COMMENGES Hadrien (dir.), BEAUGUITTE Laurent,
BUARD Élodie, CURA Robin, LE NÉCHET Florent,
LE TEXIER Marion, MATHIAN Hélène, REY Sébastien

R et espace

Traitement de l'information géographique

Publié sous licence

CC By-SA 3.0

Framasoft a été créé en novembre 2001 par Alexis Kauffmann. En janvier 2004 une association éponyme a vu le jour pour soutenir le développement du réseau. Pour plus d'information sur Framasoft, consulter http ://www.framasoft.net.

Se démarquant de l'édition classique, les Framabooks sont dits « livres libres » parce qu'ils sont placés sous une licence qui permet au lecteur de disposer des mêmes libertés qu'un utilisateur de logiciels libres. Les Framabooks s'inscrivent dans cette culture des biens communs qui, à l'instar de Wikipédia, favorise la création, le partage, la diffusion et l'appropriation collective de la connaissance.

Le projet Framabook est coordonné par Christophe Masutti. Pour plus d'information, consultez http ://framabook.org.

Copyright 2014 : Groupe ElementR, Framasoft (coll. Framabook)
R et espace est placé sous licence CC By-SA 3.0

ISBN : 979-10-92674-06-4
Prix : 15 euros
Dépôt légal : Septembre 2014
Pingouins : LL de Mars, Licence Art Libre
Couverture : création initiale par Nadège Dauvergne, Licence CC By
Mise en page avec LaTeX, `knitr` et Inkscape

Introduction

Toutes les disciplines dans lesquelles l'analyse de données occupe une place importante ont connu ces dernières années une petite R-évolution. Certains auteurs ont étudié cette évolution et la façon dont R s'intègre et s'impose dans un marché de logiciels d'analyse de données dominé par trois grands logiciels de statistique : SAS, SPSS et Stata. Ces logiciels conservent des parts de marché importantes mais R gagne en importance depuis le début des années 2000 et cette croissance ne semble pas devoir s'arrêter dans les années qui viennent.

R est un logiciel-langage très particulier qui se caractérise principalement par sa polyvalence. C'est pour cette raison qu'il concurrence, qu'il complète ou qu'il remplace toute une gamme de logiciels et de langages pré-existants. Il n'avance pas seulement sur le terrain des logiciels de statistique classique, il prend également position sur des terrains très spécifiques qui ont leurs logiciels dédiés : la statistique textuelle, l'analyse de graphes, la cartographie et la statistique spatiale en sont des exemples. Véritable langage de programmation, il entre aussi en concurrence avec d'autres langages très utilisés pour le calcul scientifique et l'analyse de données, en particulier avec Python.

Chaque année depuis 2004, les développeurs et utilisateurs de R se retrouvent dans une conférence internationale intitulée *UseR*. Une brève

analyse du contenu de ces conférences[1] montre une extension du champ d'utilisation de R, passant d'un logiciel de chercheurs spécialistes à un logiciel généraliste et pédagogique. R n'est plus seulement un logiciel d'initiés mais un logiciel d'enseignement, à la fois des statistiques et de la programmation, et certains vont jusqu'à annoncer l'avènement de R comme *lingua franca* du traitement de données et de l'analyse statistique (présentation de la conférence UseR 2013).

Ces dernières années ont vu fleurir un grand nombre de manuels, de tutoriels et de collections autour de ce logiciel, chaque domaine ayant son manuel « R pour ... ». Il nous paraissait important de proposer un manuel spécifique intégrant des questionnements et pratiques de géographes. D'abord les manuels sont rares dans ce domaine, surtout en langue française. Il existe bien un manuel complet écrit par Roger Bivand *et al.*, mais ce dernier est en anglais, il est difficile à aborder et son approche est très statisticienne.

Ce manuel adopte une approche plus généraliste de l'analyse de données géographiques et de la cartographie. En français, il n'existe pour le moment que quelques tutoriels et notes de cours sur l'analyse de données géographiques avec R, mais il s'agit soit de brèves introductions, soit d'exemples très spécifiques. Le manuel que nous proposons est bien sûr loin d'être exhaustif, mais il a l'avantage de fournir un contenu conséquent et cohérent présentant l'ensemble des principaux traitements utiles à l'analyse géographique, de la base (découverte de R) à des fonctionnalités plus avancées (cartographie, statistique spatiale).

Ce manuel est le résultat d'un ensemble de séances de formation organisées par le groupe ElementR au laboratoire de recherche Géographie-cités[2] en 2011/2012 pour un public de doctorants, d'enseignants, d'ingénieurs et de chercheurs en géographie. Le public visé est pourtant plus vaste que ce public originel. D'une part parce qu'une partie du manuel est généraliste et comporte des chapitres de prise en main, d'analyses statistiques et de représentations graphiques utiles à toute personne effectuant des études quantitatives. Mais surtout parce que la prise en compte

1. Voir la liste des conférences et les liens correspondants sur `http://www.r-project.org/conferences.html`.

2. Unité Mixte de Recherche associant le CNRS, l'Université Panthéon-Sorbonne et l'Université Paris Diderot.

de l'espace et la cartographie sont de plus en plus présentes dans d'autres disciplines, la sociologie, l'histoire ou les sciences politiques par exemple.

La création et la manipulation de données géographiques se démocratisent depuis quelques années et ne se limitent plus aux étudiants et aux chercheurs. L'usage du GPS se répand pour un usage personnel (itinéraire routier, randonnée) ou pour un usage collectif : projet OpenStreetMap, sites de collecte d'itinéraires (voir par exemple le site de la Fédération Française de Cyclisme), etc. Les données publiques nouvellement accessibles grâce au mouvement d'ouverture des données (*open data*) sont de plus en plus utilisées pour produire des cartes de thèmes d'intérêt, comme les résultats des élections présidentielles par exemple.

L'approche du manuel est celle de l'analyse spatiale, à savoir des méthodes mises en œuvre pour l'étude de l'organisation des phénomènes dans l'espace. La mise en œuvre de ces méthodes nécessite le plus souvent des mises en forme informatiques des données en amont, et des capacités pour récupérer, interpréter et représenter les informations en sortie.

L'ensemble de cette chaîne nécessitait jusqu'il y a quelques années l'utilisation de plusieurs logiciels, la plupart d'entre eux étant des logiciels propriétaires et particulièrement coûteux : SAS pour l'analyse de données, ArcGIS pour la cartographie et la statistique spatiale, et des logiciels complémentaires pour l'analyse de graphes par exemple. L'avantage de R est qu'il permet de faire la majeure partie de ces opérations dans un même flux de travail (*workflow*, c'est-à-dire la chaîne des traitements effectués). Le fait qu'il s'agisse d'un logiciel libre auquel les utilisateurs peuvent également contribuer fait que son champ s'étend de façon considérable : au début des années 2000, il y avait quelques 30 *packages* (bibliothèques de fonctions) assez généralistes ; au début de l'année 2014, il y en avait plus de 5 000.

L'intérêt du manuel est de proposer un ensemble comprenant les explications, les programmes et les données. La plupart des applications sont faites sur le même jeu de données caractérisant le même espace d'étude : Paris et la petite couronne (départements 75, 92, 93, 94). Ponctuellement, certains jeux de données d'exemple contenus dans le logiciel R sont mobilisés.

Ce manuel est divisé en trois parties indépendantes contenant chacune plusieurs chapitres. Il y a des renvois fréquents d'un chapitre à un autre,

mais chaque chapitre est autonome : en début de chapitre, les noms des fichiers de données nécessaires ainsi que les *packages* de R nécessaires au déroulement du programme sont précisés. La liste complète des *packages* utilisés figure en annexe du manuel. Une brève bibliographie est également proposée qui signale des références spécifiques permettant d'approfondir le volet technique (programmation) et/ou le volet théorique (méthodes d'analyse).

La première partie **Manipulation des données et programmation** comporte des éléments de langage nécessaires pour débuter avec R et manipuler les données : le chapitre 1 est une très brève prise de contact avec le logiciel et le fil du manuel, le chapitre 2 présente des éléments de prise en main et illustre différentes méthodes pour manipuler les données. Le chapitre 3 introduit des éléments plus avancés de programmation avec la mise en œuvre de boucles et de fonctions.

La deuxième partie présente des méthodes statistique d'**Exploration des données géographiques**. Ainsi les chapitre 4 et 5 abordent les méthodes de traitements statistiques univariés et bivariés classiquement utilisés en analyse spatiale. Le chapitre 6 décline des questions nécessitant l'utilisation de méthodes factorielles multivariées. Enfin le chapitre 7 présente les méthodes de classification.

La dernière partie, **Éléments spécifiques de traitement de l'espace**, regroupe quatre chapitres illustrant des aspects plus spécialisés en géographie. Le chapitre 8 revient sur des fondamentaux de l'analyse de réseaux. Le chapitre 9 fait le point sur les dispositifs de visualisation, préalable nécessaire à la présentation des techniques de cartographie dans le chapitre 10. Enfin, le chapitre 11 présente des éléments de statistiques spatiales et, en particulier, une initiation à l'autocorrélation spatiale.

CHAPITRE 1

Prise de contact

1.1 R dans une coquille de noix

R est un langage créé en 1993 par Robert Gentleman et Ross Ihaka, de l'Université d'Aukland. Il s'agit d'une nouvelle implémentation d'un langage plus ancien, le langage S créé à la fin des années 1970 dans les laboratoires Bell.

R est un logiciel libre, gratuit et multiplateforme (Linux, Windows, Mac). En pleine expansion, il concurrence avec succès les logiciels commerciaux qui détenaient ce marché : SAS, SPSS et Stata. Robert Muenchen propose sur son site Internet[1] des analyses quantifiées de cette concurrence. Il est aussi l'auteur de manuels facilitant la migration des utilisateurs des grands logiciels commerciaux vers le logiciel R.

R est composé d'un socle commun (`r-base`) sur lequel se greffe un ensemble de *packages*. Un *package* est une bibliothèque de fonctions implémentées par les utilisateurs et mises à disposition de tous par l'intermédiaire de dépôts regroupés dans le cadre du *Comprehensive R Archive*

1. `http://r4stats.com`.

Network (CRAN). Cette structure modulaire, commune à de nombreux logiciels libres, explique la vaste étendue des applications possibles : l'expansion du logiciel n'est limité que par le travail que les utilisateurs du monde entier mettent à disposition de l'ensemble des autres utilisateurs.

La structure modulaire du logiciel R peut être vue comme un arbre de dépendances : un *package* dépend de fonctions implémentées dans d'autres *packages*, qui eux-mêmes dépendent de fonctions implémentées dans d'autres *packages*, etc. Cet arbre est une structure hiérarchique dans le sens où les *packages* spécialisés ont tendance à dépendre de *packages* plus généralistes.

L'un des aspects les plus déroutants de ce logiciel, pour les débutants et pour les utilisateurs de logiciels de statistique classiques, est qu'il existe toujours de multiples façons d'effectuer une tâche. C'est la conséquence de la structure modulaire qui vient d'être décrite : d'abord, des fonctions identiques ou semblables sont implémentées indépendamment dans plusieurs *packages* ; ensuite, le développement très rapide du logiciel et de ses modules mène souvent à l'amélioration de fonctions pré-existantes. Dans ce dernier cas, il y a superposition entre l'ancienne façon de faire, qui se maintient, et la nouvelle façon de faire qui attire de plus en plus d'utilisateurs, surtout si elle est meilleure à tous points de vue (cf. Section 2.5).

Le champ extensible de R fait qu'il est possible de manipuler tous types d'objets. Ceci permet d'intégrer dans un même flux de travail des analyses de données statistiques, spatiales et temporelles, de produire des tableaux, des graphiques et des cartes. Ce flux de travail intégré est plus efficace et plus sûr, car il supprime les continuelles importations et exportations pour passer d'un logiciel à l'autre (d'un logiciel de statistique à un logiciel de cartographie par exemple).

1.2 Installation

Le logiciel se compose d'une simple console sous Linux ou bien d'un ensemble de trois fenêtres sous Windows et MacOS. La console regroupe trois fonctions : elle affiche les résultats, elle permet d'écrire du code et

elle renvoie les messages d'erreur. Deux types d'interfaces permettent de compléter la console :

— les interfaces graphiques (GUI - *Graphical User Interface*) comme R Commander, qui proposent un ensemble de menus et de boutons et permettent de travailler sans connaître la syntaxe ;
— les environnements de développement (IDE - *Integrated Development Environment*), dont le plus répandu est RStudio que nous utilisons dans ce manuel. RStudio intègre dans une même interface la console, l'éditeur de script, le contenu de l'espace de travail, l'historique, l'aide, les graphiques et l'accès aux *packages*.

Pour commencer à travailler sur ce manuel, il faut installer le logiciel R [1] puis le logiciel RStudio [2].

En ce qui concerne l'utilisation des *packages*, il faut bien distinguer deux choses : installer un *package* et le charger. On peut installer des centaines de *packages* sur une machine, mais au lancement R ne les charge pas tous. Il faut donc les charger selon les besoins, ce qui se fait soit en cochant le *package* voulu dans l'onglet correspondant de RStudio soit en utilisant la fonction `library()`.

1.3 Utilisation de RStudio

RStudio est un environnement de développement dédié à R, il est pratique, complet et en rapide évolution. L'interface graphique de RStudio se compose de quatre fenêtres (cf. Figure 1.1) : l'éditeur de code, la console, l'espace de travail dans lequel s'affichent les objets créés avec une information sur leur nature et leur contenu, et enfin une fenêtre qui permet de gérer les *packages* (installation, mise à jour, chargement) et d'accéder à l'aide.

Il y a principalement deux façons d'écrire et d'exécuter des lignes de code : dans la console et dans l'éditeur de code. La console est utile pour écrire des commandes courtes qui ne sont pas destinées à être conservées, par exemple installer ou charger un *package*. L'éditeur, en revanche, est

1. `http://cran.r-project.org`.
2. `http://rstudio.org`.

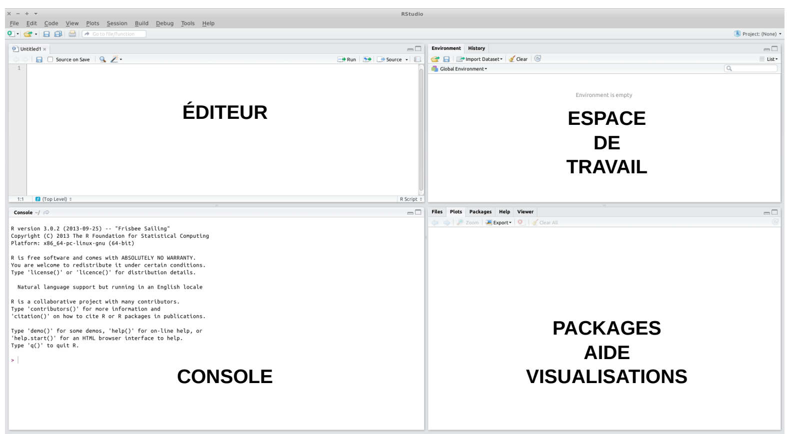


FIGURE 1.1 – Structure de l'interface graphique de RStudio

utilisé pour écrire du code pérenne que l'utilisateur souhaite conserver ou partager.

Pour ouvrir un nouveau fichier de code, on utilise le menu **Fichier > Nouveau > Script R**. Il y a plusieurs options disponibles à la création d'un fichier (C++, Markdown, HTML, etc.) parce qu'il est possible d'utiliser d'autres langages et de les combiner avec du code R. Ce manuel en est un exemple : le corps de texte est écrit avec LaTeXet le code avec R.

RStudio propose de nombreuses fonctionnalités intéressantes, à commencer par la gestion de projets. L'onglet **Projects**, situé en haut à droite de l'interface, permet de créer des projets et d'ouvrir des projets existants. Ceci facilite l'accès à des espaces de travail différenciés correspondant aux différents travaux en cours de l'utilisateur.

Le bouton **Importer** situé dans la fenêtre **Environment** est utile pour importer des tableaux en format texte (txt ou csv). Il présente toutes les options nécessaires pour importer correctement ses données : intitulé de colonne, séparateur de colonnes, séparateur décimal.

La fenêtre située en bas à droite permet en premier lieu de gérer l'installation et le chargement des *packages*, avec les boutons de l'onglet intitulé **Packages**. Dans cet onglet apparaissent tous les *packages* installés sur la machine. S'y trouvent également plusieurs boutons pour installer les

packages, les mettre à jour ou les charger en cochant les cases correspondantes.

Dans cette même fenêtre, l'onglet **Help** permet d'accéder à la documentation de toutes les fonctions des *packages* installés et chargés. L'onglet **Plots** propose des options de visualisation et d'exportation des graphiques en plusieurs formats courants, par exemple svg, png ou pdf.

L'ensemble des boutons de l'interface RStudio exécute des commandes visibles dans la console, commandes qui peuvent bien sûr être écrites (dans la console ou dans l'éditeur de code) et exécutées directement. Par exemple, le bouton d'installation de packages exécute la fonction `install.packages()`, le bouton d'importation de données exécute la fonction `read.table()`.

RStudio présente aussi la capacité de compléter automatiquement les termes en cours d'écriture (capacité qualifiée par la suite d'autocomplétion). L'autocomplétion fonctionne avec la touche `Tab` du clavier, qui complète automatiquement les noms des objets chargés dans l'espace de travail ou le nom des fonctions et arguments à utiliser. On l'utilise dans plusieurs cas :

— saisir les premières lettres d'un objet créé par l'utilisateur (ou d'une fonction) et appuyer sur la touche `Tab` pour finir d'écrire le nom de l'objet (ou de la fonction) ;
— saisir le nom d'un *data.frame* ou d'une liste (*list*), le $ permettant de charger les éléments stockés dans ces objets, puis appuyer sur `Tab`. L'interface propose alors l'ensemble des variables dans une liste déroulante ;
— saisir le nom d'une fonction, ouvrir la parenthèse et appuyer sur `Tab`. L'interface propose alors la liste des arguments que la fonction prend en entrée ainsi que la description correspondant à chaque argument.

Enfin, RStudio propose de nombreux raccourcis clavier intéressants : le raccourci `Alt` + `-` renvoie l'opérateur d'assignation accompagné d'un espace avant et un espace après (`<-`). Le raccourci `Ctrl` + `Entrée` exécute le code écrit dans la fenêtre d'édition (script) : si une partie du code est sélectionnée, c'est cette partie qui est exécutée ; si le curseur est placé sur une ligne donnée, c'est seulement cette ligne qui est exécutée. Enfin,

les raccourcis `Ctrl` + `1` et `Ctrl` + `2` permettent de passer de la console à l'éditeur de code et *vice versa*.

1.4 Conventions d'écriture

La lisibilité d'un programme est un impératif majeur à respecter : lisibilité pour le programmeur lui-même mais aussi pour les autres utilisateurs qui seraient amenés à l'utiliser. Dans ce cadre il est crucial de documenter son programme, c'est-à-dire d'en commenter les principales étapes. Un programme non commenté devient rapidement incompréhensible, y compris pour la personne qui en est l'auteur. Pour cela, on utilise le croisillon (`#`) : tout ce qui vient après ce symbole n'est pas considéré comme des commandes à exécuter.

Même si les conventions d'écriture ne semblent pas cruciales, surtout pour un utilisateur débutant ou isolé, il faut savoir que tous les langages de programmation ont des conventions qui facilitent l'échange et la coopération. Il existe plusieurs guides de style ou recueils de bonnes pratiques à ce sujet. Ils diffèrent essentiellement sur la façon de nommer les objets : doit-on écrire `MonObjet`, `monObjet`, `mon.objet` ? Pour le reste tous les guides de style sont d'accord sur les notations suivantes :

- Espaces : toujours placer un espace avant et après un opérateur (+, −, =, etc.). Toujours placer un espace après une virgule, mais pas avant.
- Mise en forme du code : l'écriture de commandes longues, souvent des fonctions ou des ensembles de conditions devraient aussi respecter certaines conventions pour faciliter leur lecture, en particulier en termes de saut de ligne et d'alinéa (*indentation*).

Dans ce manuel, deux conventions d'écriture sont utilisés : les fonctions sont notées avec des séparations en majuscules (MonObjet, convention dite *camel case*) et les variables sont notées avec une minuscule sur la première lettre (monObjet, convention dite *lower camel case*).

Une autre question se pose dans l'écriture du code, celle de la langue utilisée pour désigner les objets. Comme le logiciel, ses *packages* et ses fonctions sont en anglais, les objets créés seront également notés en anglais et `myObject` sera préféré à `monObjet`.

1.5 Versions et mises à jour

R est un logiciel à structure modulaire dont la base (`r-base`) et les différents *packages* sont mis à jour au rythme du travail des contributeurs, c'est-à-dire à un rythme variable. L'ensemble composé de la base (`r-base`) et d'un certain choix de *packages* est donc un ensemble mouvant, dont la dynamique est propre à chaque utilisateur et à ses pratiques de mise à jour. Il arrive que la version la plus récente d'un *package* ne puisse être installée que sur la version la plus récente de `r-base`.

D'une façon générale, au vu de la dynamique qui porte ce logiciel, il est conseillable de rester le plus à la page possible et de mettre à jour les composants du logiciel régulièrement. Le journal des modifications, de la base et des *packages*, peut être suivi sur le site du CRAN cité plus haut. Le présent manuel a été rédigé et testé sur une base logicielle mise à jour en juillet 2014 qui comprend : la version 3.1 de R, la version 0.98 de RStudio et les dernières mises à jour de tous les *packages* utilisés.

1.6 L'exemple et les données

Ce manuel présente des manipulations sur plusieurs jeux de données qui sont téléchargeables sur le site Internet de Framabook [1]. Presque toutes les applications sont faites sur le même jeu de données caractérisant le même espace d'étude : Paris et la petite couronne, espace décrit au niveau communal (143 communes pour 4 départements 75, 92, 93, 94). On propose trois jeux de données statistiques mis à disposition par l'Insee :

— données socioéconomiques des recensements de 1999 et de 2007,
— série temporelle des populations communales depuis 1936,
— données de mobilité résidentielle (changements de domicile) en 2008.

Certaines données cartographiques sont également utilisées, données vectorielles et *raster* en accès libre sur le site de l'IGN correspondant au même espace d'étude : Paris et la petite couronne.

1. `http://framabook.org`.

Vecteur et *raster* : la distinction entre format vecteur et format *raster* est essentielle pour le traitement informatique d'images. Le format *raster*, ou *bitmap*, représente l'image par une matrice de pixels dont les valeurs se traduisent par des couleurs. Les photographies en sont l'exemple le plus courant. Le format vecteur représente des objets géométriques par leurs caractéristiques (points, segments, polygones) et leurs coordonnées.

À côté de ce jeu de données principal, dans certains exercices, des jeux de données d'exemple contenus dans R sont utilisés. Parmi ces jeux de données, ceux contenus dans le *package* `HistData` sont particulièrement intéressants. Il s'agit de données issues de l'histoire de la statistique et de la visualisation de données, comme ceux de Galton sur la régression, ceux de Minard sur la campagne de Russie de Napoléon ou ceux de Snow sur le choléra.

1.6.1 Description du fichier SocEco9907.csv

Ensemble de variables socio-économiques renseignées au recensement de la population de 1999 et de 2007. Le nom de ces variables indique leur contenu et la date du recensement, par exemple **PCAD99** et **PCAD07** pour la proportion de cadres en 1999 et en 2007. Pour les variables indiquant une proportion nous signalons entre parenthèses la population de référence, qui peut être soit la population totale, soit la population de plus de 15 ans, soit la population active, soit la population active occupée.

