat <- dat@coords[, "Latitude"]
> # Extraction du nom des villes
> labs <- as.character(dat@data[, "Ville"])
> # Rajout du nom des capitales sur la carte
> text(x = lon, y = lat, labels = labs, pos = 4, font = 2, col = "white")
```

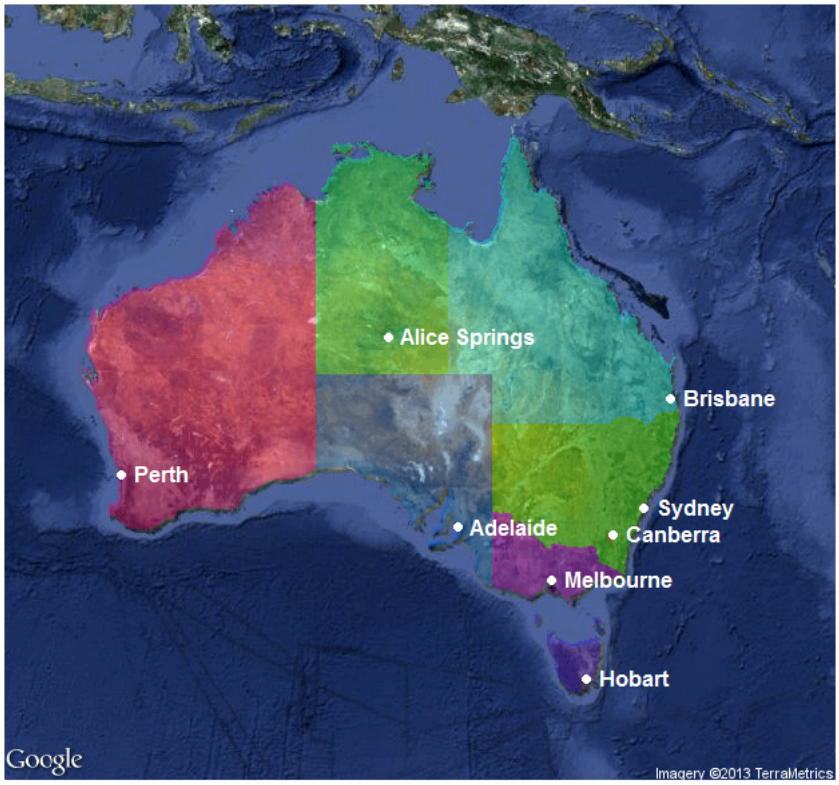

FIGURE 3.12 – Carte des états australiens sur fond satellitaire

3.3 Cartographie d'un objet matriciel

Regardons maintenant comment représenter un objet de type raster. Le *package* `raster` offre quelques outils intéressants (Section 1.5).

Chargeons tout d'abord ce *package* dans la session de travail.

```
> # Chargement du package 'raster'
> library(raster)
```

3.3.1 Carte de base

Commençons par importer le raster donnant l'altitude de l'Australie.

```
> # Importation de l'altitude de l'Australie
> alt <- raster(paste(root, "/Australia/AUS_alt.grd", sep = ""))
```

La fonction `plot()` permet de représenter rapidement un raster. Elle va automatiquement catégoriser les valeurs d'altitude et sélectionner une rampe de couleurs par défaut.