CODGEO Code communal

LIBGEO Nom de la commune (arrondissement pour Paris)

X, Y Coordonnées géographiques en Lambert 2 étendu (hectomètres)

SURF Superficie en hectares

EMPLOI Nombre d'emplois

ACTOCC Nombre d'actifs occupés

P20ANS Proportion de moins de 20 ans (population totale)

PNDIP Proportion de non diplômés (population > 15 ans)

TXCHOM Proportion de chômeurs (population active)

INTAO Proportion d'intérimaires (population active occupée)

PART Proportion d'artisans (population active occupée)

PCAD Proportion de cadres (population active occupée)

PINT Proportion de professions intermédiaires (population active occupée)

PEMP Proportion d'employés (population active occupée)

POUV Proportion d'ouvriers (population active occupée)

PRET Proportion de retraités (population active occupée)

PMONO Proportion des familles monoparentales (familles)

PREFETR Proportion des ménages dont la personne de référence est étrangère (ménages)

RFUCQ Revenu médian (euro courant)

REFEROUI Pourcentage de votes pour le OUI au référendum européen (DATE)

1.6.2 Description du fichier PopCom3608.csv

Série temporelle des populations communales depuis 1936 produites par les éditions successives du Recencement de la Population (RP).

CODGEO Code communal

LIBGEO Nom de la commune (arrondissement pour Paris)

SURF Superficie en hectares

POP1936 Population sans double compte au RP 1936

POP1954 Population sans double compte au RP 1954

POP1962 Population sans double compte au RP 1962

POP1968 Population sans double compte au RP 1968

POP1975 Population sans double compte au RP 1975

POP1982 Population sans double compte au RP 1982

POP1990 Population sans double compte au RP 1990

POP1999 Population sans double compte au RP 1999

POP2008 Population sans double compte au RP 2008

1.6.3 Description du fichier MobResid08.txt

Données de mobilité résidentielle en 2008. Ce fichier retrace les changements de domicile : il donne le nombre d'individus de 5 ans et plus ayant changé de commune de résidence entre la date du recensement (2008) et 5 ans auparavant, soit 2003 dans ce fichier. La variable **CODGEO** doit donc être lue comme la commune de destination et **DCRAN** comme la commune d'origine.

CODGEO Code de la commune de résidence

LIBGEO Nom de la commune de résidence (arrondissement pour Paris)

DCRAN Code de la commune de résidence antérieure

L_DCRAN Nom de la commune de résidence antérieure (arrondissement pour Paris)

NBFLUX Nombre d'individus ayant changé de commune de résidence

1.6.4 Description des données cartographiques

Trois types de données cartographiques sont utilisées : des données zonales (vectorielles) constituées des limites des 143 communes étudiées, des données ponctuelles (vectorielles) qui renseignent sur la localisation des hôpitaux et leur capacité, et des données *raster* qui renseignent sur la densité du bâti.

Le fichier des limites communales ne contient que les codes permettant de faire des jointures entre les données de l'Insee et le fond cartographique. Le fichier des hôpitaux contient plusieurs champs renseignant leur capacité exprimée en nombres de lits. Enfin, le fichier *raster* est une matrice de pixels dont la valeur représente la densité du bâti.

CHAPITRE 2

Prise en main et manipulation des données

Objectifs : *Ce chapitre permet d'acquérir quelques bases dans la manipulation de données sous R. Il permet d'abord de se familiariser avec les principaux objets (vecteurs, matrices, tables) et les opérations les plus courantes (opérations, assignations, sélections, tests conditionnels, recodages).*

Prérequis Notions de base du fonctionnement de R et de l'interface RStudio, telles que présentées dans le Chapitre 1.

Description des *packages* utilisés Les manipulations effectuées dans ce chapitre nécessite l'utilisation de deux *packages* :

- *Package* `sqldf` : il permet d'exécuter des requêtes SQL (*Structured Query Language*) sur les tableaux. Ce *package* est constitué d'une unique fonction éponyme, qui prend pour argument la requête SQL souhaitée.
- *Package* `reshape2` : il propose plusieurs fonctions utiles pour manipuler et reformater les tableaux de données.
- *Package* `plyr` : il sert principalement à diviser un objet en blocs, appliquer une fonction à chaque bloc et recoller les résultats obtenus pour chaque bloc (*split-apply-combine strategy*). Il propose également quelques fonctions simples pour recoder et trier.
- *Package* `dplyr` : ré-écriture récente de `plyr`, ce *package* très prometteur simplifie la plupart des manipulations présentées dans ce chapitre et en améliore la vitesse d'exécution.

2.1 Description et manipulation des objets

R est un langage orienté objet : tout ce qui est créé et manipulé sous R est un objet. Ces objets permettent de stocker et de structurer les données.

2.1.1 Les principaux types d'objets

- Vecteur (*vector*) : suite unidimensionnelle et ordonnée de valeurs. Les trois principaux types de vecteurs sont : *numeric* (entier ou double précision), *character* (alphanumérique), *boolean* (`TRUE`/`FALSE`).
- Facteur (*factor*) : vecteur qui ne peut prendre qu'un nombre fini de modalités prédéclarées. Ce vecteur est défini par des niveaux (`levels`), qui sont les valeurs effectivement prises par la variable, et d'étiquettes (`labels`), qui sont associées aux niveaux. Par exemple, une variable définissant le sexe des individus pourrait être codée avec les entiers 1 et 2 (`levels`) auxquels correspondraient les étiquettes (`labels`) *Homme* et *Femme*. Plusieurs fonctions de création de tableaux ou d'importation de données transforment automatiquement les champs alphanumériques en facteurs.

— Matrice (*array* ou *matrix*) : un objet *array* est une matrice, c'est-à-dire un vecteur multidimensionnel. Un objet *matrix* est un sous-type de l'objet *array*, c'est une matrice bi-dimensionnelle. Les données stockées dans une matrice sont nécessairement d'un seul et même type : booléen, entier, alphanumérique, etc. Avec RStudio, dans la fenêtre **Environment** on peut cliquer sur les objets de type matrice et les afficher.
— Liste (*list*) : liste ordonnée d'objets. Une liste permet de regrouper un ensemble d'objets de tailles différentes et de natures diverses : vecteurs, matrices ou tout type d'objet existant dans R. Par exemple, on peut stocker l'ensemble des résultats d'un modèle de régression dans une seule et même liste qui contiendrait un vecteur numérique de longueur n de valeurs espérées, un vecteur numérique de longueur n de résidus, un vecteur numérique de longueur 1 stockant une *p-value*, un vecteur booléen sur la significativité du résultat, etc.
— Tableau (*data.frame*) : tableau de données. Quand on importe un fichier depuis un tableur ou depuis un format SAS ou SPSS, l'objet créé est par défaut un *data.frame*. Pour les utilisateurs de tableurs et de logiciels d'analyse comme SAS ou SPSS, le *data.frame* sera le type d'objet le plus familier. Cet objet tableau combine des caractéristiques de liste et de matrice : c'est une liste de vecteurs assortis d'un nom (intitulé de colonne). À la différence d'une matrice, les vecteurs contenus dans le tableau peuvent être de différentes natures : valeurs numériques, alphanumériques, booléennes. À la différence d'une liste, les vecteurs contenus dans le tableau doivent être de même taille. Avec RStudio, dans la fenêtre **Environment** on peut cliquer sur les tableaux et les afficher.
— Autres objets : il existe une grande diversité de types d'objets du fait que certaines fonctions créent des objets sur mesure. La fonction `lm()` (*linear model*) créé par exemple un objet de type *lm* qui est une liste d'objets (*fitted.values*, *p.value*, etc.). Les données spatiales sont aussi stockées dans des objets particuliers qui sont détaillés au chapitre correspondant (cf. Chapitre 10).
— Fonction : deux caractéristiques importantes des fonctions doivent être soulignées, d'une part les fonctions sont des objets au même titre que les vecteurs ou les matrices, d'autre part les fonctions

peuvent prendre d'autres fonctions comme argument. Ces aspects sont détaillés dans le Chapitre 3.

L'existence de différents types d'objets peut être déroutante mais elle devient une richesse lorsqu'on sait s'en servir. Le pendant de cette diversité est qu'il faut régulièrement se soucier du format des objets et les transformer d'un type à un autre.

2.1.2 Le vecteur et le facteur

Avant de présenter les principales méthodes de création et de manipulation des objets, un bref focus s'impose sur deux types d'objets fondamentaux : le vecteur et le facteur. Un vecteur est une suite ordonnée de valeurs de même type. La fonction `c()` (*combine*) crée un vecteur en combinant des valeurs. Par exemple, la syntaxe `c(2, 4, 6)` renvoie un vecteur constitué de ces trois valeurs : ce vecteur a une taille (3), ses valeurs peuvent être utilisées dans d'autres calculs, ses valeurs ont un ordre par lequel elles peuvent être désignées (la 2e valeur est 4).

La plupart des objets utilisés dans R peuvent être décomposés en un ensemble de vecteurs. Le tableau suivant (cf. Figure 2.1.2) est constitué de 5 lignes (individus) et 3 colonnes (variables) : il peut être considéré comme la combinaison de trois vecteurs composés de cinq valeurs ou comme la combinaison de cinq vecteurs composés de trois valeurs. Dans le premier cas, le vecteur stocke une variable ; dans le second cas, le vecteur stocke un individu.

Lorsqu'on crée un tableau, qu'on le remplit ou qu'on en extrait des informations, il est utile de garder à l'esprit qu'il est décomposable en un ensemble de vecteurs.

Le facteur (*factor*) est un vecteur un peu particulier qu'il est nécessaire de savoir manipuler. Il arrive souvent de transformer des facteurs en vecteurs et des vecteurs en facteurs. Il est aussi fréquent d'empêcher la création de facteurs à la création ou l'importation de tableaux. En effet, dans l'importation d'un tableau externe, les champs alphanumériques sont automatiquement transformés en facteurs, sauf à spécifier le contraire avec l'argument `stringsAsFactors = FALSE`.

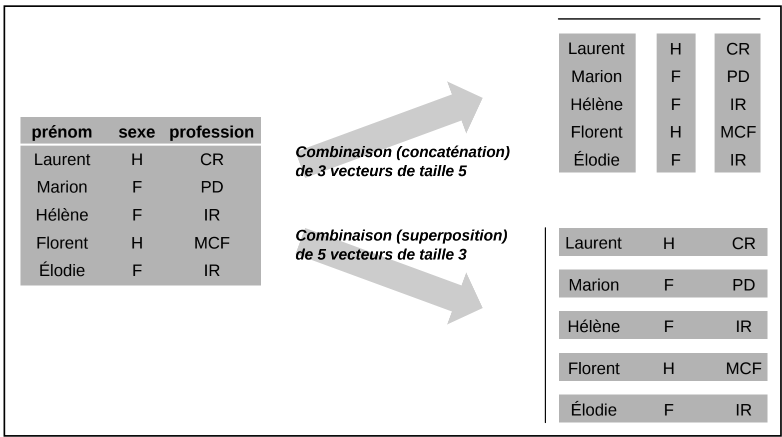


FIGURE 2.1 – Décomposition d'un tableau en vecteurs

Le facteur est très pratique pour stocker de véritables variables qualitatives, c'est-à-dire des variables dont les modalités caractérisent des sous-populations : pour des individus, le sexe ou la catégorie socioprofessionnelle ; pour des communes, le département d'appartenance ou la couleur politique. Ce format permet d'éviter des erreurs de codage et il est particulièrement intéressant pour les catégories aux intitulés très longs. Par exemple, pour travailler sur des catégories socioprofessionnelles, il est plus pertinent de coder ces catégories avec des entiers auxquels sont associées des étiquettes plutôt que de coder directement les longs intitulés de ces catégories (« Cadres et professions intellectuelles supérieures », etc.). Le format facteur fait aussi gagner de la mémoire dans un tel cas puisque ces longues étiquettes ne sont pas répétées autant de fois qu'il y a d'individus.

En revanche, le facteur n'est pas adapté lorsqu'il s'agit de stocker des identifiants : le code d'identification des individus ou le code Insee des communes par exemple. Par définition, un identifiant est un champ qui prend une valeur unique pour chaque individu et permet ainsi de l'identifier. Le format facteur ne fournit dans ce cas aucun gain de mémoire puisqu'il y a autant d'étiquettes distinctes que d'individus. Plus important, le facteur est constitué de niveaux et d'étiquettes et cette combinaison est

particulièrement dangereuse lors des manipulations de tableaux qui impliquent leur identifiant. Toute jointure de tableaux sur des identifiants stockés en format facteur est à proscrire.

2.1.3 Déclarer des objets avec l'opérateur d'assignation

Pour créer et initialiser un objet, on utilise l'opérateur d'assignation `<-`. Sans l'opérateur d'assignation le résultat de l'instruction est renvoyé dans la console mais aucun objet n'est créé. La syntaxe générale est la suivante "`Mon objet <- Contenu de mon objet`".

```
1 + 1 # renvoie le résultat dans la console

## [1] 2

myObject <- 1 + 1 # assigne le résultat à un objet
myObject # affiche le contenu de l'objet

## [1] 2
```

Dans RStudio, chaque objet créé est affiché dans la fenêtre **Environment** accompagné de certaines de ses caractéristiques. Dans cette fenêtre, on peut cliquer sur les objets créés pour les visualiser.

Il y a deux méthodes pour déclarer des objets :

- de façon implicite, c'est-à-dire en assignant à l'objet créé le résultat d'une fonction qui renvoie par défaut un certain type d'objet ;
- de façon explicite, c'est-à-dire en déclarant le type de l'objet au moment de sa création.

```
# déclaration implicite
myObject <- 1 + 1
class(myObject)

## [1] "numeric"

# déclaration explicite
myFactor <- factor(c(1,1,2,1,2,2),
                   labels = c("Homme", "Femme"))
class(myFactor)

## [1] "factor"
```

On peut créer les différents types d'objets cités précédemment avec des fonctions éponymes : `vector()`, `matrix()`, `list()`, `data.frame()`, etc. Dans les trois exemples qui suivent, on utilise la fonction `c()` pour créer un vecteur simple (Créer un vecteur) ; pour créer un vecteur et le mettre en forme dans une matrice à 3 lignes et 4 colonnes (Créer une matrice) ; pour créer un vecteur de deux noms (ID et CLASS), qu'on assigne comme noms de colonnes à un *data.frame* (Créer un *data.frame*).

Créer un vecteur :

La fonction `c()` renvoie un vecteur en combinant des valeurs d'un certain type, par exemple de valeurs numériques (*numeric*) ou des valeurs alphanumériques (*character*).

```
vecNum <- c(1, 2, 3, 4, 5)
vecNum

## [1] 1 2 3 4 5

vecChar <- c("CP", "CE1", "CE2", "CM1", "CM2")
vecChar

## [1] "CP"  "CE1" "CE2" "CM1" "CM2"
```

Créer une matrice :

La création d'une matrice se fait à partir d'un vecteur en explicitant le nombre de lignes et de colonnes à construire. Par défaut R remplit d'abord les colonnes de haut en bas puis de gauche à droite comme l'illustre l'exemple ci-dessous.

```
matNum <- matrix(c(1:12), nrow = 3, ncol = 4)
matNum


##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

Créer un *data.frame* :

Pour créer un objet *data.frame* on a recours à la fonction éponyme qui peut prendre en entrée une matrice, ou une série de vecteurs. Le dernier argument `stringAsFactors = FALSE` empêche la transformation automatique des vecteurs de chaînes de caractères en facteurs. Pour modifier ou afficher les noms de colonnes, sous la forme d'un vecteur alphanumérique, on peut utiliser la fonction `names()` ou `colnames()`. Les noms de colonnes peuvent être modifiés *a posteriori* (Option 1) ou dès la déclaration de l'objet (Option 2).

```
# Option 1
dfNumChar <- data.frame(vecNum,
                        vecChar,
                        stringsAsFactors=FALSE)
colnames(dfNumChar)

## [1] "vecNum"  "vecChar"

colnames(dfNumChar) <- c("ID", "CLASS")

# Option 2
dfNumChar <- data.frame(ID = vecNum,
                        CLASS = vecChar,
                        stringsAsFactors = FALSE)
dfNumChar

##   ID CLASS
## 1  1    CP
## 2  2   CE1
## 3  3   CE2
## 4  4   CM1
## 5  5   CM2
```

2.1.4 Modifier le type d'un objet

Il est possible de transformer un objet d'un certain type en un autre type, avec des fonctions suivant la syntaxe suivante `as.` suivi du type choisi, par exemple `as.matrix`, `as.vector`, etc. Ces transformations peuvent être classés en trois catégories : les transformations sans perte d'information, les transformations avec perte d'information, les transformations détruisant la structure de l'objet.

```
dfNum <- data.frame(col1 = vecNum,
                    col2 = c(2,5,8,7,8),
                    stringsAsFactors = FALSE)

# transformation sans perte
matNum <- as.matrix(dfNum)

# transformation avec perte des étiquettes
myVec <- as.numeric(myFactor)
myVec

## [1] 1 1 2 1 2 2

# perte d'une dimension, perte des noms des champs
vecFromMat <- as.vector(matNum)
vecFromMat

##  [1] 1 2 3 4 5 2 5 8 7 8
```

2.1.5 Désigner des lignes, des colonnes ou des valeurs

L'un des gros avantages de R vis-à-vis des logiciels de statistiques classiques est qu'il permet de désigner très facilement un ensemble de valeurs contenues dans un objet, comme dans un tableur où on peut désigner une cellule en précisant sa ligne et sa colonne. La façon de désigner ces éléments diffère selon le type d'objet dans lequel ils sont stockés.

Pour les vecteurs et les matrices, on fait appel à l'index souhaité entre crochets (`[]`) et on désigne la dimension de l'objet souhaitée en les séparant par des virgules : une dimension pour un vecteur, deux pour une matrice, trois pour un cube, etc. Par exemple, on appelle la deuxième valeur d'un vecteur en écrivant `monVecteur[2]`.

Dans le cas des tableaux (*data.frame*) et des listes, les crochets s'utilisent de la même façon que dans le cas précédent, mais il est aussi possible de désigner l'élément souhaité précédé d'un dollar (`$`). Les utilisateurs habitués aux tableaux avec des intitulés de colonne accèderont de cette façon à la variable de leur choix : `MonTableau$VARIABLE`. Pour créer une nouvelle variable, il suffit d'assigner des valeurs à une variable

dont le nom n'existe pas encore dans le tableau. Si l'utilisateur assigne des valeurs à une variable dont le nom existe déjà dans le tableau, les valeurs originelles seront écrasées par les nouvelles valeurs.

En laissant l'indexation vide pour une dimension donnée, on la récupère dans son ensemble. Par exemple, pour un tableau à deux dimensions (*data.frame* ou *matrix*), l'instruction `monTableau[2, 3]` renverra la valeur située à la deuxième ligne et troisième colonne alors que l'instruction `monTableau[2, ]` renverra la deuxième ligne et l'ensemble des colonnes. Testez les lignes suivantes :

```
# Sélection
vecNum[2]
matNum[3, 2]
matNum[, 2]
dfNumChar[, 2]
dfNumChar$CLASS
dfNumChar$CLASS[2]
dfNumChar[3, 2]

# Sélection et assignation
vecCharLong <- vecChar
vecCharLong[1] <- "Cours primaire"
vecCharLong[2:3] <- "Cours élémentaire"
vecCharLong[4:5] <- "Cours moyen"

dfNumChar$CLASS <- "CP"
```

Les utilisateurs ayant une culture de tableur ou de logiciel de statistique classique utilisent fréquemment des tableaux de variables. Ils trouveront pesant de sans cesse devoir faire référence au tableau accompagné du $ avant de désigner la variable (`monTableau$VARIABLE`). Il est possible d'attacher le tableau et de désigner directement les variables par leur nom avec la fonction `attach()` (et `detach()` pour le détacher). Cependant, ces fonctions ne permettent pas de travailler directement sur l'objet *data.frame* mais sur une copie de celui-ci. Pour cette raison elles représentent une source d'erreur et leur usage est fortement déconseillé. La fonction `with()` peut être utilisée dans le même but et elle est sans danger.

```
mean(dfNum$col2)

## [1] 6

with(dfNum, mean(col2))

## [1] 6
```

2.1.6 Traitement des valeurs manquantes

Dans R, les valeurs manquantes sont notées `NA` (*not available*). Par défaut, la plupart des fonctions de base (somme, moyenne, minimum, maximum, etc.) n'acceptent pas les variables contenant des valeurs manquantes, ce qui peut être déroutant pour les utilisateurs d'autres logiciels, en particulier les tableurs. Dans ce cas, l'utilisateur doit préciser l'action à réaliser avec les valeurs manquantes, la plus simple étant souvent de les supprimer (*remove*) : `na.rm=TRUE`.

```
vecTest <- c(2, 4, NA, 6)
mean(vecTest)

## [1] NA

mean(vecTest, na.rm = TRUE)

## [1] 4
```

La fonction `is.na()` est utile pour se renseigner sur la présence de valeurs manquantes, elle renvoie un vecteur booléen indiquant la position de ces valeurs manquantes. Cet index des valeurs manquantes peut être réutilisé de plusieurs façons : combiné avec la fonction `any()`, il permet de savoir s'il y a une ou plusieurs valeurs manquantes dans une variable ; intégré avec un assignement il peut servir à recoder ces valeurs manquantes.