```
> # Visualisation du raster
> plot(alt)
```

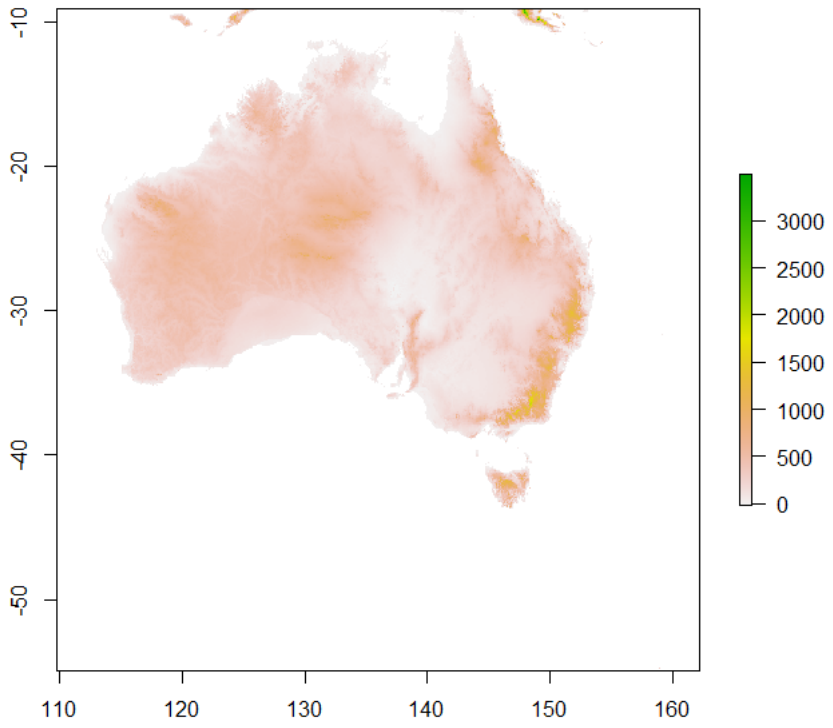

FIGURE 3.13 – Carte altitudinale de l'Australie

La fonction `plot()` utilisée pour représenter un objet matriciel fait implicitement appel aux fonctions `image()` et `image.plot()`, qui offrent de nombreux arguments pour personnaliser la visualisation. On pourra afficher l'aide de ces deux fonctions pour en savoir plus.

3.3.2 Rampe de couleurs

Nous venons de voir que lors de la visualisation cartographique d'un raster, l'algorithme choisissait automatiquement les classes de valeurs (discrétisation de la variable continue) ainsi que la couleur associée à chaque classe. Nous pouvons bien entendu personnaliser la rampe de couleurs.

Plusieurs types de gradients de couleurs sont prédéfinis sous R, mais la fonction `colorRampPalette()` permet de définir ses propres palettes de couleurs. Cette fonction accepte un argument principal qui se présente sous la forme d'un vecteur de couleurs. Cette fonction retournera une fonction qui servira à interpoler un nombre donné de couleurs parmi le gradient formé des couleurs choisies.

Définissons un gradient de couleurs passant par le vert (code hexadécimal : #acd0a5), le jaune (code hexadécimal : #efebc0), le marron (code hexadécimal : #aa8753) et le gris (code hexadécimal : #cac3b8). Les hautes altitudes seront représentées en gris. Nous allons sélectionner 100 couleurs dans cette palette de couleurs.

```
> # Definition d'une rampe de couleurs (fonction)
> rampe <- colorRampPalette(c("#acd0a5", "#efebc0", "#aa8753",
+     "#cac3b8"))
> # Visualisation
> plot(alt, col = rampe(100))
```

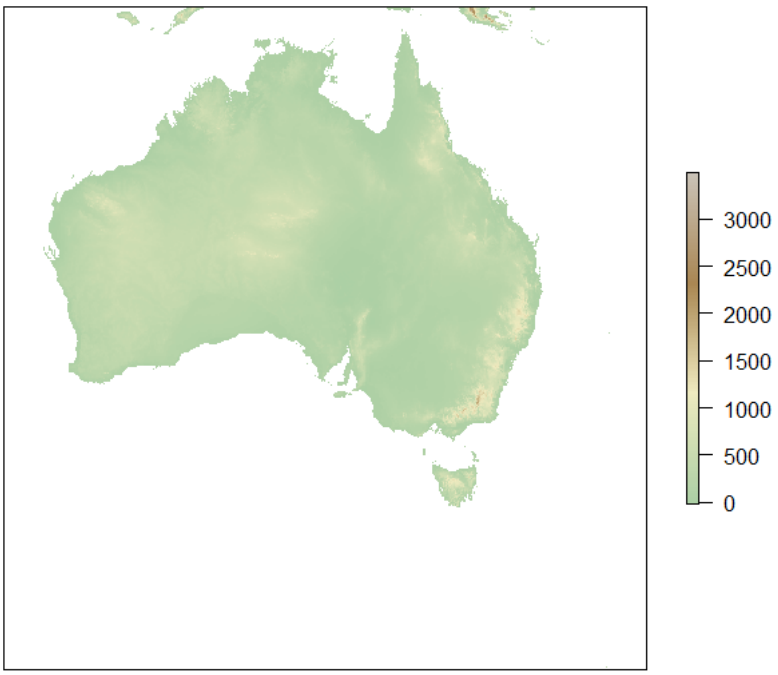

FIGURE 3.14 – Rampe de couleurs personnalisée