```
# Index des valeurs manquantes
is.na(vecTest)

## [1] FALSE FALSE  TRUE FALSE

# Présence de valeurs manquantes dans la variable
any(is.na(vecTest))

## [1] TRUE

# Option équivalente à la ligne précédente
anyNA(vecTest)

## [1] TRUE

# Recodage des valeurs manquantes en 0
vecTest[is.na(vecTest)] <- 0
```

2.1.7 Se renseigner sur les objets et leur contenu

Il est fondamental de toujours savoir quels objets ont étés créés durant la session, sur quels types d'objets on travaille, quelle est leur structure, leur taille ou leur contenu. Les commandes suivantes sont donc indispensables pour éviter de travailler à l'aveugle.

Lister et effacer les objets déclarés dans la session :

```
ls() # lister les objets
rm(vecNum) # effacer un seul objet
rm(list = ls()) # effacer tous les objets
```

Se renseigner sur la nature et le contenu des objets :

```
data(cars)
str(cars) # structure de l'objet

## 'data.frame': 50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...

class(cars) # type de l'objet

## [1] "data.frame"

dim(cars) # dimension de l'objet

## [1] 50  2

is.matrix(cars) # test sur le type de l'objet

## [1] FALSE

is.data.frame(cars) # test sur le type de l'objet

## [1] TRUE
```

Dans l'exemple ci-dessus, le jeu de données utilisé (`cars`) est un jeu de données contenu dans la base R, il est chargé avec la fonction `data()`. Pour obtenir la liste de tous les jeux de données d'exemple disponibles, on peut écrire la fonction `data()` puis utiliser la touche `Tab` pour l'autocomplétion.

2.2 Importation et exportation

Cette section permet d'introduire les commandes nécessaires à l'importation et l'exportation de données externes dans les formats les plus couramment utilisés.

2.2.1 Le répertoire de travail

Avant d'importer et d'exporter des données, il est nécessaire de savoir se renseigner sur le répertoire de travail (*working directory*) et éventuellement le modifier. Deux fonctions simples permettent de gérer ce répertoire :

— `getwd()` : renvoie le chemin du *working directory* ;
— `setwd("/chemin/dossier")` : spécifie le *working directory*.

Certaines fonctionnalités introduites par RStudio rendent plus facile la gestion de ce répertoire. Dans le menu **Session**, plusieurs boutons permettent de fixer rapidement le répertoire de travail. En outre, RStudio permet d'organiser le travail en projets grâce à l'onglet **Projects**, situé en haut à droite de l'interface. Utiliser ces fonctionnalités facilite l'accès à des espaces de travail différenciés correspondant aux différents travaux en cours de l'utilisateur.

2.2.2 Les formats de données

Les objets peuvent être créés directement sous R ou importés depuis des bases de données externes. Plusieurs formats sont disponibles :

— formats texte (csv, txt) (importation directe avec `r-base`),
— formats tableurs (xls, ods) (l'importation sous R nécessite des *packages* spécifiques),
— formats spécifiques des logiciels de statistiques (sas7bdat pour SAS, sav pour SPSS, etc.),
— formats spécifiques de bases de données (SQLite, MySQL, etc.).

2.2.3 Codage des caractères et des séparateurs

Il faut bien distinguer le format d'un fichier, indiqué par son extension (xls, csv, etc.), du codage (*encoding*) de ses caractères. Il est toujours utile de savoir quel est le codage des documents qu'on utilise, en général UTF8 ou Latin1. Il arrive souvent qu'on importe un fichier avec des champs de caractères en français et que les accents et les apostrophes soient mal affichés. Ces problèmes surviennent en général dans le cas d'une utilisation

de plusieurs systèmes d'exploitation (Linux-Windows par exemple), ou quand les données sont manipulées sur des ordinateurs ayant des normes linguistiques différentes.

Un autre aspect des conventions d'écriture doit être souligné, celui du codage des séparateurs de champs et des séparateurs décimaux. Un fichier de données au format texte (txt ou csv) suivant les conventions anglo-saxonnes aura pour séparateur de champs la virgule et pour séparateur décimal le point. Un fichier aux conventions françaises aura pour séparateur de champs le point-virgule et pour séparateur décimal la virgule. Ces informations seront nécessaires lors de l'importation du fichier.

2.2.4 Importer des données en format texte

Les données stockées dans des formats de tableurs (xls, ods) peuvent être importées directement, mais on évite bien des soucis en les enregistrant au préalable sous un format texte (txt ou csv). Il y a ensuite deux façons d'importer dans RStudio un fichier de ce type :

- par l'interface graphique : dans RStudio, dans l'onglet **Environment** de la fenêtre placée en haut à droite de l'écran, on utilise le bouton **Import Dataset**. On peut visualiser les données brutes et les données telles qu'elles apparaitront après importation. Cela permet de modifier si nécessaire le séparateur de champs, le séparateur décimal et d'inclure ou non l'intitulé de colonne ;
- en ligne de commande : on utilise la fonction `read.table()` en spécifiant le chemin et le nom du fichier, le type de séparateur de champs (`sep`), le type de séparateur décimal (`dec`), le fait d'inclure ou non les intitulés de colonnes (`header`). Le codage du fichier d'origine peut aussi être précisé : si le codage du système est UTF-8 et que le fichier importé affiche mal les accents et les apostrophes, l'argument de codage (`encoding`) peut être utilisé (*latin1* en général).

2.2.5 Exporter des données en format texte

Pour exporter un tableau de données en format texte on utilise la fonction `write.table()` en spécifiant, comme pour la fonction

`read.table()`, le nom et le chemin du fichier à créer, les séparateurs, etc. Ces fonctions peuvent être remplacées par des fonctions qui présentent certains réglages par défaut, en particulier des séparateurs, `read.csv()` par exemple.

2.2.6 Importation et exportation depuis/vers d'autres formats

Le *package* `foreign` permet de lire et d'écrire des fichiers dans les formats des grands logiciels commerciaux. Il comprend les fonctions `read.spss()` pour les formats sav, `read.xport()` pour le format xport de SAS. Pour SAS il existe aussi le *package* `sas7bdat` qui permet d'importer directement des fichiers dans ce format.

Il est également possible de stocker des données dans une base de données externes, PostgreSQL, SQLite ou autre et d'y accéder depuis R avec les *packages* correspondants (`RPostgreSQL`, `RSQLite`, etc.).

2.3 Recoder et trier

Les sélections et les recodages se font principalement avec les opérateurs logiques suivants :

- x est égal à y : `x == y`
- x n'est pas égal à y : `x != y`
- x est strictement supérieur à y : `x > y`
- x est strictement inférieur à y : `x < y`
- x est supérieur ou égal à y : `x >= y`
- x est inférieur ou égal à y : `x <= y`
- $A \cap B$ (opérateur `ET`, intersection de deux conditions) : `A & B`
- $A \cup B$ (opérateur `OU`, union de deux conditions) : `A | B`

2.3.1 Sélectionner et recoder

Le réflexe des utilisateurs de logiciels de statistiques classiques est de vouloir faire des sélections et des recodages avec des opérateurs conditionnels (`if [...] then`). Par exemple on a une liste d'individus dont l'âge est renseigné, et on souhaite discrétiser cette variable en deux classes

d'âge selon que les individus sont mineurs ou majeurs. Une pseudo-syntaxe classique donnerait quelque chose comme :

```
tant que (ligne individu != 0)

   {if AGE < 18 then CLASS = Min else CLASS = Maj}
```

À partir des méthodes présentées jusqu'ici, le premier réflexe serait d'utiliser l'indexation des valeurs : pour chaque numéro de ligne où la variable **AGE** est inférieure à 18, écrire « Min » dans la variable **CLASS** ; pour chaque numéro de ligne où la variable **AGE** est supérieure ou égale à 18, écrire « Maj » dans la variable **CLASS**.

```
dfIndiv <- data.frame(ID = c(1, 2, 3, 4, 5),
                      AGE = c(12, 17, 24, 45, 8),
                      SEX = c("H", "H", "F", "F", "F"))

# Assignation conditionnelle dans une nouvelle variable
dfIndiv$CLASS[dfIndiv$AGE < 18] <- "Min"
dfIndiv$CLASS[dfIndiv$AGE >= 18] <- "Maj"
dfIndiv$CLASS

## [1] "Min" "Min" "Maj" "Maj" "Min"
```

La fonction `ifelse()` simplifie ce travail. Elle sert à assigner une valeur à une variable en fonction d'un test conditionnel et peut être appliquée sur un tableau. Cette fonction s'utilise de la façon suivante :

```
ifelse(test, valeur si vrai, valeur si faux)
```

```
dfIndiv$CLASS <- ifelse(dfIndiv$AGE < 18, "Min", "Maj")
dfIndiv$CLASS

## [1] "Min" "Min" "Maj" "Maj" "Min"
```

Attention, il ne faut pas confondre la fonction `ifelse()` avec la fonction `if()` qui sert à exécuter une instruction en fonction du résultat du test (cf. Section 3.2).

La fonction `subset()` est très utile pour sélectionner des parties d'un tableau en fonction de la valeur de certaines variables ou pour conserver certaines colonnes d'un tableau. Cette fonction, comme la fonction

`ifelse()`, peut prendre comme argument un test à conditions multiples utilisant les opérateurs de comparaison.

```
subset(dfIndiv, dfIndiv$SEX == "F" & dfIndiv$AGE > 18)

##   ID AGE SEX CLASS
## 3  3  24   F   Maj
## 4  4  45   F   Maj
```

Le *package* `plyr` (puis `dplyr`, cf. Section 2.5) propose deux fonctions utiles pour le recodage : `mapvalues()` et `revalue()` qui permettent d'assigner de nouvelles valeurs à un vecteur numérique, alphanumérique ou à un facteur.

```
library(plyr)
revalue(dfIndiv$SEX, c("F" = "Femme", "H" = "Homme"))

## [1] Homme Homme Femme Femme Femme
## Levels: Femme Homme

mapvalues(dfIndiv$SEX,
          from = c("F", "H"),
          to = c("Femme", "Homme"))

## [1] Homme Homme Femme Femme Femme
## Levels: Femme Homme
```

Une opération de recodage très fréquente est la discrétisation, qui consiste à découper une variable continue en classes. Il serait bien sûr possible de faire ce recodage avec une suite de tests conditionnels définis dans un emboîtements de fonctions `ifelse()`. Cependant, R dispose d'une fonction plus pratique pour discrétiser une variable continue : la fonction `cut()`.

```
valBreaks <- c(8, 18, 45)
dfIndiv$AGEDISCRET <- cut(dfIndiv$AGE,
                          breaks = valBreaks,
                          include.lowest = TRUE,
                          right = FALSE)
dfIndiv

##   ID AGE SEX CLASS AGEDISCRET
## 1  1  12   H   Min     [8,18)
## 2  2  17   H   Min     [8,18)
## 3  3  24   F   Maj    [18,45]
## 4  4  45   F   Maj    [18,45]
## 5  5   8   F   Min     [8,18)
```

Cette fonction prend un vecteur numérique comme argument et le découpe en fonction de seuils (`breaks`) définis au préalable. Les intervalles peuvent être fermés sur la droite ou sur la gauche (`right`) et inclure la valeur maximum ou minimum selon le cas (`include.lowest`). Le vecteurs de seuils est ici défini à la main, mais il est bien sûr possible d'utiliser des fonctions pour définir des seuils fréquemment utilisés, comme les quantiles avec la fonction `quantile()`. Cette manipulation est présentée dans la Section 3.3.2.

2.3.2 Trier

Pour trier les valeurs d'un objet dans un certain ordre, croissant, décroissant ou alphabétique, deux fonctions peuvent être utilisées : `sort()` et `order()`. Il y a deux différences majeures entre ces fonctions : `sort()` ne permet de trier les valeurs que d'une seule variable, alors que `order()` permet de trier un tableau de valeurs en fonction d'une ou plusieurs variables. Autre différence notable, `order()` renvoie le rang tenu par les valeurs, alors que `sort()` renvoie les valeurs elles-mêmes.

Ainsi pour trier l'ensemble d'un tableau selon les valeurs prises par une variable, il faudra toujours utiliser `order()`. L'intérêt de la fonction `order()` réside dans la possibilité de trier un tableau de plusieurs variables selon l'ordre des valeurs prises par une variable en particulier. Si `data` est le nom d'un tableau et `data$var` le nom de la

variable selon laquelle le tableau doit être trié, on utilise l'instruction `data[order(data$var), ]`. Cette syntaxe indique qu'il faut ordonner les lignes du tableau (première dimension entre crochets) en fonction du rang des valeurs de la variable `var`.

```
# Trier un vecteur
sortedAge <- sort(dfIndiv$AGE)
sortedAge

## [1]  8 12 17 24 45

# Trier un tableau selon une variable
sortedIndiv <- dfIndiv[order(dfIndiv$AGE), ]

# Trier un tableau selon plusieurs variables
sortedCars <- cars[order(cars$speed, cars$dist), ]
```

La syntaxe de la fonction `order()` est peu lisible. Le *package* `plyr` (puis `dplyr`, cf. Section 2.5) propose une fonction plus pratique pour trier un tableau de données : `arrange()`.

```
library(plyr)
sortedCars <- arrange(cars, speed, dist)
```

Si trier les valeurs d'un vecteur numérique ne pose pas de problème particulier, il est plus délicat de trier un facteur. En effet, celui-ci est composé de valeurs (`levels`) et d'étiquettes (`labels`), le tri peut s'appliquer à l'un ou l'autre selon la syntaxe employée.

```
sexFactor <- factor(c(1,1,2,1,2,2), labels = c("H", "F"))
as.numeric(sexFactor)

## [1] 1 1 2 1 2 2

as.character(sexFactor)

## [1] "H" "H" "F" "H" "F" "F"
```

Si le tri s'applique aux valeurs, les hommes (codés 1) seront placés avant les femmes (codées 2). Si le tri s'applique aux étiquettes, les femmes

(lettre f) seront placées avant les hommes (lettre h). Il est donc plus sûr de transformer le facteur avec les fonctions `as.character()` ou `as.numeric()` puis d'appliquer explicitement le tri aux étiquettes ou aux valeurs.

2.4 Manipulation avancée

La première partie de ce chapitre a présenté des procédures simples de calcul, de sélection, de recodage et de tri. La section suivante introduit les principales techniques de manipulation avancée des données, à savoir les techniques qui permettent de transformer la structure même des données par des agrégations, des jointures et des transpositions.

Les données utilisées ici concernent les communes de Paris et de la petite couronne, c'est-à-dire les départements 75, 92, 93 et 94. L'importation se fait en utilisant les fonctions présentées plus haut (cf. Section 2.2) : `read.table()`, `read.csv()` ou interface graphique de RStudio.

```
socEco9907 <- read.csv("data/SocEco9907.csv",
                       sep = ";",
                       stringsAsFactors = FALSE)

popCom3608 <- read.csv("data/PopCom3608.csv",
                       sep = ";",
                       stringsAsFactors = FALSE)
```

2.4.1 Superposition, concaténation, jointure

De la même façon que la fonction `c()` combine des valeurs dans un vecteur ou une liste, les fonctions `rbind()` et `cbind()` permettent de prendre un vecteur, une matrice ou un tableau et de les combiner par ligne ou par colonne. La combinaison par ligne (*row*) est appelée superposition (`rbind`) et la combinaison par colonne (`column`) est appelée concaténation (`cbind`).

Ces deux fonctions doivent être utilisées avec précaution. La fonction `rbind()` permet de superposer deux tableaux dont les colonnes sont identiques, c'est-à-dire qui contiennent le même nombre de colonnes et

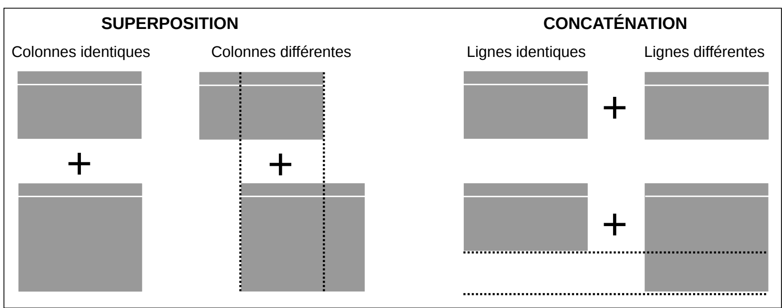


FIGURE 2.2 – Superposition et concaténation

dont les noms sont les mêmes. Prenant en compte le nom de colonne, la fonction permet une superposition même si les colonnes ne sont pas dans le même ordre. Si les colonnes sont différentes, la fonction renvoie un message d'erreur.

De la même façon, si les deux tableaux ont un nombre de lignes différent, la fonction `cbind()` renvoie un message d'erreur. En revanche, si le nombre de lignes est le même, la fonction concatène les deux tableaux sans se soucier de leur contenu. Elle est donc inadaptée pour combiner des données de différentes sources sur les mêmes individus statistiques (individus, communes, régions, etc.). Il faut dans ce cas faire une concaténation selon un identifiant qui garantit que l'identité de chaque individu est conservée. Cette concaténation avec identifiant est appelée jointure.

C'est la fonction `merge()` qui permet d'effectuer cette jointure de deux tableaux en fonction d'une variable commune. L'exemple suivant joint les deux tableaux **socEco9907** et **popCom3608** à l'aide de l'identifiant commun nommé **CODGEO** dans les deux tableaux.

```
jointCom <- merge(socEco9907, popCom3608, by = "CODGEO")
```

Le cas présenté ici est le plus simple : les deux tableaux ont exactement les mêmes individus statistiques (*i.e.* les lignes se correspondent deux à deux) et le nom de la variable identifiant (code de la commune) est le même dans les deux tableaux.

La fonction `merge()` s'adapte à tous les cas de jointures grâce à des arguments complémentaires. L'argument `by` peut être divisé en `by.x` et `by.y` pour désigner le nom de la colonne qui sert d'identifiant dans chacun des deux tableaux. Les arguments `all`, `all.x`, `all.y` servent dans les cas où les deux tableaux n'ont pas les mêmes individus statistiques. Quatre grands cas de jointures peuvent être distingués :

— *inner join* : conservation des seules lignes communes aux deux tableaux (`all=FALSE`) ;
— *left join* : conservation de toutes les lignes du tableau de gauche même si elles n'ont pas de lignes correspondantes dans le tableau de droite (`all.x=TRUE`) ;
— *right join* : conservation de toutes les lignes du tableau de droite même si elles n'ont pas de lignes correspondantes dans le tableau de gauche (`all.y=TRUE`) ;
— *outer join* : conservation de toutes les lignes des deux tableaux (`all=TRUE`).

2.4.2 Agrégations et traitements par blocs

Les procédures d'agrégation servent en premier lieu à calculer un résumé numérique selon une variable d'agrégation. Cette procédure met en jeu trois arguments : la variable sur laquelle est calculé le résumé numérique (variable quantitative), la variable d'agrégation (variable qualitative) et la fonction d'agrégation (somme, moyenne, max, etc.). Trois fonctions sont utiles dans ce cas : `by()`, `tapply()` et `aggregate()`.

L'exemple suivant consiste à calculer la population totale par département à partir de la variable population renseignée à la commune. Il demande d'abord de créer une variable indiquant le département par extraction des deux premiers caractères du code de la commune. Cette variable est créée avec la fonction `substr()` (*substring*) qui prend comme arguments la variable alphanumérique (le code Insee) et la position du premier et du dernier caractère à extraire. La fonction `tapply()` est employée car c'est la plus simple (et la plus limitée) des trois : elle n'accepte qu'une seule variable de calcul et qu'une seule variable d'agrégation.

```
popCom3608$DEP <- substr(popCom3608$CODGEO, 1, 2)
tapply(popCom3608$POP2008, popCom3608$DEP, sum)

##      75      92      93      94
## 2211297 1549619 1506466 1310876

tapply(popCom3608$POP1936, popCom3608$DEP, sum)

##      75      92      93      94
## 2829755 1019627  776378  685204
```

Il est souvent utile de faire ce type de traitement sur plusieurs variables quantitatives et/ou plusieurs variables d'agrégation. C'est dans ce cas que les fonctions `by()` et `aggregate()` sont employées. Cette procédure peut servir à renvoyer un simple résultat à analyser mais également à recréer une information à un niveau spatial plus agrégé. À partir des données communales, il serait par exemple intéressant de créer des tableaux au niveau départemental regroupant les mêmes variables de population et d'emploi.

```
popDep <- aggregate(popCom3608[ , c("POP1936", "POP2008")],
          by = list(popCom3608$DEP),
          FUN = sum)
popDep$EVOL3608 <- with(popDep, POP2008 / POP1936 - 1)
popDep$EVOL3608

## [1] -0.2186  0.5198  0.9404  0.9131
```

On constate ici le contraste entre la décroissance de la population au centre de l'agglomération parisienne (-0,22 % pour le département 75) et la croissance de la population des départements de petite couronne (92, 93, 94) entre 1936 et 2008.

Au-delà du simple calcul de résumés numériques selon une ou plusieurs variables d'agrégation, il est parfois nécessaire d'appliquer des fonctions plus compliquées à un tableau de données divisé en blocs. Pour ce type de traitement (*split-apply-combine strategy*), il faudra utiliser les fonctions du *package* `plyr`. Elles fonctionnent comme celles qui viennent d'être présentées avec les trois mêmes arguments principaux (données, indices,

fonction). Les deux premières lettres de leur nom indique le type des objets en entrée et en sortie de la fonction, par exemple `ddply()` prend en entrée un tableau (*data.frame*) et renvoie un tableau. Plusieurs options sont possibles, par exemple `dlply()` (tableau, liste) ou `mdply()` (matrice, tableau).

Le traitement par blocs peut être combiné avec tout type de fonctions, que celles-ci soient propres à l'utilisateur ou implémentées dans les *packages* existants. Parmi ces dernières, trois sont spécialement utiles : `transform()`, `summarise()` et `subset()`. Ce premier exemple calcule la moyenne et l'écart-type de la population communale de 2008 par département.

```
depAgg08 <- ddply(.data = popCom3608,
                  .variables = "DEP",
                  .fun = summarise,
                  POPMEAN08 = mean(POP2008),
                  POPSD08 = sd(POP2008))

str(depAgg08)

## 'data.frame': 4 obs. of  3 variables:
##  $ DEP      : chr  "75" "92" "93" "94"
##  $ POPMEAN08: num  110565 43045 37662 27891
##  $ POPSD08  : num  71227 26127 24250 21623
```

Ce second exemple standardise les populations communales en 2008 pour chacun des quatre départements considérés. Ces variables centrées-réduites (moyenne = 0, écart-type = 1) sont calculées avec la fonction `scale()` qui est appliquée à chaque sous-ensemble correspondant à chacun des quatre départements.

```
standPop08 <- ddply(.data = popCom3608,
                    .variables = "DEP",
                    .fun = transform,
                    STDPOP08 = scale(POP2008))
```

Le traitement par blocs est particulièrement utile pour analyser des données géographiques puisqu'il est fréquent de faire des traitements à plusieurs niveaux emboîtés : communes, départements, régions par exemple.

2.4.3 Transposition variables-observations

Il est courant d'avoir à transformer la structure même des données en réorganisant les lignes et les colonnes. Cette procédure, nommée ici transposition variables-observations, se fait avec les fonctions du *package* `reshape2`. Elle ne doit pas être confondue avec la transposition du calcul matriciel qui consiste à inverser les lignes et les colonnes et qui se fait avec la fonction `t()`.

Lorsqu'on travaille sur des tableaux statistiques, une ligne ne représente pas toujours un individu statistique et une colonne une variable. Dans certains cas, ce type de formalisation n'est pas pertinent ou n'existe pas. Par exemple dans une matrice origine-destination (origine des flux en ligne, destination des flux en colonne) la distinction entre individu et variable n'a pas de sens.

De façon générale, on distingue deux types de format de données : d'une part, le format large (*wide*), qui est celui utilisé jusqu'à présent dans le manuel et dans lequel une ligne représente un individu statistique caractérisés par plusieurs variables en colonne. D'autre part, le format long (*long*), qui est très utilisé par exemple dans les bases de données temporelles. Pour manipuler des données longitudinales (la même variable observée à plusieurs pas de temps), il y a toujours deux options de stockage :

- sous forme *wide*, une ligne représente un individu statistique et chaque colonne la mesure pour une date donnée ;
- sous forme *long*, il y a trois colonnes : l'identifiant de l'individu, la date de la mesure et la valeur de la mesure. Ainsi chaque ligne représente une combinaison unique individu-date.

Les deux formats représentent bien sûr exactement la même information et c'est dans la facilité d'usage selon différents objectifs que réside le choix du format. Ce sont les fonctions `melt()` et `cast()` du *package* `reshape2` qui permettent le passage de l'un à l'autre.

Ce *package* part d'une distinction essentielle des variables d'un tableau entre variables identifiants et des variables de mesure. Les variables identifiants (*id variables*) sont discrètes, elles désignent les unités d'enregistrement. Les variables de mesures (*measured variables*) enregistrent les mesures correspondant à chaque unité d'enregistrement. Il est conseillé, mais

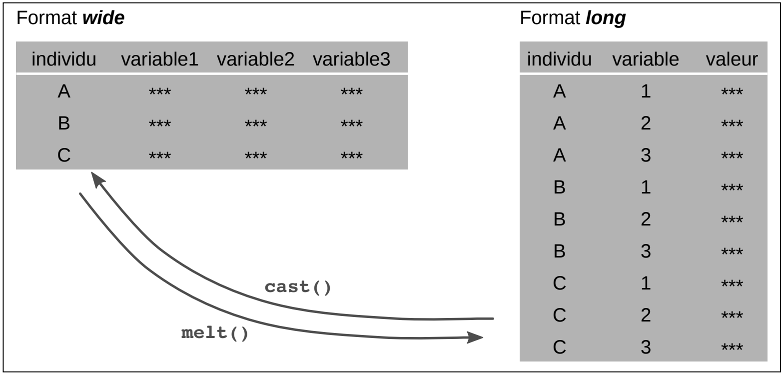


FIGURE 2.3 – Transposition variables-observations

non nécessaire, de commencer la transposition par la fonction `melt()`, pour bien identifier les variables identifiants et les variables de mesure. Cette fonction permet de préparer les données en vue d'une utilisation avec d'autres commandes offertes par le même *package*.

```
library(reshape2)
meltedPopCom <- melt(popCom3608,
                     id.vars = c("CODGEO","LIBELLE","SURF"),
                     variable.name = "YEAR")

head(meltedPopCom)


##   CODGEO                  LIBELLE SURF    YEAR value
## 1  75101 Paris 1er Arrondissement  183 POP1936 37062
## 2  75102  Paris 2e Arrondissement   99 POP1936 41445
## 3  75103  Paris 3e Arrondissement  117 POP1936 63571
## 4  75104  Paris 4e Arrondissement  160 POP1936 62547
## 5  75105  Paris 5e Arrondissement  254 POP1936 97396
## 6  75106  Paris 6e Arrondissement  215 POP1936 81403
```

Dans l'exemple précédent, l'argument `id.vars` sert à désigner les variables identifiants, l'argument `variable.name` sert à nommer la variable qui va stocker les mesures et l'argument `measure.vars` n'est pas précisé. N'étant pas précisé, la fonction `melt()` comprend que toutes les variables non désignées comme identifiant sont des variables de mesure.

La fonction `cast()` se décline suivant le type d'objet renvoyé : `acast()` pour une matrice (*array*) et `dcast()` pour un tableau (*data.frame*). La formule utilisée par la fonction `cast()` sépare les variables à mettre en colonne des variables à mettre en ligne avec le tilde (symbole ˜ qui s'écrit avec `AltGr + 2` sur un clavier français). Elle accepte plusieurs variables à mettre en colonne et/ou plusieurs variables à mettre en ligne, en ce cas la formule utilise l'opérateur + :

```
column_var ~ row_var
```

```
cvar_1 + cvar_2 + cvar_n ~ rvar_1 + rvar_2 + rvar_n
```

Voici un exemple de transposition variables-observations permettant de revenir au tableau initial *wide*, avant son re-positionnement au format *long* par la fonction `melt()`.

```
castedByCode <- dcast(meltedPopCom, CODGEO ~ YEAR)
head(castedByCode[, 0:6])

##   CODGEO POP1936 POP1954 POP1962 POP1968 POP1975
## 1  75101   37062   37330   36543   32332   22793
## 2  75102   41445   41744   40864   35357   26328
## 3  75103   63571   64030   62680   56252   41706
## 4  75104   62547   62998   61670   54029   40466
## 5  75105   97396   98099   96031   83721   67668
## 6  75106   81403   81991   80262   70891   56331
```

La transposition permet aussi de récupérer, à partir du format *long*, le code géographique en colonne et le temps en ligne :

```
castedByYear <- dcast(meltedPopCom, YEAR ~ CODGEO)
head(castedByYear[, 0:6])

##      YEAR 75101 75102 75103 75104 75105
## 1 POP1936 37062 41445 63571 62547 97396
## 2 POP1954 37330 41744 64030 62998 98099
## 3 POP1962 36543 40864 62680 61670 96031
## 4 POP1968 32332 35357 56252 54029 83721
## 5 POP1975 22793 26328 41706 40466 67668
## 6 POP1982 18509 21203 36094 33990 62173
```

Le passage d'un format à l'autre est fréquent dès qu'il s'agit de traiter des flux entre des lieux. Une information de ce type comprend un couple

de lieux (origine-destination) et des mesures correspondantes à ce couple : une distance ou un flux par exemple. L'exemple suivant s'applique sur les données de migrations résidentielles (changements de domicile) produites par l'Insee, données contenues dans le fichier **MobResid08.txt**. Ce fichier est chargé avec la fonction `read.csv2()` qui lit un fichier texte selon les conventions françaises (cf. Section 2.2.3) : le point-virgule comme séparateur de colonnes et la virgule comme séparateur décimal.

```
residFlows <- read.csv2("data/MobResid08.txt",
                        stringsAsFactors = FALSE)
colnames(residFlows)

## [1] "CODGEO"  "LIBGEO"  "DCRAN"   "L_DCRAN" "NBFLUX"
```

Les colonnes **CODGEO** et **LIBGEO** donnent le code et le nom de la commune d'origine de la migration résidentielle, les colonnes **DCRAN** et **L_DCRAN** donnent le code et le nom de la commune de destination et la colonne **NBFLUX** donne le nombre de migrants correspondant à chaque couple origine-destination. Il s'agit donc ici d'un format *long* que l'on souhaite transformer en format *wide*. Le tableau de données pourrait être transformé directement avec `dcast()` mais il est plus sûr d'utiliser d'abord `melt()` pour déclarer les identifiants et les mesures.

```
meltedFlows <- melt(residFlows,
                    id.vars = c("CODGEO", "DCRAN"),
                    measure.vars = "NBFLUX")

odMatrix <- dcast(meltedFlows, CODGEO ~ DCRAN)
dim(odMatrix)

## [1] 143 144
```

On obtient ainsi une matrice origine-destination de 143 lignes par 143 colonnes qui stocke tous les flux résidentiels entre les 143 communes de Paris et de petite couronne. Une 144e colonne est créée qui contient les identifiants des communes : pour ne conserver que les flux, cette colonne pourrait être supprimée, à condition de conserver les identifiants en tant que noms de lignes avec la fonction `row.names()`.

2.5 *Packages* polyvalents

Ce chapitre a présenté les fonctions les plus utiles pour manipuler des données, en particulier des tableaux de données : sélections, tris, recodages, superpositions et jointures. La plupart de ces fonctions font partie du tronc commun de R (`r-base`). Il existe cependant deux *packages* polyvalents qui permettent d'effectuer ces opérations et qui présentent de nombreux avantages : `sqldf` et `dplyr`.

Le *package* `sqldf` contient une seule fonction éponyme avec laquelle il est possible d'exécuter des requêtes SQL. Le langage normalisé SQL (*Structured Query Language*) est très expressif et il est utilisé pour la manipulation des bases de données (sqlite, MySQL, LibreOffice Base, MS Access) ainsi que dans certains logiciels de statistiques (SAS) ou de cartographie (MapInfo). Les utilisateurs qui ont déjà une pratique du SQL pourront ainsi directement s'exprimer dans ce langage.

Voici un exemple simple qui consiste à faire une sélection de lignes et de colonnes : sélection des communes dont la superficie est supérieure à 1 500 ha et conservation de trois colonnes du tableau, le code de la commune, son nom et sa surface).

```
library(sqldf)
selecArea <- sqldf("select CODGEO, LIBELLE, SURF
                    from popCom3608
                    where SURF > 1500")
selecArea

##   CODGEO                  LIBELLE SURF
## 1  75112 Paris 12e Arrondissement 1632
## 2  75116 Paris 16e Arrondissement 1637
## 3  93005         AULNAY-SOUS-BOIS 1576
## 4  93073       TREMBLAY-EN-FRANCE 2463
```

La langage SQL exécuté dans R est identique à celui exécuté dans les autres logiciels, c'est tout l'avantage de la normalisation de ce langage de requête. Il n'est pas décrit plus précisément ici et l'utilisateur peut se référer aux très nombreux manuels et tutoriels sur le sujet.

Le *package* `dplyr` est une ré-écriture récente (janvier 2014) du *package* `plyr`. Au moment d'écrire ces lignes, il est difficile d'évaluer l'im-

pact qu'il aura sur la communauté d'utilisateurs de R mais il s'agit probablement d'un jalon majeur dans l'évolution du logiciel. Pour résumer, il permet de faire toutes les opérations décrites dans ce chapitre, sa syntaxe est plus simple et plus lisible, il est plus rapide et il résoud bon nombre de problèmes liés à la manipulation de très gros jeux de données.

Problèmes de mémoire : à la différence des bases de données et des logiciels d'analyse classiques, tous les objets utilisés dans R sont chargés dans la mémoire vive de l'ordinateur. La gestion de la mémoire est donc un problème important avec R et constitue un frein pour les utilisateurs voulant manipuler des données massives. La solution la plus simple dans ce cas est de stocker les données dans une base de données externe et d'y accéder depuis R, tâche que le *package* `dplyr` rend simple et efficace.

Les fonctions principales proposées par ce *package* sont `filter()`, `arrange()`, `select()`, `mutate()`, `group_by()` et `summarise()` auxquelles il faut ajouter les fonctions dédiées au jointures, `left_join()`, `inner_join()`, etc. La plupart de ces fonctions peuvent être exécutées sur un tableau classique (*data.frame*) mais il est conseillé de passer par le format spécifique de `dplyr`, ce qui se fait très simplement grâce à la fonction `tbl_df()`.

Quelques manipulations similaires à celles présentées plus haut avec les fonctions classiques sont répétées ici pour illustrer l'intérêt de `dplyr`. Dans un premier temps, les communes du département de Seine-Saint-Denis (93) sont sélectionnées, elles sont ensuite triées en fonction de leur population en 2008 et un tableau final est créé qui ne contient que le nom de la commune et la population.

```
library(dplyr)

##
## Attaching package: 'dplyr'
##
## The following objects are masked from 'package:plyr':
##
##     arrange, desc, failwith, id, mutate, summarise,
summarize
##
## The following objects are masked from 'package:stats':
##
##     filter, lag
##
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

popComTbl <- tbl_df(popCom3608)
selecDep93 <- filter(popComTbl, DEP == "93")
sortedPop93 <- arrange(selecDep93, POP2008)
finalTab <- select(sortedPop93, LIBELLE, POP2008)
```

Problèmes de redondance : en 2014, plus de 5 000 *packages* sont disponibles et certaines des fonctions proposées ont des noms identiques. Par défaut, si on charge un *package* qui contient une fonction déjà chargée, c'est la fonction nouvellement chargée qui masque la fonction précédente. C'est le cas ici avec les fonctions `arrange()` ou `mutate()` chargées plus haut avec `plyr` et chargée à nouveau avec `dplyr`. Pour être sûr de savoir quelle fonction est appelée, le nom du *package* peut être spécifié devant le nom de la fonction avec la syntaxe suivante (exemple) : `plyr::arrange()`.

Le *package* `dplyr` propose une nouvelle syntaxe pour enchaîner une suite d'opérations. Comparez les deux options suivantes qui exécutent le même enchaînement que dans l'exemple précédent. Dans le premier cas, l'enchaînement se fait de l'intérieur vers l'extérieur (syntaxe classique) ; dans le second cas, l'enchaînement se fait de gauche à droite grâce à l'opé-

rateur `%>%`, il est plus lisible puisqu'il correspond à l'ordre du raisonnement.

```
finalTab <- select(
              arrange(
                filter(popComTbl, DEP == "93"),
                POP2008),
    LIBELLE, POP2008)

finalTab <- popComTbl %>%
  filter(DEP == "93") %>%
  arrange(POP2008) %>%
  select(LIBELLE, POP2008)
```

Les deux autres fonctions principales de `dplyr` sont illustrées dans l'exemple suivant. Il s'agit de calculer la densité de population dans les communes en 2008 puis de calculer la densité moyenne par département.

```
popComTbl %>%
  group_by(DEP) %>%
  mutate(DENS = POP2008 / SURF) %>%
  summarise(AVGDENS = mean(DENS))

## Source: local data frame [4 x 2]
##
##   DEP AVGDENS
## 1  75  233.42
## 2  92  111.10
## 3  93   74.33
## 4  94   65.26
```

La page web dédiée au *package* `dplyr`[1] contient plusieurs vignettes très didactiques pour aller plus loin.

Au terme de ce chapitre, l'utilisateur a toutes les cartes en main pour importer et exporter des données, pour les examiner et pour les manipuler. Il dispose d'un ensemble de fonctions permettant d'effectuer les opérations les plus fréquentes en toute sécurité.

1. `http://cran.r-project.org/web/packages/dplyr/index.html`.

CHAPITRE 3

Introduction à la programmation

Objectifs : *Ce chapitre explique les principales procédures de programmation avec R : les structures itératives, les structures conditionnelles et les fonctions. Cette introduction à la programmation procure une grande souplesse dans la manipulation des données et permet d'automatiser de nombreuses procédures.*

Prérequis Notions de base du fonctionnement de R et de l'interface RStudio, telles que présentées dans les Chapitres 1 et 2.

Description des *packages* utilisés Les manipulations effectuées dans ce chapitre ne nécessitent pas l'utilisation de *packages* particuliers.

3.1 Structures itératives

Les structures itératives et les structures conditionnelles sont les briques élémentaires les plus simples des algorithmes. Les boucles `while` et `for` permettent de répéter des instructions, le test conditionnel `if...else` permet d'effectuer des instructions en fonction de l'état des variables à un moment donné.

La boucle `for`, « pour chacun de », permet de répéter une instruction sur un nombre d'itérations déterminé à l'avance. La boucle `while`, « tant que », permet de répéter une instruction tant qu'une certaine condition est vérifiée. Ainsi, la boucle `for` est un cas particulier d'une boucle `while` : elle est plus simple à utiliser et peut être préférée dans la plupart des cas. Voici la pseudo-syntaxe correspondant à ces deux structures itératives :

```
for (i in 1:n)
{
 instruction répétée n fois
 i prendra successivement toutes les valeurs entières
 comprises entre 1 et n
}

while (condition d'arrêt) {
   instruction
}
```

Les exemples d'application qui suivent présentent l'utilisation de ces structures itératives par ordre de difficulté croissant. Certains utilisent les données déjà manipulées dans les chapitres précédents sur les communes de Paris et petite couronne.

```
popCom3608 <- read.csv("data/PopCom3608.csv",
                       sep = ";",
                       stringsAsFactors = FALSE)

socEco9907 <- read.csv("data/SocEco9907.csv",
                       sep = ";",
                       stringsAsFactors = FALSE)
```

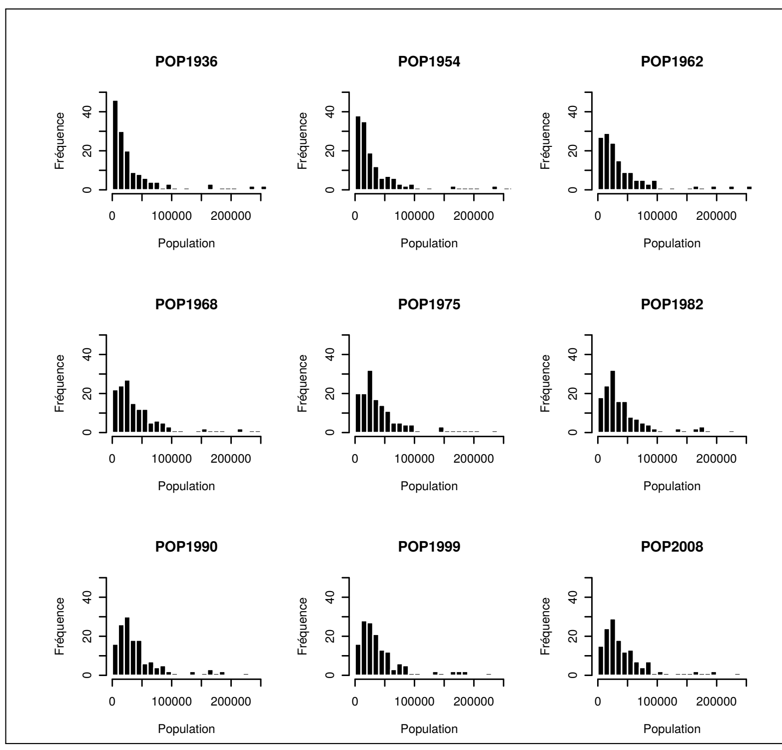
3.1.1 Exemple 1 : simple boucle `for`

À partir des données de population des recensements de 1936 à 2008, on souhaite explorer par des graphiques chacune des années du recensement sans avoir à ré-écrire la ligne de code pour chaque année. L'instruction doit être comprise de la façon suivante : pour chacune des colonnes du tableau entre 3 et 11 (toutes les variables de population), afficher l'histogramme correspondant. L'instruction suivante contient l'essentiel :

```
for(i in 3:11){
  hist(popCom3608[ , i])
}
```

Les arguments supplémentaires utilisés ici servent à personnaliser l'apparence des graphiques (cf. Chapitre 9). La fonction `par()` fixe les paramètres d'affichage graphique, dans le cas suivant les neuf graphiques seront affichés en ligne sur une grille de 3 lignes par 3 colonnes. Les arguments de la fonction `hist()` servent à afficher le nombre de barres (`breaks`), le titre (`main`), la couleur de remplissage (`col`) et de bordure (`border`), les limites pour les axes (`xlim` et `ylim`) ainsi que les titres des axes (`xlab` et `ylab`).

```
par(mfrow = c(3, 3))
for(i in 3:11){
  popYear <- colnames(popCom3608)
  hist(popCom3608[ , i],
       breaks = 20,
       main = popYear[i],
       col = "black",
       border = "white",
       xlim = c(0, 250000),
       ylim = c(0, 50),
       xlab = "Population",
       ylab = "Fréquence")
}
```



Cette série de graphiques montre le peuplement progressif de la proche banlieue parisienne. Sur les 143 communes que compte l'espace d'étude, en 1936 presque la moitié étaient peuplées de moins de 10 000 habitants. Au cours du XXe siècle la distribution de la population est modifiée et ce sont les communes moyennes, de 20 000 à 40 000 habitants, qui deviennent majoritaires.

3.1.2 Exemple 2 : double boucle `for`

Cet exemple consiste à calculer un distancier entre communes à partir des coordonnées stockées dans le tableau **socEco9907**. Pour cela on parcourt deux fois la table dans deux boucles imbriquées, afin de traiter tous les couples possibles de communes. L'indexation des lignes et des colonnes du tableau est utilisée pour parcourir le tableau en utilisant la

syntaxe présentée dans la Section 2.1, le terme `monTableau[i, j]` désignant la *ième* ligne et la *jème* colonne. Dans cet algorithme, on utilise les colonnes suivantes :

— la colonne 1 contient la liste des identifiants des communes,
— la colonne 3 contient les coordonnées X des communes,
— la colonne 4 contient les coordonnées Y des communes.

Les deux méthodes suivantes génèrent un distancier dans deux formats différents. Le premier exemple crée un format long qui prend la forme d'un tableau à trois colonnes (Com_i - Com_j - $Dist_{ij}$). Le second exemple crée un format large qui prend la forme d'une matrice carrée dont les 143 lignes sont les communes d'origine, les 143 colonnes sont les communes de destination et les valeurs sont les distances pour chaque couple de commune. Le passage d'un format à l'autre ne pose aucun problème grâce aux fonctions présentées dans la Section 2.4.3.

Pour construire un tableau en format long, on créé d'abord trois vecteurs vides : deux pour stocker les identifiants des communes et un pour stocker la distance entre chaque couple de communes.

```
commA <- vector()
commB <- vector()
distAB <- vector()
```

Puis, on remplit ces vecteurs progressivement à l'aide d'une double boucle `for` qui calcule, à chaque itération, la distance euclidienne entre un couple de communes [1]. L'approche retenue ici consiste à utiliser un compteur numérique, k, qui est mis à jour à chaque passage dans la boucle la plus petite.

1. La distance calculée est en hectomètres. Plus de détails sur les systèmes de projection dans le Chapitre 10.

```
k <- 1
n <- nrow(socEco9907)
for(i in 1:(n-1)) {
 for(j in (i+1):n) {
  commA[k] <- socEco9907[i,1]
  commB[k] <- socEco9907[j,1]
  distAB[k] <- sqrt((socEco9907[i,3] - socEco9907[j,3])^2 +
                    (socEco9907[i,4] - socEco9907[j,4])^2)
  k <- k + 1
 }
}
```

Une fois remplis ces trois vecteurs, ils sont réunis dans un même tableau qui contiendra 3 colonnes et 10 153 lignes correspondant aux couples uniques de communes et excluant la distance d'une commune à elle-même (diagonale de la matrice dans l'exemple 2).

```
distCom <- data.frame(COMA = commA,
                      COMB = commB,
                      DISTANCE = distAB)

dim(distCom)

## [1] 10153     3
```

On crée dans ce second exemple une matrice origine-destination complète qui prendra la forme d'une matrice carrée et symétrique de 143 lignes et 143 colonnes. Cette approche consiste à créer une matrice vide, puis à la remplir par une double boucle.

```
n <- nrow(socEco9907)
matDist <- matrix(0, nrow = n, ncol= n,
                  byrow=FALSE,
                  dimnames = list(socEco9907$CODGEO,
                                  socEco9907$CODGEO))

for(i in 1:(n-1)) {
 for(j in (i+1):n) {
  temp <- sqrt((socEco9907[i,3] - socEco9907[j,3])^2 +
               (socEco9907[i,4] - socEco9907[j,4])^2)
  matDist[i,j] <- temp
  matDist[j,i] <- temp
 }
}

dim(matDist)

## [1] 143 143
```

La construction d'un distancier par une double boucle est intéressante ici dans un cadre pédagogique, mais il s'agit d'une très mauvaise solution en termes de vitesse de calcul. C'est un cas typique dans lequel l'utilisation d'une fonction écrite dans un langage compilé sera bien meilleure (cf. Section 3.3.1), pas exemple la fonction `dist()` ou la fonction `rdist()` du *package* `fields`.

3.2 La structure conditionnelle `if...else`

La fonction `ifelse()` présentée dans la Section 2.3 sert à assigner une valeur à une variable en fonction d'un test conditionnel et peut être appliquée sur un tableau directement. Il ne faut pas la confondre avec la structure conditionnelle `if...else` qui sert à exécuter des instructions en fonction du résultat du test. Cette structure peut tester une seule condition ou bien un enchaînement de conditions.

Cas 1 (un test conditionnel)

```
if (test) {
  instructions si VRAI
```

```
  } else {
  instructions si FAUX
}
```

Cas 2 (deux tests conditionnels)

```
if (test1) {
  instructions si VRAI test1
  } else if (test2) {
  instructions si VRAI test2
  } else {
  instructions si FAUX test1 et test2
}
```

La structure `if...else` est le plus souvent utilisée dans une boucle ou dans une fonction. C'est l'exemple présenté dans la section suivante.

3.3 Les fonctions

Il y plusieurs raisons qui peuvent motiver la conception de fonctions propres à l'utilisateur, voici les principales : si ce dernier doit répéter plusieurs fois la même procédure et ne souhaite pas ré-écrire à chaque fois le code correspondant, s'il veut rendre la procédure plus générique et pas seulement appliquée à des données particulières, si la procédure qu'il a mise au point est suffisamment générique et utile et qu'il souhaite la mettre à disposition d'autres utilisateurs. Les fonctions sont des « boîtes » auxquelles on donne des valeurs d'entrée (arguments de la fonction) et desquelles on retire des valeurs de sortie (résultats de la fonction). Avec R, il n'y a pas de contraintes fortes sur les types d'objets qui peuvent être en argument ou en sortie d'une fonction.

3.3.1 Premier aperçu sur les fonctions

```
MaFonction <- function(arguments) {
  traitements
  return(sorties)
}
```

Voici la syntaxe générale pour écrire une fonction. Exécuter ce bloc de code, « sourcer la fonction », crée un objet nommé `MySum` dans l'environnement de travail. Cet objet peut être utilisé comme n'importe quelle fonction prédéfinie dans R, en fournissant les arguments demandés (ici `a` et `b`) on peut afficher ou récupérer une sortie (ici la somme de `a` et `b`) :

```
MySum <- function(a, b) {
  sumAB <- a + b
  return(sumAB)
}

MySum(2, 3)

## [1] 5
```

L'un des intérêts du logiciel libre (ouvert) vis-à-vis des logiciels propriétaires (fermés) est l'accès au code. Dans la pratique scientifique, l'utilisateur peut examiner précisément ce que fait un algorithme et s'il fait effectivement ce qu'il annonce. Pour voir le code d'une fonction R, il suffit d'exécuter la fonction sans les parenthèses ni les arguments.

```
matrix
plot
```

Le code de la fonction `matrix()` permet de présenter l'intégration de plusieurs langages de programmation. Parmi les fonctions implémentées dans les *packages* disponibles sur le CRAN, certaines sont écrites entièrement avec R alors que d'autres font appel à du programme écrit dans des langages compilés qui produisent des exécutables très rapides, en particulier Fortran, C et C++. L'appel à ce type d'exécutables se fait avec différentes fonctions, en particulier `.Internal()`, `.Primitive()`, `.Call()` ou encore `.Fortran()`.

Le code de la fonction `plot()` permet de présenter la notion de fonction générique. De nombreuses fonctions implémentées dans R ont un comportement différent selon le type d'objet auquel elles s'appliquent. Par exemple, la fonction `plot()` produit un nuage de points lorsqu'elle s'applique à un couple de vecteurs numériques, elle produit des graphiques

d'aide à l'interprétation si l'argument est un objet de type `lm` (*linear model*, cf. Section 5.2.3). Il en est de même de la fonction `abline()`, `summary()` et bien d'autres. La fonction applique des « méthodes » différentes selon le type d'objet donné en entrée. La fonction `methods()` renvoie la liste des différentes méthodes correspondantes à une fonction :

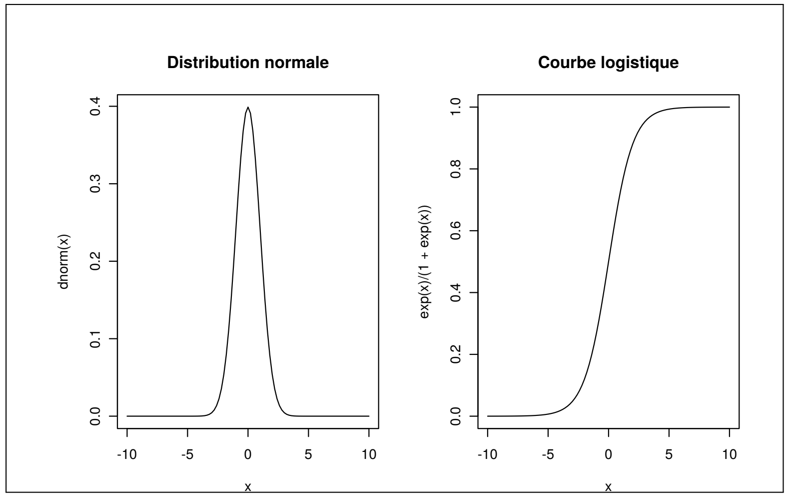
```
methods(plot)
```

L'utilisateur qui veut entrer dans le détail de la programmation objet avec R trouvera de nombreuses ressources en anglais. En français, elles sont plus rares, on citera en particulier les manuels en ligne de Christophe Genolini [1].

Les fonctions possèdent parfois des fonctionnalités de représentation graphique. La fonction `curve()` prend ainsi en argument une fonction et des bornes et affiche le graphique correspondant. À titre d'exemple on peut représenter la densité de probabilité de la loi normale. Il est aussi possible de représenter graphiquement des fonctions définies directement en argument de la fonction `curve()`, `x` étant par défaut l'argument de la fonction mathématique qu'on souhaite représenter.

```
curve(expr = dnorm,
      from = -10,
      to = 10,
      main = "Distribution normale")

curve(exp(x)/ (1 + exp(x)),
      from = -10,
      to = 10,
      main = "Courbe logistique")
```

1. http://christophe.genolini.free.fr/webTutorial/index.php.



Il est par ailleurs possible de dériver une fonction analytiquement dans R (et également de faire du calcul formel, c'est-à-dire de la manipulation analytique d'objets mathématiques) : par exemple, la dérivée de $x \rightarrow ax^b$ est $x \rightarrow abx^{b-1}$ ce que retrouve bien R. La possibilité de dériver des fonctions est rendue possible par la fonction `D()`.

```
D(expression(a * x^b),"x")

## a * (x^(b - 1) * b)
```

Il est maintenant temps de créer des fonctions propres. Trois applications sont présentées dans les sections qui suivent : une fonction qui discrétise automatiquement une variable continue, une fonction qui calcule l'équilibre de Wardrop et une fonction qui calcule les vols de Syracuse.

3.3.2 Discrétisation automatique

La discrétisation de variables continues est une opération fréquente dans l'analyse de données géographiques. Elle est particulièrement utilisée pour la cartographie (cf. Chapitre 10). La méthode de discrétisation utilisée dépend de la distribution de la variable. Si cette distribution est

normale, on discrétise souvent autour de la moyenne et de l'écart-type. En revanche si la distribution n'est pas normale, en particulier si elle est unimodale et très dissymétrique, on discrétise souvent en quantiles.

La fonction définie ci-dessous discrétise une variable continue en fonction d'un test de normalité : discrétisation autour de la moyenne si la distribution est normale, discrétisation en quartiles sinon.

```
DiscretiMatic <- function (vec) {
  normTest <- shapiro.test(vec)

  if(normTest$p.value > 0.1) {
    print("Discrétisation autour de la moyenne")
    valBreaks <- c(min(vec),
                   mean(vec) - sd(vec),
                   mean(vec),
                   mean(vec) + sd(vec),
                   max(vec))
    varDiscret <- cut(vec,
                      breaks = valBreaks,
                      include.lowest = TRUE,
                      right = FALSE)

  } else {
    print("Discrétisation en quartiles")
    valBreaks <- quantile(vec,
                          probs = c(0, 0.25, 0.5, 0.75, 1))
    varDiscret <- cut(vec,
                      breaks = valBreaks,
                      include.lowest = TRUE,
                      right = FALSE)
  }

  return(varDiscret)
}

popCom3608$POPDISCR36 <- DiscretiMatic(popCom3608$POP1936)

## [1] "Discrétisation en quartiles"
```

La fonction prend donc un vecteur numérique en entrée (argument) et renvoie un facteur en sortie. Elle produit au passage un message indiquant quelle méthode de discrétisation a été appliquée. Les fonctions peuvent

être utilisées de toutes les façons possibles : une fonction peut exécuter des opérations et ne rien renvoyer en sortie, elle peut renvoyer des objets ou des listes d'objets en sortie, elle peut afficher des graphiques et des messages ...

3.3.3 Calcul de l'équilibre de Wardrop

Deux routes permettent de se rendre du point A au point B. On cherche à savoir combien de personnes vont prendre chacune des deux routes sachant qu'elles ont des caractéristiques différentes en termes de temps de trajet (mesuré en minutes) et de capacité (mesuré en nombre maximum de véhicules par heure). La route 1 est plus courte mais qu'il ne s'agit que d'une route de faible capacité. La route 2 est une route de capacité plus importante mais le temps de trajet est plus long que sur la route 1 (cf. Figure 3.1).

Si l'ensemble des 4 000 véhicules cherche à prendre la route 1, celle-ci sera saturée et la vitesse plus faible. Si tout le monde prend la route 2, c'est une mauvaise stratégie puisqu'il existe une route inoccupée et plus rapide. Cet exercice consiste à trouver les flux optimaux sur chacune des deux routes, c'est-à-dire la répartition qui permet de minimiser le temps de transport total sur les deux routes.

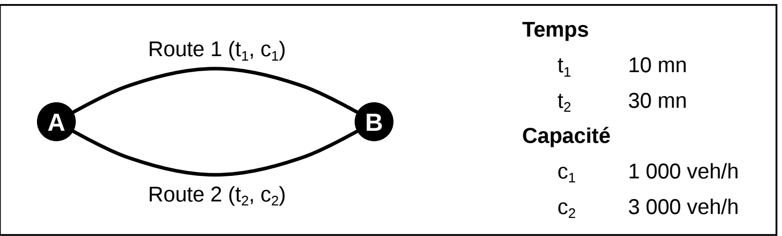


FIGURE 3.1 – Attributs des deux routes reliant les points A et B

On estime empiriquement que le temps de trajet dépend principalement de deux facteurs : le temps à vide, et la capacité de la route. Les flux sur les routes 1 et 2 sont respectivement notés $q1$ et $q2$, sachant que $q1 + q2$ égale le total de 4 000 véhicules. Les temps de trajet varient en fonction de la congestion. Cette variation est documentée empiriquement depuis

les années 1930 et formalisée par les équations suivantes (les équations et la valeur des paramètres sont des classiques du *Bureau of Public Roads* américain) :

$$t1 = 10 * (1 + 0.15 * (\frac{q1}{1000})^4)$$

$$t2 = 30 * (1 + 0.15 * (\frac{q2}{3000})^4)$$

En application de la théorie économique dite de l'équilibre de Wardrop, dans la configuration qui minimise le temps de trajet des individus $t1 = t2$. Il faut donc trouver les valeurs de $q1$ et $q2$ qui assurent cette égalité. Autrement dit, on cherche x, nombre de voitures prenant la route 1, qui soit tel que :

$$10 * (1 + 0.15 * (\frac{x}{1000})^4) - 30 * (1 + 0.15 * (\frac{4000 - x}{3000})^4) = 0$$

Il s'agit de trouver x tel que $f(x) = 0$ pour une certaine fonction f. On décide d'une première estimation x_0 et on chercher ensuite successivement des estimations plus précises d'un x tel que $f(x) = 0$ par application de la méthode de Newton (cf. Figure 3.2). Si x_0 est la première estimation du nombre de voitures sur la route 1, et sous des conditions mathématiques qu'on espère ici vérifiées, $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$ est une meilleure estimation encore du flux (f' étant la dérivée de la fonction f). Et ainsi de suite : cette suite de valeurs converge vers la bonne solution. Il convient alors de définir la fonction f, sa dérivée, ainsi que la fonction de récurrence : $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

Dans l'exemple présenté ici cette convergence numérique est effective, mais il ne s'agit pas de démontrer que cette méthode marche dans tous les cas. Par ailleurs, dès lors qu'on cherche à affecter un trafic sur un réseau plus complexe, cette méthode ne peut s'appliquer et il faut avoir recours à des algorithmes spécifiques.

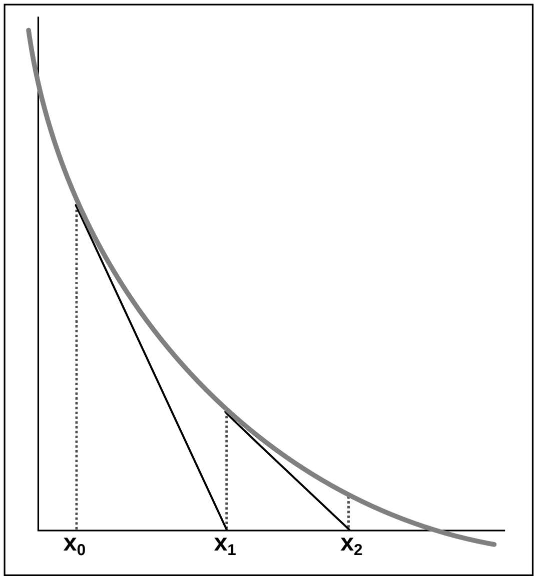


FIGURE 3.2 – Convergence avec la méthode de Newton

Dans un premier temps, les valeurs des constantes sont définies :

```
alpha <- 0.15
beta <- 4
t1 <- 10
t2 <- 30
c1 <- 1000
c2 <- 3000
Q <- 4000
```

Puis les fonctions :
— *f1* est la différence entre les temps de trajets sur les deux routes (quantité à minimiser),
— *f2* est la dérivée de *f1*,
— *f3* est la fonction utilisée pour mener à bien la méthode de Newton.

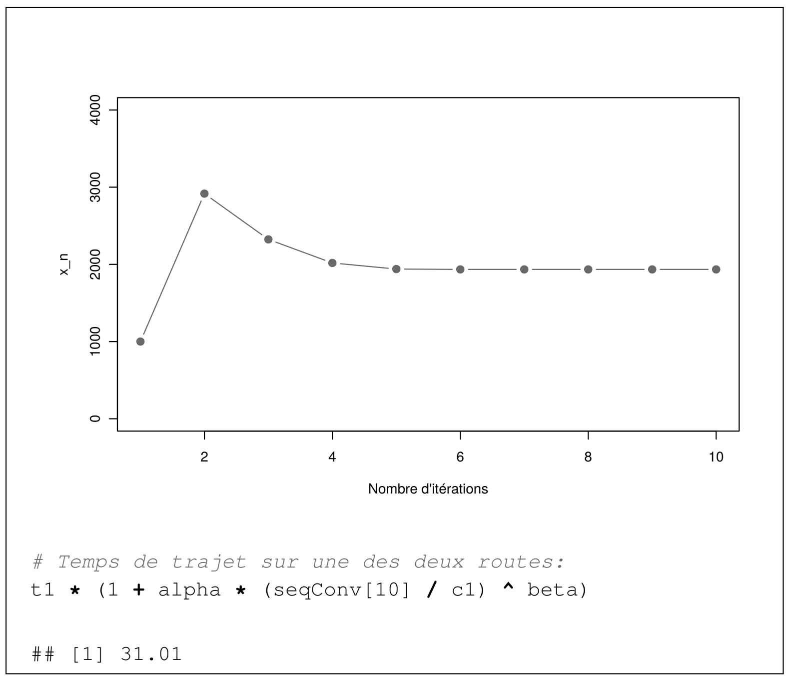
```
f1 <- function (x) {
  t1 * (1 + alpha * (x / c1) ^ beta) -
    t2 * (1 + alpha * ((Q - x) / c2) ^ beta)
}

f2 <- function (xx) {
  eval({x <- xx;
        (D(expression(
          t1 * ( 1 + alpha * (x / c1) ^ beta) -
          t2 * (1 + alpha * ((Q - x) / c2) ^ beta)),
          "x"))
        })
}

f3 <- function(x) {
  x - f1(x) / f2(x)
}
```

Une fois définies les constantes et les fonctions, la résolution numérique peut commencer. On décide arbitrairement d'une première estimation du nombre de voitures qui emprunteront la route 1 à l'équilibre, par exemple 1000 véhicules. On calcule x_1 selon la méthode de Newton à partir de cette première estimation, puis x_2, et ainsi de suite. L'algorithme ci-dessous calcule ainsi les dix premiers termes de cette suite.

```
seqConv <- NULL
seqConv[1] <- 1000
for(i in 2:10) {
  seqConv[i] <- f3(seqConv[i - 1])
}
```

On peut ainsi représenter la convergence de cette suite vers le trafic à l'équilibre et reporter le temps de trajet dans cette configuration.

```
plot(seqConv,
     type = "b",
     pch = 19,
     col = "dimgrey",
     ylim = c(0,4000),
     xlab = "Nombre d'itérations",
     ylab = "x_n")
```




```
# Temps de trajet sur une des deux routes:
t1 * (1 + alpha * (seqConv[10] / c1) ^ beta)

## [1] 31.01
```

Dans cet exemple, la valeur qui satisfait l'équilibre de Wardrop est de 2 000 véhicules sur chacune des deux routes.

3.3.4 Calcul des vols de Syracuse

Pour poursuivre l'apprentissage des fonctions et des tests conditionnels, cet exemple s'intéresse à une suite mathématique : la suite de Syracuse [1]. Partant d'un nombre quelconque, à chaque itération, le nombre suivant vaut :

— la moitié du nombre si celui-ci est pair,
— le triple du nombre augmenté de 1 si celui-ci est impair.

Autrement dit, à partir d'un entier non nul u_0, on définit la suite u_n par :

$$\begin{cases} u_{n+1} & = & u_n/2 & si & u_n & pair \\ u_{n+1} & = & 3u_n+1 & si & u_n & impair \end{cases}$$

1. `http://fr.wikipedia.org/wiki/Conjecture_de_Syracuse`.

Il semble, bien que personne ne l'ait démontré, que quelle que soit la valeur de l'entier de départ, au bout d'un certain nombre d'itérations on finit toujours par tomber sur 1. C'est ce que les fonctions suivantes vont permettre de vérifier empiriquement sur un certain nombre de cas.

Pour cela, on crée un vecteur qu'on initialise ensuite avec une valeur donnée. À l'aide d'une boucle `while`, on complète les valeurs de la suite. Il faut tester à chaque itération si le nombre est pair ou impair (avec la fonction `%%` qui renvoie le modulo) et, selon le résultat de ce test, calculer la valeur suivante.

```
Syracuse <- function(nbr)  {
  seqSyr <- nbr
  i <- 1
  while (seqSyr[i] != 1) {
    if (seqSyr[i] %% 2 == 0) {
      seqSyr[i + 1] <- seqSyr[i] / 2
      }
    else {
      seqSyr[i + 1] <- 3 * seqSyr[i] + 1
    }
    i <- i + 1
  }
  return(seqSyr)
}
```

Une seconde fonction est créée qui renvoie la longueur de la suite, c'est-à-dire le nombre d'itérations nécessaires pour converger vers 1. Ce nombre est qualifié de « durée du vol du Syracuse ».

```
SyracuseDuration <- function(x, fct) {
  length(fct(x))
}
```

Entrées et sorties des fonctions : à la création d'une fonction, tous les arguments spécifiés entre parenthèses sont les entrées de la fonction. Si un objet (données ou fonctions) n'est pas spécifié dans les arguments, R va le chercher dans l'environnement global. Quant aux sorties, R renvoie par défaut le dernier objet calculé par la fonction (voir la fonction `SyracuseDuration`). Si l'utilisateur souhaite être plus explicite ou renvoyer plusieurs objets en sortie, il peut utiliser la fonction `return()` (voir la fonction `Syracuse`).

Une fonction doit être, autant que possible, générique et sûre pour l'utilisateur. L'une des règles d'or dans l'écriture de fonctions est d'éviter ce qui a été fait plus haut avec les fonctions `f1()`, `f2()` et `f3()` : faire appel à des variables globales, c'est-à-dire des objets existant dans l'environnement global et non spécifiés dans les arguments de la fonction. Pour approfondir ce sujet, il existe d'excellentes ressources, en anglais le livre en ligne d'Hadley Wickham [1], en français les manuels en ligne de Christophe Genolini [2].

Par exemple, la fonction `f1()` fait appel à plusieurs variables - alpha, beta, t1, t2, etc. - qui sont définies en dehors de la fonction. La fonction ainsi écrite n'est pas générique et elle peut être source d'erreurs. Elle aurait plutôt dû s'écrire de la façon suivante :

```
f1 <- function (x, alpha, beta, c1, c2, t1, t2, Q) {
  t1 * (1 + alpha * (x / c1) ^ beta) -
    t2 * (1 + alpha * ((Q - x) / c2) ^ beta)
}
```

La durée des vols de Syracuse peut être calculée, grâce à la fonction définie précédemment, à partir d'un ensemble de valeurs de départ. Dans cet exemple, les vols de Syracuse sont calculés à partir des populations des communes de Paris et de petite couronne en 2008. L'idée est de déterminer s'il y a un lien la valeur de départ et la durée de vol, puis d'afficher le vol de la commune pour laquelle la durée de vol est la plus longue. L'application d'une fonction à tous les éléments d'un vecteur ou d'une

1. `http://adv-r.had.co.nz/`.
2. `http://christophe.genolini.free.fr/webTutorial/index.php`.

liste se fait grâce aux fonctions de type `apply` présentées dans la section suivante.

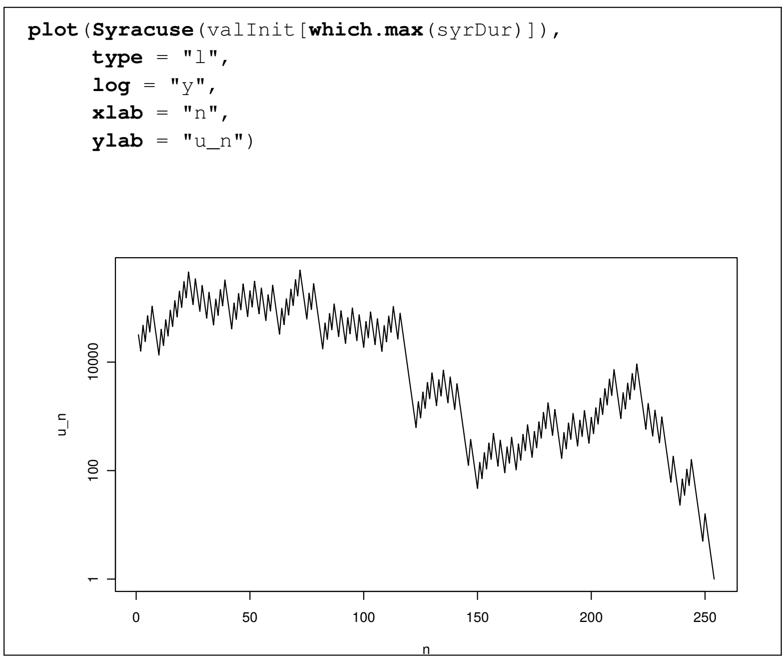
```
valInit <- popCom3608$POP2008
syrDur <- sapply(valInit,
                 SyracuseDuration,
                 fct = Syracuse,
                 simplify = TRUE)

plot(valInit, syrDur, pch = 19,
     xlab = "Population communale",
     ylab = "Durée de vol de Syracuse")
```


Il n'y a apparemment pas de lien entre la valeur de départ et la durée de vol, d'ailleurs la durée la plus importante (254 itérations) est obtenue pour une commune d'importance moyenne : Châtenay-Malabry.

```
max(syrDur)

## [1] 254

popCom3608$LIBELLE[which.max(syrDur)]

## [1] "CHATENAY-MALABRY"
```

Et voici la représentation graphique du vol de Syracuse de la commune de Châtenay-Malabry.

```
plot(Syracuse(valInit[which.max(syrDur)]),
     type = "l",
     log = "y",
     xlab = "n",
     ylab = "u_n")
```



Il est bien évident que l'application des durées de vol de Syracuse à des populations communales n'a pas d'autre sens que de jouer avec cette fonction mathématique et d'approfondir les techniques de création et de manipulation de fonctions avec R.

3.4 L'application de fonctions sur des ensembles

Le chapitre précédent a présenté certains traitements par blocs qui consistent à appliquer une fonction à un bloc de données (cf. Section 2.4.2). Ces fonctions ont été exclusivement utilisées pour produire des résumés numériques sur des sous-groupes : `tapply()` ou `aggregate()` par exemple. Cette section présente de façon plus générale comment appliquer des fonctions à des ensembles avec la famille

de fonctions `apply()`, sans qu'il s'agisse nécessairement de résumés numériques calculés selon une variable d'agrégation.

Ces fonctions sont inspirées du paradigme de la programmation fonctionnelle qui raisonne sur des ensembles. Elles sont très employées et remplacent avantageusement les boucles généralement plus lentes et plus gourmandes en ressources. L'exemple le plus simple consiste à calculer des moyennes ou des totaux marginaux sur un tableau. C'est le type de traitement que l'utilisateur d'un tableur fait fréquemment quand il entre une fonction en bout de ligne ou de colonne puis qu'il étire la fonction sur la dimension choisie.

	A	B	C	D
1	DEPARTEMENT	POP1936	POP2008	EVOLUTION
2	Paris	2830	2211	**-21.9%**
3	Hauts-de-Seine	1020	1550	**52.0%**
4	Seine-Saint Denis	776	1506	**94.0%**
5	Val-de-Marne	685	1311	**91.3%**
6	**Total**	**=SUM(B2:B5)**		-

1 - Application d'une fonction sur les lignes
Calculer le taux pour la ligne **Paris**
Tirer vers le bas pour étendre le calcul du taux à toutes les lignes

2 - Application d'une fonction sur les colonnes
Calculer la somme pour la colonne **POP1936**
Tirer vers la droite pour étendre le calcul de la somme à toutes les colonnes

FIGURE 3.3 – Application d'une fonction sur un tableur

Pour faire ce type de calcul, par exemple calculer la somme de la population de la région d'étude à toutes les dates du recensement, une solution serait d'appliquer une boucle qui calcule successivement cette somme pour chaque colonne. L'utilisation de la fonction `apply()` sera dans ce cas bien meilleure, à la fois en lisibilité et en efficacité.

```
# Application de la fonction avec une boucle
sumPop <- vector()
for(i in 3:11){
  sumTemp <- sum(popCom3608[ , i])
  sumPop <- append(sumPop, sumTemp)
}

# Application de la fonction avec apply
sumPop <- apply(popCom3608[ , 3:11], 2, sum)
```

La syntaxe générale est la suivante :

```
apply(tableau, dimension(s), fonction)
```

Les données attendues doivent être de type matrix ou array, s'il s'agit d'un *data.frame* l'objet est automatiquement transformé. Le deuxième argument est un entier désignant la dimension sur laquelle la fonction doit être appliquée : `1` pour les lignes et `2` pour les colonnes (s'il s'agit d'un tableau à deux dimensions). Il est aussi possible de travailler sur des matrices aux dimensions plus nombreuses, trois dimensions par exemple pour un cube de données. La fonction à appliquer peut être un résumé numérique simple comme c'est le cas ici, ou bien une fonction plus compliquée, créée par l'utilisateur si besoin.

Recherche d'efficacité : en termes de vitesse de calcul les boucles sont presque toujours plus mauvaises que les fonctions de la famille `apply`. De plus, le principe de la boucle, qui s'appuie à chaque itération sur les variables à l'état précédent, interdit toute tentative de calcul parallèle. Ceci est un des gros avantages des fonctions de la famille `apply` qui peuvent facilement être exécutées sur plusieurs processeurs avec des *packages* tels que `parallel` ou `multicore`.

Pour appliquer une fonction élément par élément sur un objet à une seule dimension, un vecteur ou une liste, la fonction à utiliser est `lapply()`. La syntaxe générale est la suivante :

```
lapply(liste, fonction)
```

Par défaut le résultat renvoyé par `lapply()` est une liste. Il existe d'autres fonctions dérivées de `lapply()` qui renvoient d'autres types d'objet en sortie : `sapply()` et `vapply()` sont les plus courantes. C'est l'exemple présenté plus haut pour le calcul de la durée des vols de Syracuse (cf. Section 3.3.4).

L'exemple présenté ci-dessous reprend les données historiques de la campagne de Russie des troupes de Napoléon. Ces données sont issues de la représentation cartographique de Charles Joseph Minard [1], considérée par certains comme le meilleur graphique jamais produit.

1. `http://commons.wikimedia.org/wiki/File:Minard.png`.

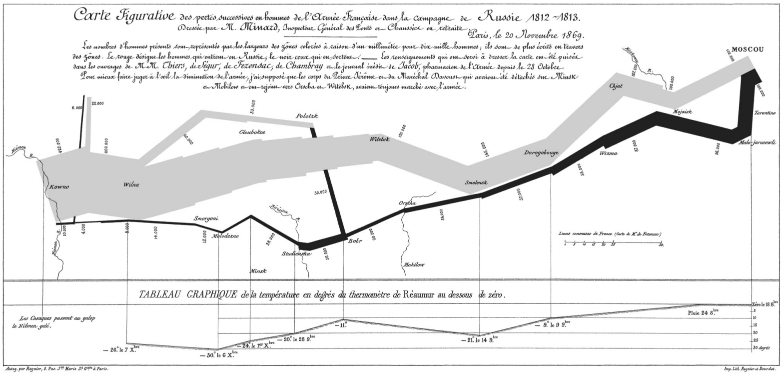


FIGURE 3.4 – Le graphique de Minard (source : Wikimedia)

Les données sont mises à disposition dans le package `HistData`, elles contiennent les coordonnées des points constitutifs des segments de la carte, le nombre de survivants à chaque point, la direction (avancée ou retraite) ainsi que le groupe qui distingue : (1) l'armée principale, (2) le flanc gauche en direction de Polotsk et (3) le bataillon du Nord en direction de Riga.

```
library(HistData)
data(Minard.troops)
```

L'idée est de calculer les pertes, en valeur absolue et relative, pour ces trois groupes. Dans un premier temps, le tableau est réorganisé dans une liste de trois éléments correspondant aux trois groupes de l'armée.

```
minardGroups <- list(
   MAIN = Minard.troops$survivors[Minard.troops$group == 1],
   LEFT = Minard.troops$survivors[Minard.troops$group == 2],
   NORD = Minard.troops$survivors[Minard.troops$group == 3]
)
```

Dans un second temps, on construit une fonction qui, pour chaque élément de la liste, récupère le minimum et le maximum puis calcule la différence et le rapport entre le deux. Cette fonction est ensuite appliquée à la liste grâce à la fonction `lapply()` ou `sapply()`.

```
Casualties <- function(vec){
  absDif <- max(vec) - min(vec)
  relDif <- round(absDif / max(vec), digits = 2)
  return(list(ABS = absDif, REL = relDif))
}

# Sortie en format liste
casGroupsList <- lapply(minardGroups, Casualties)

# Sortie en format matrice
casGroupsMat <- sapply(minardGroups,
                       Casualties,
                       simplify = TRUE)
casGroupsMat

##      MAIN   LEFT  NORD
## ABS 336000 32000 16000
## REL 0.99   0.53  0.73
```

Les pertes sont très différentes dans les trois groupes : le flanc gauche de l'armée n'a perdu « que » 53 % de son effectif alors que l'armée principale a été complètement décimée perdant 99 % de son effectif. Ce calcul aurait pu être effectué sur le tableau d'origine avec les fonctions des packages `plyr` et `dplyr` présentés dans le chapitre précédent.

Comme annoncé en introduction, il y a toujours différentes façons d'aboutir à un même résultat et chacun est libre de choisir celle qui lui convient le mieux. Dans ce choix, il faut savoir se frayer un chemin dans l'ensemble des solutions possibles et en choisir une qui soit lisible, pour soi-même et pour les autres, et qui soit efficace, surtout si le traitement est gourmand en puissance de calcul.

Au terme de ce chapitre, l'utilisateur sait utiliser les structures itératives et les tests conditionnels. Il comprend mieux l'objet fonction et il est capable de créer ses propres fonctions. Enfin, il sait appliquer ces fonctions à des blocs de données, ce qui lui confère une force de frappe importante pour attaquer des jeux de données de grande taille.

CHAPITRE 4

Analyse univariée

Objectifs : *Ce chapitre vise à présenter des manipulations simples nécessaires pour calculer des variables dérivées, réaliser des analyses univariées et afficher les représentations graphiques associées.*

Prérequis Valeurs centrales (moyenne, médiane) ; mesures de dispersion (variance, écart-type) ; discrétisation de variables continues ; représentations graphiques univariées.

Description des *packages* utilisés La grande majorité des fonctions utilisées font partie des *packages* de base installés et chargés par défaut : le *package* `base` et le *package* `stats`. Seul un *package* supplémentaire sera utile ici : `classInt` développé par Roger Bivand, qui contient un ensemble de fonctions utiles pour la discrétisation de variables continues.

4.1 Calculs simples et recodages

Les données utilisées sont les mêmes que dans les chapitres précédents, elles sont décrites à la Section 1.6.

```
socEco9907 <- read.csv("data/SocEco9907.csv",
                       sep = ";",
                       stringsAsFactors = FALSE)

popCom3608 <- read.csv("data/PopCom3608.csv",
                       sep = ";",
                       stringsAsFactors = FALSE)
```

Comme préalable aux manipulations présentées ce chapitre, plusieurs nouveaux champs sont calculés. Les variables suivantes sont créées : le taux d'évolution de la population entre 1936 et 2008 et la densité de population en 2008 ainsi que le taux d'emplois résidents en 2006 au niveau des communes et des arrondissements parisiens. Ce taux d'emploi est défini comme le rapport entre le nombre d'emplois localisés dans une commune et le nombre d'actifs résidant dans cette commune.

Deux nouvelles variables sont donc créées dans le tableau **popCom3608** et une nouvelle variable dans le tableau **socEco9907**. Pour ce genre d'instructions, la fonction `with()` (cf. Section 2.1.5) est utile pour éviter la répétition du tableau de référence.

```
popCom3608$EVOLPOP <- with(popCom3608,
                           POP2008 / POP1936 - 1)
popCom3608$DENSITE <- with(popCom3608, POP2008 / SURF)
socEco9907$TXEMPL <- with(socEco9907, EMPLOI06 / ACTOCC06)
```

Deux nouvelles variables sont ensuite créées, qui seront utilisées par la suite pour analyser les configurations spatiales à Paris et en petite couronne : le département d'appartenance (**CODDEP**) et la distance à Paris (**DISTCONT**). La variable **CODDEP** est une extraction des deux premiers chiffres du code communal (**CODGEO**) produite avec la fonction `substr()` (cf. Section 2.4.2) :

```
socEco9907$CODDEP <- substr(socEco9907$CODGEO, 1, 2)
```

La variable **Distance à Paris** sera créée en deux temps : on calcule d'abord une variable continue de distance au centre de Paris en considérant que le 1er arrondissement est une approximation du centre géographique (barycentre) de Paris, puis on discrétise cette variable en quatre classes : < 5 km, 5-10 km, 10-15 km, > 15 km. Un rappel sur la formule de la distance euclidienne entre deux points i et j de coordonnées X et Y :

$$D_{ij} = \sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2}$$

Pour référencer les coordonnées géographiques du 1er arrondissement, deux solutions sont proposées : stocker ces coordonnées dans un vecteur à part (Option 1), ou bien faire une sélection à l'intérieur de la formule de calcul de la distance (Option 2), ce qui peut être effectué en indiquant les numéros de ligne et de colonne des coordonnées du 1er arrondissement ou en appelant les variable X et Y par exemple (cf. Section 2.1.5). La distance obtenue est divisée par 10 pour obtenir une mesure en kilomètres parce que le système de projection est en hectomètres (plus de détails sur les systèmes de projection dans le Chapitre 10).

```
# Option 1
coordPremier <- as.numeric(socEco9907[1, 3:4])
socEco9907$DISTCONT <- 0.1 * sqrt(
  (socEco9907$X - coordPremier[1]) ** 2 +
    (socEco9907$Y - coordPremier[2]) ** 2
)

# Option 2
socEco9907$DISTCONT <- 0.1 * sqrt(
  (socEco9907$X - socEco9907[1, 3]) ** 2 +
    (socEco9907$Y - socEco9907[1, 4]) ** 2
)
```

Maintenant il ne reste plus qu'à discrétiser la variable continue **DISTCONT** dans une nouvelle variable **DISTCLASS**. Trois options sont proposées : la première est simple à comprendre mais pas très efficace, elle consiste à assigner une valeur entre 1 et 4 en sélectionnant les cas où la

condition est vérifiée. La deuxième est meilleure, elle consiste à emboîter une série de tests conditionnels avec la fonction `ifelse()`. La troisième option est sans doute la meilleure, elle consiste à créer la variable discrète avec la fonction `cut()` qui découpe une variable continue selon des seuils (`breaks`). Dans les trois cas, il est intéressant de vérifier avec la fonction `class()` le type de la variable créée : il s'agit d'un vecteur numérique dans les options 1 et 2 et d'un facteur dans l'option 3.

```
# Option 1
socEco9907$DISTCLASS[socEco9907$DISTCONT < 5] <- 1
socEco9907$DISTCLASS[socEco9907$DISTCONT >= 5] <- 2
socEco9907$DISTCLASS[socEco9907$DISTCONT >= 10] <- 3
socEco9907$DISTCLASS[socEco9907$DISTCONT >= 15] <- 4

# Option 2
socEco9907$DISTCLASS <-
  ifelse(socEco9907$DISTCONT < 5, 1,
         ifelse(socEco9907$DISTCONT < 10, 2,
                ifelse(socEco9907$DISTCONT < 15, 3, 4)))


# Option 3
breaksDist <- c(0, 5, 10, 15, max(socEco9907$DISTCONT))

socEco9907$DISTCLASS <- cut(socEco9907$DISTCONT,
                            breaks = breaksDist,
                            include.lowest = TRUE)
```

4.2 Résumés statistiques

Quelques mesures de centralité et de dispersion sont calculées pour avoir un aperçu des distributions des variables utilisées. On peut calculer ces mesures au coup par coup : minimum, maximum, moyenne, médiane, écart-type, ou bien utiliser la fonction `summary()` qui renvoie toutes ces mesures sauf l'écart-type. Attention, il est nécessaire mais non suffisant de calculer ces mesures : pour explorer correctement les variables, il faut les visualiser (sur l'importance de la visualisation, voir Section 5.2.1).

```
min(popCom3608$EVOLPOP)

## [1] -0.5527

max(popCom3608$EVOLPOP)

## [1] 22.78

mean(popCom3608$EVOLPOP)

## [1] 1.851

median(popCom3608$EVOLPOP)

## [1] 0.6818

sd(popCom3608$EVOLPOP)

## [1] 3.15

summary(popCom3608$EVOLPOP)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -0.553   0.193   0.682   1.850   2.190  22.800
```

Dans les *packages* de base, il y a peu de fonctions permettant de calculer des mesures pondérées ou de produire des tableaux de contingence pondérés. Ces fonctions sont implémentées dans le *package* `Hmisc` : `wtd.mean()`, `wtd.table()` ou encore `wtd.quantile()`.

Ces fonctions sont particulièrement utiles pour qui travaille avec des données d'enquête issues d'un échantillon et dont les résultats sont assortis d'une variable de pondération. Dans le cadre d'un travail plus poussé sur ce type d'enquête, le *package* `survey` sera le plus indiqué : il permet bien sûr de calculer des mesures pondérées, mais il donne surtout tous les éléments pour traiter des enquêtes au *design* complexe.

Ici les variables ne sont pas pondérées, mais ces fonctions pourraient être utilisées pour calculer la proportion de cadres sur l'espace d'étude

en 1999. Faire la moyenne des proportions de cadres par commune n'est pas satisfaisant puisque cela revient à attribuer un même poids à toutes les communes quel que soit leur effectif de cadres.

À partir de la variable d'effectif de la population active occupée (**ACTOCC99**) et de la variable de proportion des cadres par commune (**PCAD99**), on peut calculer l'effectif total des cadres en multipliant les deux variables puis en sommant le résultat. Le rapport entre cette somme et la somme des actifs occupés donnera la proportion de cadres sur l'espace d'étude. Une solution plus rapide, qui ne demande pas la création d'une nouvelle variable, consiste à calculer la moyenne des proportions de cadres, pondérée par l'effectif de la population active occupée grâce à la fonction `wtd.mean()`.

```
library(Hmisc)
```

```
# Option 1
socEco9907$NCAD99 <- with(socEco9907, PCAD99 * ACTOCC99)

sum(socEco9907$NCAD99) / sum(socEco9907$ACTOCC99)

## [1] 26.45

# Option 2
wtd.mean(socEco9907$PCAD99, weights = socEco9907$ACTOCC99)

## [1] 26.45
```

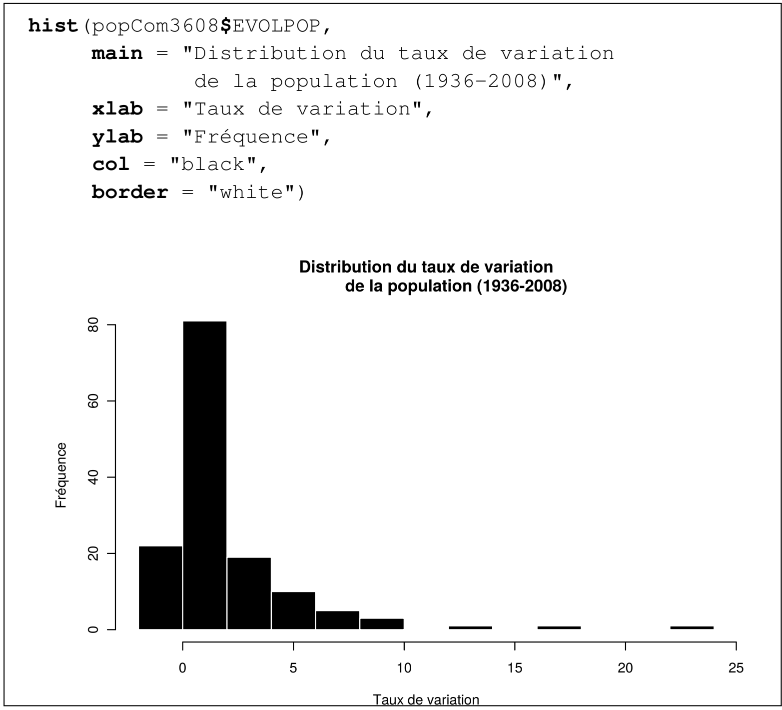
4.3 Représentation graphique des distributions statistiques

Pour explorer correctement les variables, il faut les visualiser (voir l'exemple d'Anscombe, Section 5.2.1). R propose un grand nombre de fonctions graphiques pour représenter graphiquement la distribution d'une variable. Il existe en outre plusieurs *packages* spécialisés dans les représentations graphiques, mais ici sont utilisées uniquement les fonctions

du *package* `graphics`, installé et chargé par défaut. Le nom des fonctions est en général explicite quant au type de graphique créé : `plot()`, `hist()`, `barplot()`, `boxplot()`, etc.

La fonction `hist()` renvoie par exemple l'histogramme de distribution d'une variable quantitative. Le graphique met en évidence une vingtaine de communes pour lesquelles le taux de variation est négatif, qui correspondent au vingt arrondissements parisiens plus quelques communes limitrophes.

```
hist(popCom3608$EVOLPOP,
     main = "Distribution du taux de variation
             de la population (1936-2008)",
     xlab = "Taux de variation",
     ylab = "Fréquence",
     col = "black",
     border = "white")
```



Il s'agit maintenant de calculer et visualiser l'évolution de la population depuis 1936 sur l'ensemble des 143 communes étudiées. Dans un premier temps on somme les populations communales pour chacun des recensements de 1936 à 2008. Ceci peut être fait au coup par coup, pour chaque année, avec la fonction `sum()`, mais il est plus rapide d'utiliser la fonction `apply()` qui applique une fonction à l'ensemble d'une dimension

d'un tableau (cf. Section 3.4). Elle prend comme arguments l'ensemble de colonnes à considérer (ici les colonnes 3 à 11), la dimension sur laquelle appliquer la fonction (ici la dimension 2, qui indique les colonnes) et enfin la fonction à appliquer (ici la fonction `sum()`).

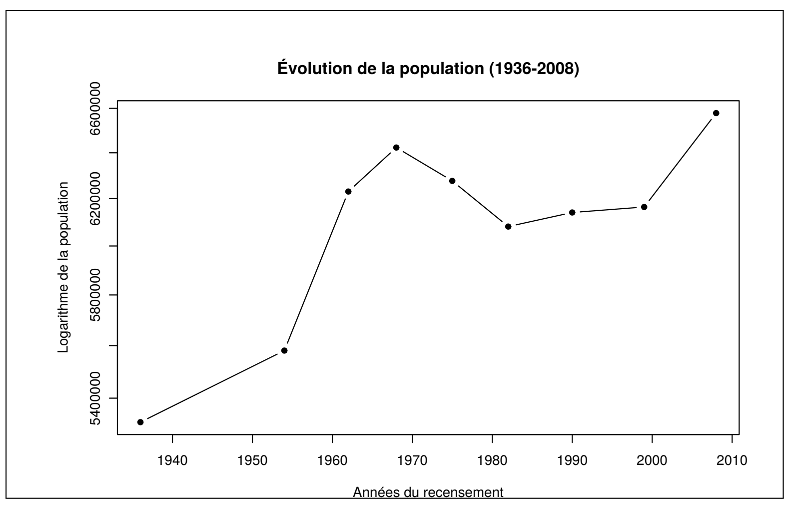
```
somPop3608 <- apply(popCom3608[,3:11], 2, sum)
```

Concernant les dates des recensements, deux méthodes sont présentées : créer un vecteur en spécifiant chaque valeur et les combiner avec la fonction `c()`, ou bien extraire les dates des noms de colonnes de l'objet créé précédemment (**somPop3608**). Noter ici l'emboîtement de trois fonctions, `names()`, `substr()` et `as.integer()` : prendre les noms dans l'objet **somPop3608**, en extraire les caractères 4 à 7 et les transformer en entiers :

```
yearsLab <- c(1936, 1954, 1962, 1968, 1975,
              1982, 1990, 1999, 2008)

yearsLab <- as.integer(substr(names(somPop3608), 4, 7))
```

Une fois créés ces deux vecteurs, on peut en faire la représentation graphique. Peu importe qu'il s'agisse de deux vecteurs distincts ou de deux champs appartenant au même tableau. Ici on précise plusieurs options, les titres du graphique et des axes, le type de représentation des valeurs (`p` pour *points*, `l` pour *lines*, `b` pour *both*), le fait que l'axe *y* est logarithmique et le type de point choisi (`pch` pour *point character*). On peut obtenir la liste des symboles ponctuels dans l'aide (`?pch`) :

```
plot(somPop3608 ~ yearsLab,
     type = "b",
     log = "y",
     pch = 20,
     main = "Évolution de la population (1936-2008)",
     xlab = "Années du recensement",
     ylab = "Logarithme de la population")
```

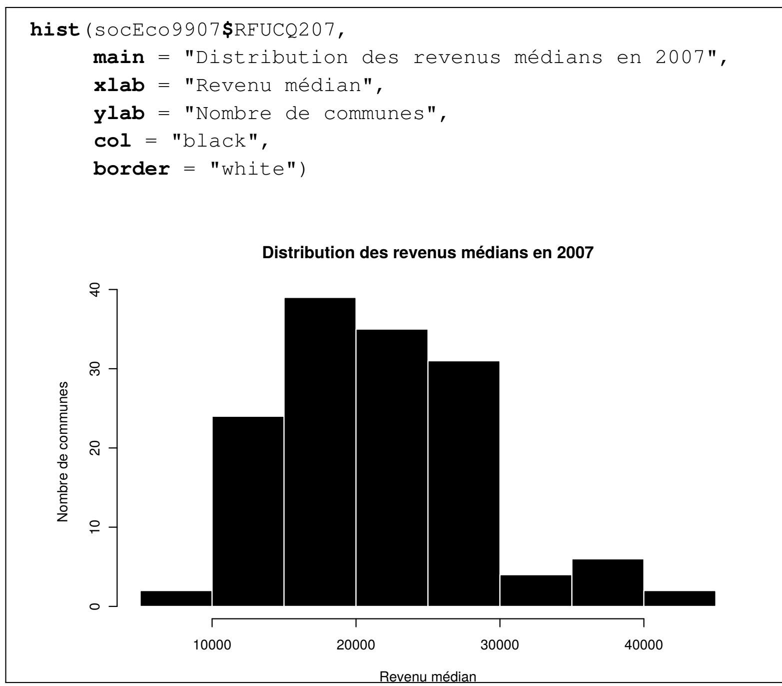


On cherche maintenant à avoir des résumés de l'ensemble des variables renseignant sur la structure socio-économique des communes en 2007 (de la variables **TXCHOMA07** à la variable **RFUCQ207**). Cette requête est immédiate grâce à la fonction `summary()` appliquée à un ensemble de colonnes par la fonction `apply()`. Le résultat est un tableau que l'on transpose avec la fonction `t()` pour obtenir un rendu plus lisible :

```
summaries07 <- apply(socEco9907[ ,25:35], 2, summary)
t(summaries07)

##           Min. 1st Qu. Median     Mean 3rd Qu.  Max.
## TXCHOMA07    6     8.0     10    11.50    14.0    23
## INTAO07      0     1.0      1     1.34     2.0     3
## PART07       2     4.0      4     4.88     5.5    11
## PCAD07       5    15.0     24    26.00    37.5    54
## PINT07      16    23.0     26    25.40    28.0    39
## PEMP07      15    22.0     30    28.60    35.0    43
## POUV07       3     8.5     14    15.10    21.5    37
## PRET07       8    13.0     15    15.20    17.0    21
## PMONO07      5     8.0     10    10.50    12.0    19
## PREFETR07    4    10.0     14    16.30    21.0    46
## RFUCQ207  9400 16100.0  20900 21700.00 26200.0 42900
```

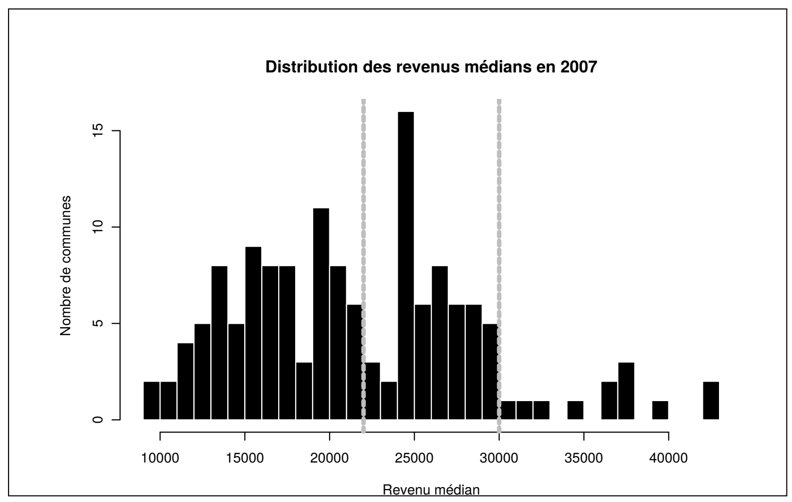
Finalement, on s'intéresse plus précisément à la distribution de la variable de revenus médians en 2007 :

```
hist(socEco9907$RFUCQ207,
     main = "Distribution des revenus médians en 2007",
     xlab = "Revenu médian",
     ylab = "Nombre de communes",
     col = "black",
     border = "white")
```



À partir de l'histogramme, on recode la variable de revenus médians en 2007 en trois classes pour préparer les analyses bivariées. La sortie par défaut de la fonction `hist()` n'indique aucun seuil qui permettrait de discrétiser la variable de façon pertinente. Si on force cette fonction à faire des barres plus fines (`breaks`) on observe une distribution trimodale, avec un seuil vers 22 000 € et un autre vers 30 000 €.

```
hist(socEco9907$RFUCQ207,
     main = "Distribution des revenus médians en 2007",
     breaks = 35,
     xlab = "Revenu médian",
     ylab = "Nombre de communes",
     col = "black",
     border = "white")

abline(v = c(22000, 30000), col = "grey", lty = 2, lwd = 4)
```



On crée ensuite la variable discrétisée en découpant la variable continue selon les seuils observés dans l'histogramme précédent, grâce à la fonction `cut()` :

```
breaksRev <- c(0, 22000, 30000, max(socEco9907$RFUCQ207))

socEco9907$REVCLASS <- cut(socEco9907$RFUCQ207,
                           breaks = breaksRev,
                           include.lowest = TRUE,
                           labels = c("[9000, 22000[",
                                      "[22000, 30000[",
                                      "[30000, 40000["))
```

À noter qu'il existe un *package* spécialisé dans la discrétisation nommé `classInt`. Il comprend des fonctions très utiles pour faire des cartes choroplèthes (cf. Chapitre 10) et ainsi retrouver les modes de discrétisation des logiciels classiques de cartographie. L'algorithme de Jenks utilisé ici crée des classes en minimisant la variance intraclasse et en maximisant la variance interclasse. L'objet créé par la fonction `classIntervals()` du *package* `classInt` est un objet spécifique de type *classIntervals*. Les seuils sont contenus dans un attribut `breaks` utilisé pour discrétiser comme précédemment. Au passage on note que ces

seuils sont comparables à ceux fixés avec la méthode visuelle, à partir de l'histogramme.

```
library(classInt)
```

```
discretJenks <- classIntervals(var = socEco9907$RFUCQ207,
                               n = 3,
                               style = "jenks")

discretJenks$brks

## [1]  9404 20334 30274 42923

breaksJenks <- discretJenks$brks

socEco9907$RFUCQ207CLASSJENKS <- cut(socEco9907$RFUCQ207,
                                     breaks = breaksJenks,
                                     include.lowest = TRUE)
```

Au terme de ce chapitre, l'utilisateur dispose des principales fonctions de statistique univariée pour explorer ses données : calculer des mesures de centralité (moyenne, médiane), des mesures de dispersion (variance, écart-type) et produire des graphiques simples, en particulier des histogrammes. Ces résumés statistiques, ces représentations graphiques et ces nouvelles variables calculées ou recodées permettent maintenant d'aborder les analyses bivariées.

CHAPITRE 5

Analyse bivariée

Objectifs : *Ce chapitre propose un ensemble de méthodes et de représentations graphiques de statistique bivariée appliquées à l'étude de données géographiques.*

❋

Prérequis Analyse univariée, corrélation et régression linéaire, analyse de la variance, tests d'hypothèses.

Description des *packages* utilisés Les *packages* de base installés et chargés par défaut suffisent pour les analyses proposées ici. Néanmoins, les *packages* spécialisés dans la représentation graphique, comme `ggplot2` ou `lattice`, peuvent s'avérer très utiles pour représenter des relations bi- ou multivariées comme la répartition d'une distribution au sein des différentes modalités d'une variable qualitative.

5.1 Préambule

5.1.1 Types de variables

Les représentations graphiques et les méthodes d'analyse des relations entre deux variables diffèrent selon la nature des variables. Deux grands types sont distingués : les variables quantitatives, sur lesquelles des résumés numériques peuvent être calculés (âge pour des individus, population pour des communes) ; les variables qualitatives, qui regroupent les individus dans un nombre fini de modalités (sexe pour des individus, département d'appartenance pour des communes).

Il faut bien distinguer la nature de la variable de son format de stockage, numérique ou alphanumérique. Une variable qualitative peut être codée avec des caractères alphanumériques (`Homme` et `Femme` par exemple) aussi bien qu'avec des caractères numériques (`0` et `1` par exemple, `0` signifiant Homme et `1` signifiant Femme). Une variable quantitative peut être codée avec des caractères numériques aussi bien qu'avec des caractères alphanumériques : `soixante-deux` voire `LXII` pour 62.

L'analyse d'une relation bivariée avec deux types de variables possibles, quantitatif et qualitatif, se résume à trois cas : relation entre deux variables quantitatives (Section 5.2), relation entre une variable qualitative et une variable quantitative (Section 5.3) et relation entre deux variables qualitatives (cf. Section 5.4). Ces trois cas sont étudiés successivement : (1) relation entre la densité d'habitants au kilomètre carré en 2008 et la distance euclidienne au centre de Paris ; (2) relation entre le revenu par habitant en 2007 et les classes de distance à Paris ; (3) relation entre le résultat obtenu au référendum européen de 2005 et le département d'appartenance de la commune.

5.2 Relation entre deux variables quantitatives

Cet exemple s'intéresse à la relation entre la densité de population et la distance au centre de l'agglomération parisienne, centre défini de façon simple comme le centroïde du 1er arrondissement de Paris. Avant de commencer, on récupère certaines variables calculées dans le chapitre précédent.

```
dataCom <- merge(socEco9907, popCom3608, by = "CODGEO")
```

5.2.1 Représentation graphique de la relation

Aucune démarche s'appuyant sur des données empiriques ne peut se passer d'exploration. L'exploration des données s'appuie sur les mesures de centralité et de dispersion présentée dans le chapitre précédent, mais elle doit aussi utiliser les outils de visualisation graphique. C'est ce qu'a voulu montrer Francis Anscombe dans un exemple célèbre [1], repris par Edward Tufte dans son manuel de visualisation.

Anscombe propose quatre paires de variables *x-y* : prises une à une, ces variables ont les mêmes propriétés statistiques (moyenne, médiane, variance) ; prises deux à deux, les mesures de corrélation (coefficient de Pearson, droite de régression) sont également identiques. En revanche, la visualisation de ces relations écarte toute ressemblance entre ces quatre configurations.

Comme beaucoup de jeux de données historiques, les données d'Anscombe sont contenues dans R et peuvent être appelées avec la fonction `data()`. Pour produire le nuage de points, c'est la fonction `plot()` qui est utilisée. Il s'agit d'une fonction graphique générique (cf. Section 3.3.1), appliquée à deux variables quantitatives (vecteurs numériques), elle produit un nuage de points (*scatterplot*) : un individu statistique i est représenté par un point dont les coordonnées sont les valeurs x_i et y_i prises par les deux variables à l'étude.

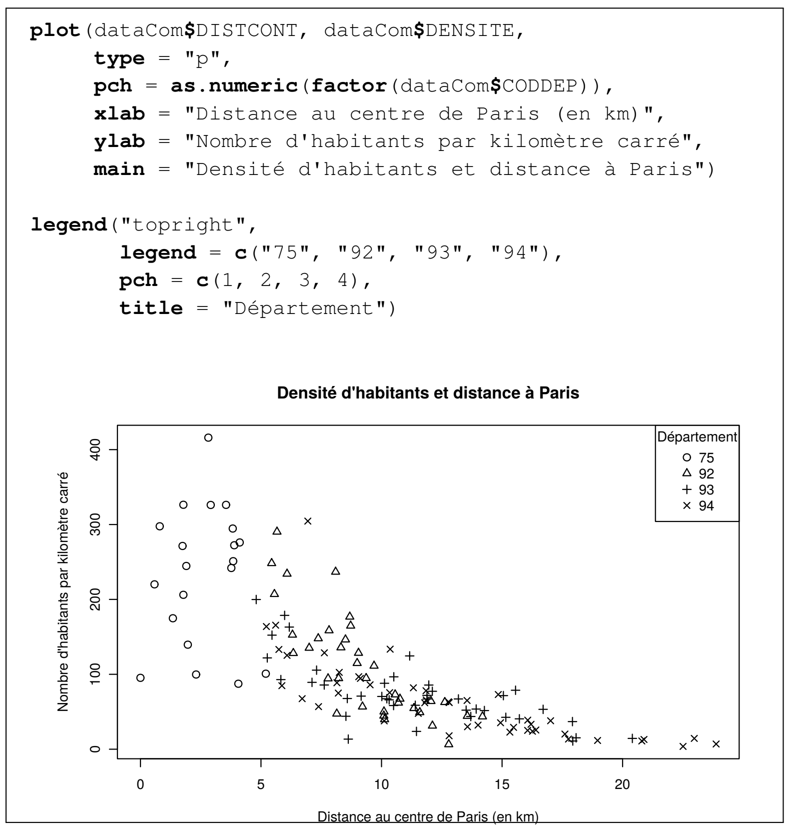
1. `http://en.wikipedia.org/wiki/Anscombe%27s_quartet`.

```
data(anscombe)

plot(anscombe$x1, anscombe$y1,
     xlim = c(4, 20),
     ylim = c(4, 14), pch = 20)
abline(lm(anscombe$y1 ~ anscombe$x1), lty = 2)

plot(anscombe$x2, anscombe$y2,
     xlim = c(4, 20),
     ylim = c(4, 14),
     pch = 20)
abline(lm(anscombe$y2 ~ anscombe$x2), lty = 2)

plot(anscombe$x3, anscombe$y3,
     xlim = c(4, 20),
     ylim = c(4, 14),
     pch = 20)
abline(lm(anscombe$y3 ~ anscombe$x3), lty = 2)
```

Dans le cadre d'une démarche exploratoire, il pourrait être intéressant de repérer les points en ajoutant le nom ou le code des communes, ou en-

core de différencier les départements d'appartenance par une variation de couleurs ou de symboles. La fonction `text()` permet d'ajouter un label aux points, la variation de couleurs peut être spécifiée avec l'argument `col` de la fonction `plot()` et la variation de symboles avec l'argument `pch`. C'est cette dernière option qui est utilisée ici pour des raisons de lisibilité sur un support papier en noir et blanc.

```
plot(dataCom$DISTCONT, dataCom$DENSITE,
     type = "p",
     pch = as.numeric(factor(dataCom$CODDEP)),
     xlab = "Distance au centre de Paris (en km)",
     ylab = "Nombre d'habitants par kilomètre carré",
     main = "Densité d'habitants et distance à Paris")

legend("topright",
       legend = c("75", "92", "93", "94"),
       pch = c(1, 2, 3, 4),
       title = "Département")
```



À la lecture du graphique, on retrouve les principes de la loi de Clark pour le cas parisien : la densité décroît avec la distance au centre selon une fonction exponentielle négative.

5.2.2 Calcul et significativité des corrélations

Afin de mesurer l'intensité des relations linéaires entre les variables quantitatives retenues pour l'analyse, il faut étudier les corrélations entre variables. Dans une démarche exploratoire, une telle analyse est particulièrement intéressante car elle permet d'éviter de traiter, par la suite, des informations redondantes.

Avant d'analyser spécifiquement une relation, on a souvent besoin de calculer la matrice de corrélations entre un ensemble de variables. C'est ce que l'on propose de faire pour un ensemble de variables quantitatives décrivant les communes de la petite couronne et les arrondissements parisiens pour l'année 2007 que nous intégrons dans un nouveau tableau : le taux de chômage, la proportion de ménages dont la personne de référence est étrangère, le revenu médian communal, la densité et la distance au centre de l'agglomération. Ce nouveau tableau est ensuite donné comme argument de la fonction `cor()` qui calcule les corrélations entre toutes les variables. La méthode utilisé ici est celle de Pearson, mais l'argument `method` permet d'en utiliser d'autres. La méthode Spearman est adaptée pour analyser des relations non linéaires parce qu'elle ne travaille pas directement sur les valeurs prises par les variables mais sur les rangs.

```
varCorr <- dataCom[, c(25, 34, 35, 39, 55)]
cor(varCorr, method = "pearson")

##           TXCHOMA07 PREFETR07 RFUCQ207 DISTCONT  EVOLPOP
## TXCHOMA07    1.0000    0.9201 -0.80343 -0.17094 -0.12933
## PREFETR07    0.9201    1.0000 -0.74938 -0.17607 -0.13228
## RFUCQ207    -0.8034   -0.7494  1.00000 -0.09421 -0.06114
## DISTCONT    -0.1709   -0.1761 -0.09421  1.00000  0.61367
## EVOLPOP     -0.1293   -0.1323 -0.06114  0.61367  1.00000
```

La matrice de corrélation proposée par la fonction `cor()` permet de connaître la valeur du cœfficient de corrélation pour chaque couple de variables. La valeur de ce cœfficient est bornée dans un intervalle compris entre -1 et $+1$. Un cœfficient négatif proche de -1 indique une forte corrélation négative entre les variables : quand l'une augmente, l'autre a tendance à décroître. À l'inverse, un cœfficient proche de $+1$ signifie qu'il existe une corrélation positive entre les variables, autrement dit que

les deux variables ont tendance à croître ou à décroître conjointement. Enfin, un cœfficient de corrélation proche de zéro décrit une absence de relation linéaire entre les deux variables.

Dans la matrice de corrélation ci-dessus, on remarque une forte corrélation linéaire positive entre le taux de chômage et la proportion de ménages dont la personne de référence est étrangère, les deux variables étant négativement corrélées au revenu médian de la commune. Le cœfficient de corrélation entre la densité et la distance au centre est assez élevé (0,75).

Le nuage de points ci-dessus montre pourtant que la relation entre densité et distance au centre n'est pas linéaire. Pour étudier cette relation et tester sa significativité, deux options sont envisageables : la première consiste à utiliser une méthode qui ne requiert pas la linéarité de la relation ni la normalité des variables (Spearman) ; la seconde consiste à transformer la variable **DENSITE** avec une fonction logarithme. La variable ainsi transformée a une distribution presque normale et la relation devient linéaire. Ces deux options sont illustrées ici.

```
cor.test(dataCom$DISTCONT,
         dataCom$DENSITE,
         method = "spearman")

cor.test(dataCom$DISTCONT,
         log(dataCom$DENSITE),
         method = "pearson")
```

La fonction renvoie la valeur du cœfficient de corrélation, les bornes de l'intervalle de confiance, la valeur du *t* de Student et la *p-value*. Ce test confirme la relation négative observée sur le nuage de points. La fonction `cor.test()` donne les éléments suivants en sortie : la statistique de test, le degré de liberté, la valeur de la probabilité critique (*p-value*), l'intervalle de confiance du cœfficient et le cœfficient de corrélation. La valeur de p est inférieure à 0.001, aussi, le cœfficient de Pearson est statistiquement significatif (non nul). En outre, la valeur absolue du cœfficient de Pearson est relativement élevée ($-0,85$), ce qui indique une relation linéaire forte entre les deux variables. Le signe du cœfficient renseigne, enfin, sur le fait que la relation est négative : lorsque la distance au centre de Paris augmente, la densité de population décroît.

5.2.3 Régression linéaire simple

La relation linéaire entre les deux variables quantitatives est modélisée avec la fonction `lm()` (*linear model*) qui permet de calculer des régressions linéaires simples et multiples. La variable à expliquer (dépendante) est la variable de densité, c'est elle que l'on cherche à expliquer statistiquement par la variable de distance, qui est la variable explicative ou indépendante. Ce modèle s'écrit de la façon suivante :

$$y_i = b_0 + b_1 x_i + \epsilon_i$$

Le nuage de points de la section précédente montre que la relation entre densité de population et distance au centre n'est pas linéaire mais exponentielle. Dans ce cas, il est utile d'opérer une transformation semi-logarithmique : on modélisera le logarithme de la variable de densité. Le terme y_i est la densité de la commune i et x_i est la distance de cette commune au centre de l'agglomération. Les paramètres de la droite de régression sont notés b_0 et b_1, ils désignent respectivement l'intercept de la droite, c'est-à-dire la valeur prise par y lorsque x est égal à zéro, et sa pente, qui est le facteur constant qui lie x et y. Enfin, ϵ_i est le terme d'erreur ou résidu pour la commune i, c'est-à-dire la différence entre valeur observée et valeur modélisée de la variable y.

Transformation logarithmique : dans un modèle de régression linéaire simple, la transformation est dite logarithmique lorsque les deux variables *x* et *y* sont passées en logarithmes, elle est dite semi-logarithmique lorsque cette transformation n'est appliquée qu'à la variable à expliquer (y). Ici la relation entre densité (y) et distance au centre (x) est de type $y = exp(ax)$, la transformation semi-logarithmique donne ainsi une relation linéaire de type $log(y) = ax$.

La fonction `lm()` produit une liste d'éléments renseignant sur les estimations du modèle. La fonction `names()` liste les noms de ces différents résultats et la fonction `summary()` permet quant à elle de connaître les principaux résultats du modèle : résidus, cœfficients et qualité de l'ajus-

tement linéaire. Plus particulièrement, le tableau donne, pour chacun des deux paramètres du modèle b_0 (intercept) et b_1 (pente) :

- son estimation (colonne 1),
- l'estimation de son erreur standard (colonne 2),
- la valeur de la statistique du test de Student (colonne 3),
- la significativité du cœfficient (valeur de p).

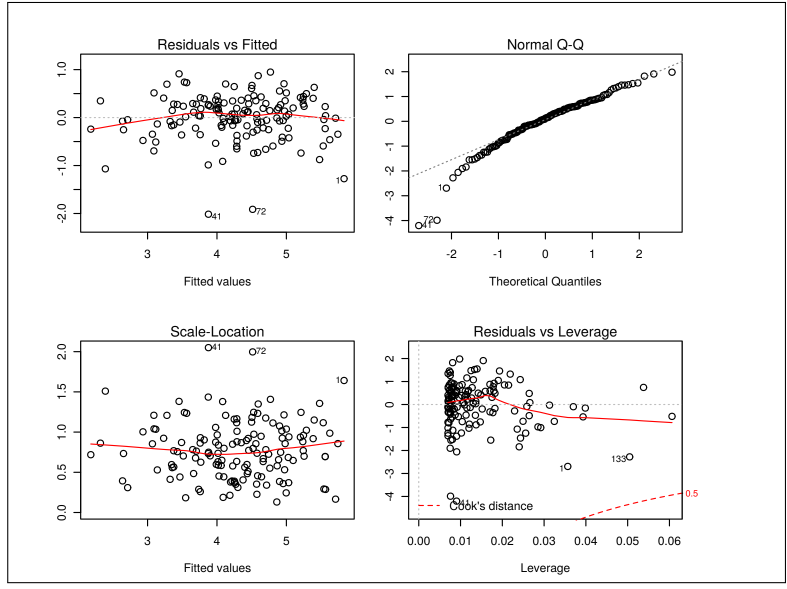
```
lmDensDist <- lm(log(DENSITE) ~ DISTCONT, data = dataCom)
summary(lmDensDist)
```

Le cœfficient de détermination (`R-squared`) est égal à 0,72, ce qui indique que 72 % de la variation du logarithme de la densité de population des communes étudiées s'expliquent par la variation de leur distance au centre de Paris (en 2008). Le signe du paramètre b_1 confirme l'observation du sens de la relation sur le nuage de points : la distance au centre de Paris a un effet négatif sur la densité de population des communes de la petite couronne parisienne. Il indique que le logarithme de la densité varie en moyenne de $0,15$ lorsque la distance au centre de Paris augmente d'un kilomètre. Le paramètre b_0 informe sur la valeur théorique de la densité de population lorsque la distance au centre de Paris est égale à 0.

La fonction `plot()` appliquée aux résultats d'un modèle de régression linéaire obtenus par la fonction `lm()` permet de représenter les quatre principales hypothèses au cœur de ce modèle :

- la normalité des résidus par rapport aux valeurs prédites (en haut à gauche de l'image)
- la normalité globale des résidus (en haut à droite de l'image)
- la corrélation entre les valeurs de la variable explicative et le carré des résidus standardisés (en bas à gauche de l'image)
- l'existence de valeurs extrêmes altérant l'estimation des paramètres (en bas à droite de l'image)

```
plot(lmDensDist)
```

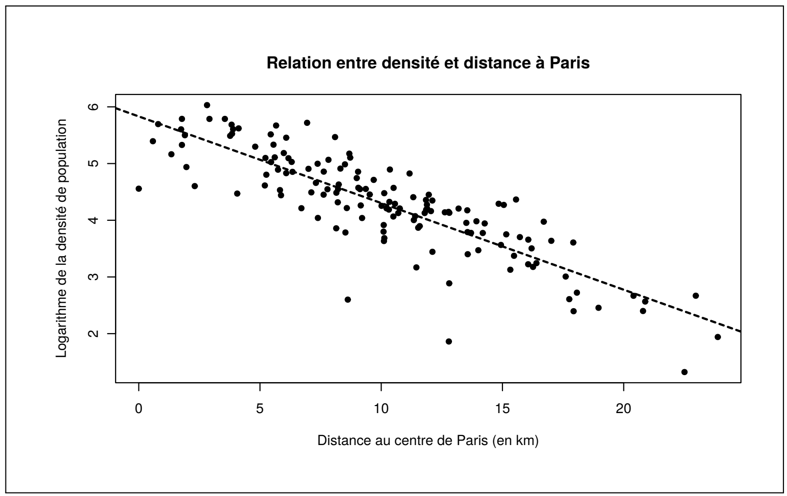


Les figures renseignant sur la normalité des résidus (figures du haut) montrent que la variance des résidus a tendance à être constante quelque soit la valeur de la densité. Les estimations de la densité de population par la distance au centre sont aussi bonnes pour toute la gamme de valeurs de densité. En outre, la figure 3 (en bas à gauche) semble montrer qu'il n'y a pas d'auto-corrélation des résidus puisque les valeurs de ces derniers ne sont pas déterminées par les valeurs de la variable dépendante. Les densités de population ne sont pas plus sur- ou sous-estimées par la distance au centre selon qu'elles sont faibles ou élevées.

Les objets spécifiques : les fonctions utilisés pour calculer des modèles statistiques renvoient des objets de type spécifique : objet `lm` pour la fonction `lm()` ou objet `htest` pour la fonction `chisq.test()` par exemple. Ces objets sont des listes qui stockent tous les éléments nécessaires au calcul et à l'interprétation du modèle. On accède à ces objets avec l'opérateur `$` de la même façon qu'on accède à une colonne de *data.frame* : `objetModele$residuals` pour récupérer les résidus par exemple. Penser à utiliser la touche `Tab` pour l'autocomplétion.

Pour améliorer le modèle, il serait intéressant de corriger la densité de population des 12e et 16e arrondissements, en effet presque la moitié de leur surface correspond aux bois de Boulogne et de Vincennes respectivement. Il serait aussi possible de recommencer l'analyse de régression sans intégrer les cas extrêmes (*outliers*). Le nuage de points est finalement affiché et la droite de régression est surimposée avec deux arguments graphiques : `lty` (*line type*) et `lwd` (*line width*).

```
plot(dataCom$DISTCONT,
     log(dataCom$DENSITE),
     pch = 20,
     col = "black",
     xlab = "Distance au centre de Paris (en km)",
     ylab = "Logarithme de la densité de population",
     main = "Relation entre densité et distance à Paris")

abline(lmDensDist, lty = 2, lwd = 2)
```



L'analyse visuelle peut être complétée d'une cartographie des résidus du modèle : les outils de cartographie sont présentés dans le Chapitre 10.

5.3 Relation entre une variable quantitative et une variable qualitative

L'objectif de cet exercice est de décrire l'organisation spatiale de la richesse en Île-de-France : l'hypothèse est que le niveau départemental permet de résumer l'hétérogénéïté de la richesse dans la région.

5.3.1 Représentation graphique de la relation

La fonction `boxplot()` est utilisée pour produire des graphiques appelés diagrammes en boîtes ou boîtes à moustaches (*boxplot* ou *box-and-whisker plot*). Ces graphiques résument visuellement la distribution statistique d'une variable quantitative autour de différentes mesures de centralité et de dispersion.

Dans le cas d'une analyse entre une variable quantitative et une variable qualitative, la boîte à moustache montre la distribution de la variable quantitative pour chacune des modalités de la variable qualitative.

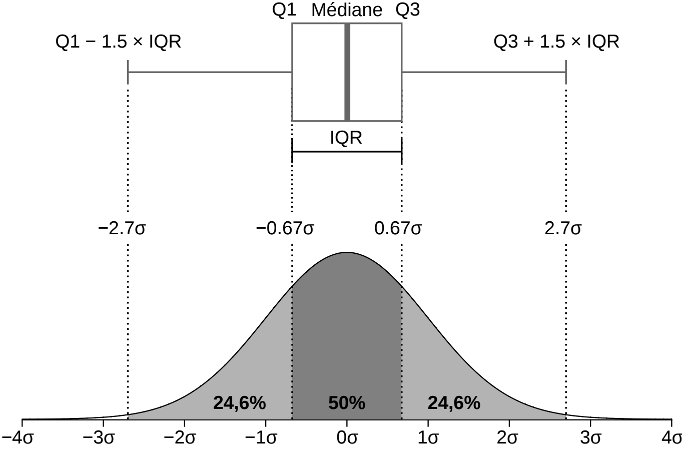
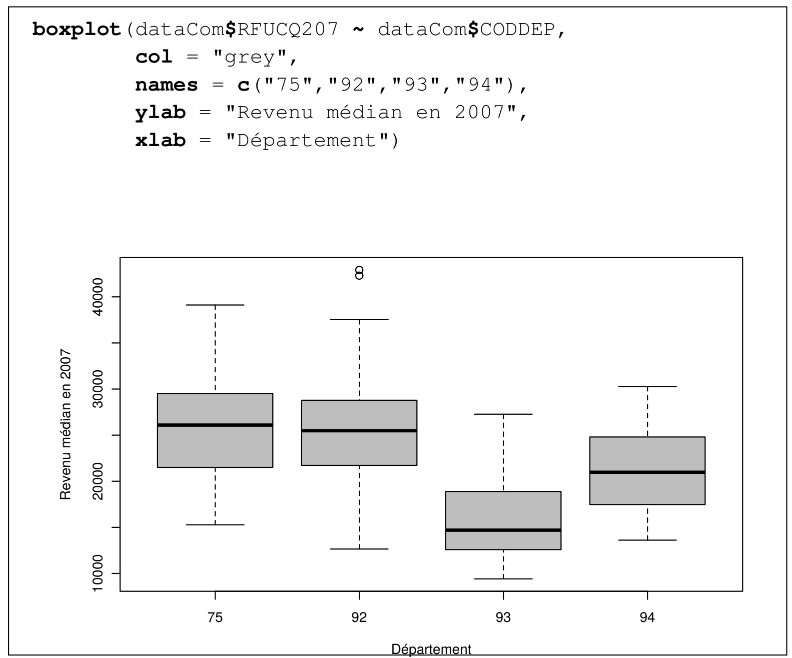


FIGURE 5.1 – Construction de la boîte à moustache (source : Wikimédia)

```
boxplot(dataCom$RFUCQ207 ~ dataCom$CODDEP,
        col = "grey",
        names = c("75","92","93","94"),
        ylab = "Revenu médian en 2007",
        xlab = "Département")
```



L'analyse visuelle donne une première idée des inégalités de revenus entre les quatre départements à l'étude : dans le 75 (Paris) et le 92 (Hauts-de-Seine), la médiane du revenu médian communal est d'environ 26 000 euros annuels, dans le 94 (Val-de-Marne) la médiane est de 21 000 euros et dans le 93 (Seine-Saint-Denis) elle est de moins de 15 000 euros. Un commentaire s'impose sur ce revenu « médian médian » : la variable **RFUCQ207** est la médiane du revenu des individus calculé au niveau de la commune, en effet la mesure de centralité utilisée pour les variables de revenu est toujours la médiane parce que la distribution des revenus est très dissymétrique (le calcul du taux de pauvreté par exemple repose sur la médiane et non sur la moyenne). La boîte à moustache indique la médiane au niveau départemental du revenu médian communal, cette mesure présente donc deux niveaux de résumé statistique et elle est différente du revenu médian des individus calculé au niveau départemental.

5.3.2 Analyse de la variance à un facteur

Une analyse de la variance à un facteur (ANOVA) est mise en place pour modéliser la relation entre le département d'appartenance des communes et le revenu médian communal. Il s'agit plus précisément de comparer les moyennes empiriques de la variable décrivant les revenus médians des communes pour les différents départements étudiés et la variabilité autour de ces moyennes. Deux options sont possibles avec R : (1) estimer les paramètres du modèle avec la fonction `lm()` puis analyser la variance avec la fonction `anova()`, ou (2) faire directement l'analyse de la variance avec la fonction `aov()`. Celle-ci présente la décomposition de la variance totale entre la variance due au facteur (variance intergroupe) et la variance résiduelle (ou variance intra-groupe). Les deux solutions, qui offrent des résultats identiques, sont proposées ci-dessous.

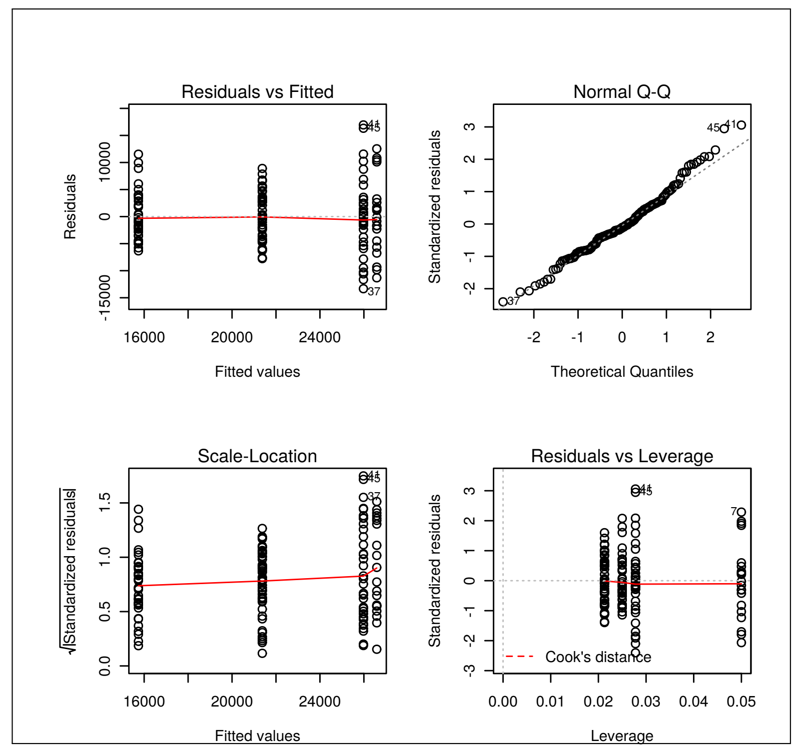
Bien que la méthode employée transforme automatiquement la variable dépendante en variable catégorielle lors de la mise en place de l'ANOVA, il est préférable d'effectuer cette transformation en amont : (1) en vérifiant que la variable catégorielle est définie ou non comme telle dans le tableau avec la fonction `is.factor()`, et (2), dans le cas contraire, en la transformant à l'aide de la fonction `as.factor()`.

```
# Transformation de la variable qualitative
is.factor(dataCom$CODDEP)
dataCom$CODDEP <- as.factor(dataCom$CODDEP)

# Anova - Option 1
lmRevDep <- lm(RFUCQ207 ~ CODDEP, data = dataCom)
anova(lmRevDep)

# Anova - Option 2
aovRevDep <- aov(RFUCQ207 ~ CODDEP, data = dataCom)
summary(aovRevDep)
```

Les résultats du modèle sont donnés dans un tableau présentant les principaux éléments, à savoir la valeur F de Fisher, les degrés de liberté (*df - degrees of freedom*), la somme des carrés des écarts (*Sum Sq*), la moyenne des carrés des écarts (*Mean Sq*) et la p-value associée au test de Fisher ($Pr > F$). Tout comme avec le premier modèle de régression linéaire issu de la régression linéaire, on vérifie les conditions d'application du modèle ainsi construit, les plus importantes étant la normalité globale des résidus et leur homoscédasticité :

— la normalité des résidus par rapport aux valeurs prédites (en haut à gauche de l'image)
— la normalité globale des résidus (en haut à droite de l'image)
— la corrélation entre les valeurs de la variable explicative et le carré des résidus standardisés (en bas à gauche de l'image)
— l'existence de valeurs extrêmes altérant l'estimation des paramètres (en bas à droite de l'image)

```
par(mfrow = c(2,2))
plot(lmRevDep)
```

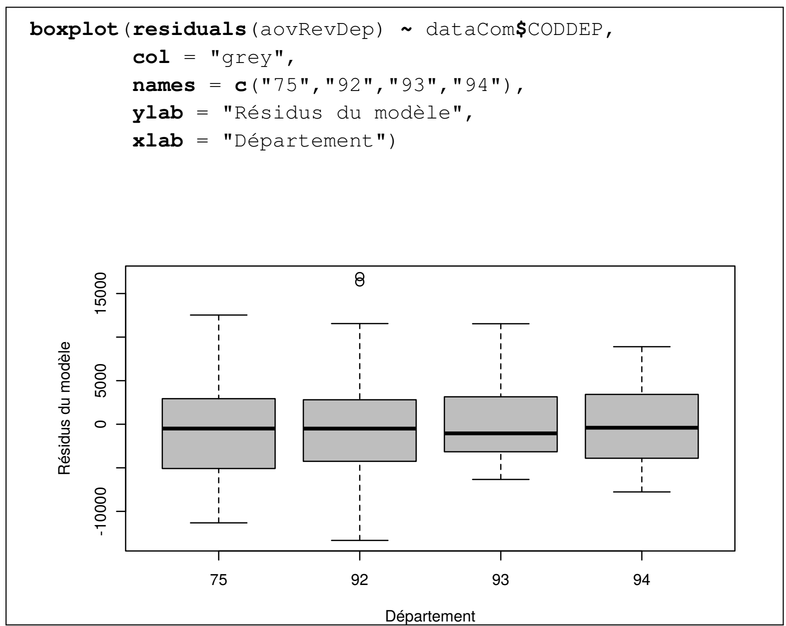


Le graphique représentant les valeurs des résidus (*residuals*) contre les valeurs ajustées (*fitted values*) par départements, en haut à gauche, montre une variance plus grande pour les départements les plus riches (où la valeur estimée est la plus importante). Cela laisse suggérer une plus grande variabilité des situations observées et donc un moins bon ajustement du modèle aux communes de ces départements. Ces résultats sont répétés dans les deux graphiques du bas. Enfin, le *quantile-quantile plot*, en haut à droite de l'image, montre une droite relativement proche d'une droite à 45 degrés, avec de rares individus statistiques s'écartant de la droite hors des valeurs les plus élevées. Ceci indique une relative normalité de la distribution des résidus malgré une légère déviance pour les cas les plus extrêmes.

Les graphiques en boîtes à moustache sont très utilisés pour l'exploration visuelle des résidus d'une analyse de la variance. Ceux-ci permettent

également de mesurer le degré d'homoscédacité des résidus en offrant une lecture plus facile.

```
boxplot(residuals(aovRevDep) ~ dataCom$CODDEP,
        col = "grey",
        names = c("75","92","93","94"),
        ylab = "Résidus du modèle",
        xlab = "Département")
```



5.4 Relation entre deux variables qualitatives

L'objectif de cet exercice est d'analyser la relation entre le vote au référendum européen [1] et le département d'appartenance (**CODDEP**). L'hypothèse est que les départements les plus riches (75 et 92) ont massivement voté pour le « oui » et les départements les plus pauvres (93) pour le « non ».

5.4.1 Représentation graphique de la relation

Dans un premier temps, la variable de réponse au recensement est créée à partir de la variable **REFEROUI** qui donne le pourcentage de vote pour

1. Référendum organisé en 2005 dans les pays membres de l'Union européenne. La question était « Approuvez-vous le projet de loi qui autorise la ratification du traité établissant une constitution pour l'Europe ? ». En France, le « non » a recueilli 55 % des votes.

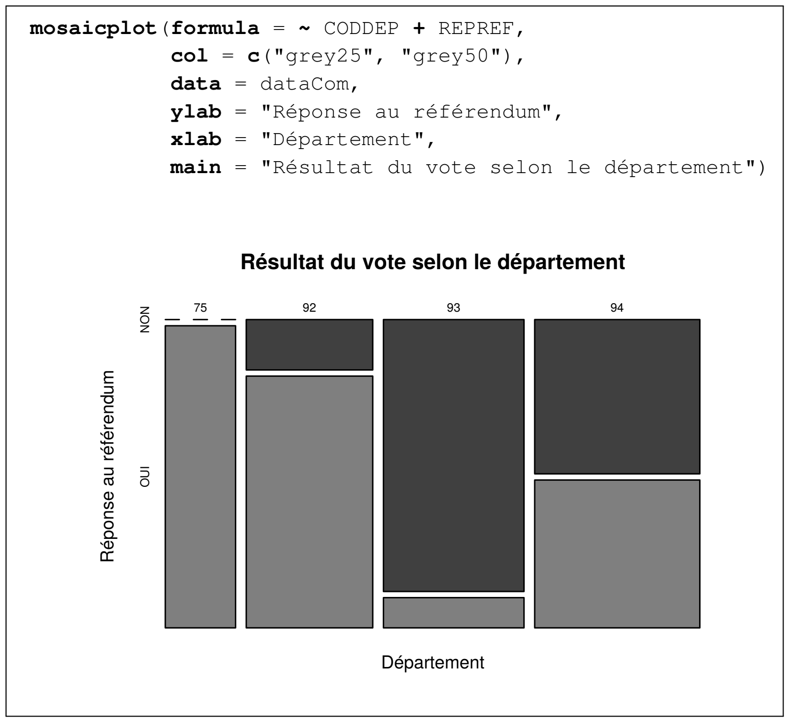
le « oui » au niveau des communes. Les communes affichant un vote supérieur à 50 % sont considérées comme globalement pour et celles affichant une vote inférieure à 50 % comme globalement contre.

```
dataCom$REPREF <- ifelse(dataCom$REFEROUI > 50,
                         "OUI",
                         "NON")
```

Le premier réflexe en présence de variables qualitatives est de faire des tableaux de dénombrement (selon une seule variable) et des tableaux de contingence (tris croisés selon deux variables), ce que permet de faire la fonction `table()`.

```
crossTabDepRef <- table(dataCom$CODDEP, dataCom$REPREF)
crossTabDepRef


##
##       NON OUI
##   75    0  20
##   92    6  30
##   93   36   4
##   94   24  23
```

La fonction `mosaicplot()` complète cette première analyse. Il s'agit de la représentation graphique du tableau de contingence : la longueur des segments sur les deux axes correspond aux fréquences relatives correspondantes et l'aire des rectangles est également proportionnelle à la fréquence relative de la sous-population représentée. Par exemple, le département du Val-de-Marne (94) compte 43 communes, soit environ 33 % du total de 143 communes à l'étude : sur l'axe horizontal, la longueur du segment de ce département correspond donc au tiers de la longueur totale de l'axe. De même, 24 de ces 47 communes ont voté pour le « non » soit environ la moitié : sur l'axe vertical, la longueur du segment correspond donc à la moitié de la longueur totale de l'axe. Enfin, cette sous-population des communes du Val-de-Marne ayant voté pour le « non », composée de 24 communes, représente 17 % de l'ensemble à l'étude : la surface du rectangle correspondant fait également 17 % de la surface utile du graphique.



5.4.2 Analyse de la répartition observée

Pour deux variables X et Y, comprenant respectivement m et n modalités, une sous-population est désignée par l'appartenance à une modalité de chacune des deux variables. Les communes du Val-de-Marne ayant majoritairement voté « non » au référendum sont un exemple de sous-population, leur effectif, au croisement des modalités des deux variables est dit « effectif conjoint ».

Le total marginal des lignes est noté n_j, celui des colonnes est noté n_i et l'effectif total est noté n··.

Le tableau de contingence présenté plus haut contient des valeurs absolues, il peut utilement être transformé pour travailler sur des valeurs relatives. Trois types de proportions sont envisageables :

— la proportion des effectifs conjoints sur l'effectif total,

	Y_1	Y_2	...	Y_j	Total	
X_1	n_{11}	n_{12}		n_{1j}	$\mathbf{n_{1\cdot}}$	Total marginal des colonnes
X_2	n_{21}	n_{22}		n_{2j}	$\mathbf{n_{2\cdot}}$	
⋮						
X_i	n_{i1}	n_{i2}		n_{ij}	$\mathbf{n_{i\cdot}}$	
Total	$\mathbf{n_{\cdot 1}}$	$\mathbf{n_{\cdot 2}}$		$\mathbf{n_{\cdot j}}$	$\mathbf{n_{\cdot\cdot}}$	Effectif total
	Total marginal des lignes					

FIGURE 5.2 – Notation des effectifs dans un tableau de contingence

— la proportion des effectifs conjoints sur le total marginal des colonnes (profil ou pourcentage en ligne),

— la proportion des effectifs conjoints sur le total marginal des lignes (profil ou pourcentage en colonne).

Les profils en ligne et en colonne sont calculés ici en passant par la fonction `prop.table()`. Cette fonction demande comme argument un objet tabulé (le tableau de contingence) et non les variables qualitatives elles-mêmes. L'argument `margin` permet de spécifier le total de référence, celui des lignes (dimension `1`), celui des colonnes (dimension `2`) ou l'effectif total (option par défaut).

```
rowPct <- prop.table(crossTabDepRef, margin = 1)
colPct <- prop.table(crossTabDepRef, margin = 2)
```

Les deux profils disent des choses bien différentes : le profil en ligne dit que 51,1 % des communes du Val-de-Marne (94) ont voté « non » au référendum, le profil en colonne dit que 36,3 % des communes qui ont voté « non » se trouvent dans le Val-de-Marne. Dans l'analyse de la relation entre les deux variables c'est le profil en ligne qui est intéressant parce que, par convention, un tableau de contingence présente la variable à expliquer (dépendante) en colonne et la variable explicative (indépendante) en ligne. Ici, la question est de savoir si l'appartenance départementale explique le vote au référendum, c'est donc la proportion qui rapporte le vote aux totaux départementaux qui est pertinente.

5.4.3 Analyse des écarts à l'indépendance

Le tableau de contingence dénombre les communes selon le croisement des modalités des deux variables, il donne la répartation observée. Cette répartition observée est comparée à une répartition théorique sous hypothèse d'indépendance, c'est-à-dire la répartition qu'on observerait s'il n'y avait aucun lien entre les deux variables. Cette répartition théorique consiste donc à redistribuer les effectifs conjoints tout en conservant les effectifs marginaux :

$$n_{ij}^e = \frac{n_{i.}n_{.j}}{n_{..}}$$

La fréquence conjointe espérée (n_{ij}^e) est le produit des totaux marginaux en ligne ($n_{i.}$) et en colonne ($n_{.j}$) divisé par l'effectif total ($n_{..}$). L'analyse ci-dessous permet de déterminer l'existence d'un lien entre le département et le vote au référendum au niveau de la commune. L'observation du tableau de contingence laisse penser que la répartition de ce vote entre départements n'est pas homogène. Le test du χ^2 calculé avec la fonction `chisq.test()` mesure la validité statistique d'une telle hypothèse.

L'objet créé par la fonction `chisq.test()` est de type `htest`, c'est une liste qui contient tous les éléments nécessaires à l'analyse : effectifs théoriques, résidus bruts ou résidus relatifs (cf. Encadré p. 93).

```
chi2DepRef <- chisq.test(x = dataCom$CODDEP,
                         y = dataCom$REPREF)
chi2DepRef


##
##  Pearson's Chi-squared test
##
## data:  dataCom$CODDEP and dataCom$REPREF
## X-squared = 61.14, df = 3, p-value = 3.36e-13
```

Le test nous donne la statistique du χ^2 (*chi-squared*), le degré de liberté associé au test (le produit du nombre de modalités - 1 de chacune des variables) et la significativité de la relation (valeur de p). La probabilité d'obtenir une valeur du χ^2 observé aussi élevée dans un échantillon de

la taille observée sous l'hypothèse d'indépendance des deux variables est inférieure à 0.001%. Il est donc possible de rejeter l'hypothèse d'indépendance pour un risque de première espèce (rejeter l'hypothèse nulle alors que celle-ci est vraie, faux positif).

Il y a une relation entre les résultats du vote au référendum au niveau communal et le département d'appartenance des communes : dans les départements les plus riches (75 et 92), le résultat est massivement positif ; dans le département le plus pauvre (93), il est largement négatif ; dans le Val-de-Marne (94) les résultats sont équilibrés.

Au terme de ce chapitre, l'utilisateur connait les principales fonctions permettant d'explorer et de modéliser des relations bivariées entre deux variables quantitatives, entre deux variables qualitatives ou entre une variable quantitative et une variable qualitative. Ces mêmes fonctions pevent être utilisées pour modéliser des relations multivariées et faire, par exemple, des analyses de la variance à plusieurs facteurs ou des régressions linéaires multiples.

CHAPITRE 6

Analyses factorielles

Objectifs : *Ce chapitre présente deux exemples d'analyses multivariées sous R : une analyse en composantes principales (ACP) et une analyse factorielle des correspondances (AFC). À l'aide de ces méthodes, il s'agit de comprendre comment l'espace de la petite couronne parisienne s'organise, de pointer les ressemblances et différenciations entre communes au regard d'une série d'indicateurs socio-démographiques.*

Prérequis Notions théoriques sur l'analyse en composantes principales et l'analyse factorielle des correspondances.

Description des *packages* utilisés Plusieurs *packages* proposent des fonctions pour faire des analyses factorielles, les deux principaux étant `ade4` et `FactoMineR` qui intègrent un ensemble de méthodes d'analyse des données dite « à la française ». C'est `ade4` qui sera utilisé ici, à la fois parce que le *package* est très bien conçu, mais aussi parce qu'il est très bien documenté. Il dispose en particulier d'un site web dédié [1] qui contribue à la qualité de cette documentation.

6.1 Analyse en composantes principales

Les données utilisées sont les mêmes que dans les chapitres précédents, elles sont décrites à la Section 1.6. C'est le tableau **socEco9907** qui est utilisé ici, tableau qui comporte un ensemble de variables socio-démographiques décrivant les communes de Paris et la petite couronne. L'analyse en composantes principales est une méthode qui s'applique uniquement sur des variables quantitatives, ici sur les caractéristiques socio-démographiques des communes de la petite couronne et des arrondissements parisiens pour l'année 2007. Toutes les variables allant de la population de moins de 20 ans (**P20ANS**) au revenu médian (**RFUCQ207**) sont utilisées, ainsi que la variable de distance au centre de l'agglomération (**DISTCONT**) créée dans la Section 4.1.

6.1.1 Réaliser l'analyse

L'ACP (PCA en anglais, *principal components analysis*) révèle la structure de différenciation entre les communes au regard des combinaisons de variables socio-démographiques contenues dans le tableau. Elle permet de dégager quelques axes de différentiation de nombreuses variables probablement très corrélées les unes avec les autres : les communes avec une forte proportion de cadres auront certainement un fort revenu médian et une faible proportion d'ouvriers par exemple.

Les données à analyser étant très hétérogènes quant à l'ordre de grandeur et à la dispersion, il faut d'abord centrer-réduire les variables. Cette opération consiste à centrer toutes les variables sur la même moyenne

1. `http://pbil.univ-lyon1.fr/ade4`.

(moyenne égale à 0) et de les réduire à la même dispersion (écart-type égal à 1). Elle peut être faite en amont avec la fonction `scale()` ou directement dans la fonction `dudi.pca()` qui calcule l'ACP. C'est cette seconde option qui est adoptée ici avec les arguments `center` et `scale`.

La fonction `dudi.pca()` ne prenant comme argument que des variables quantitatives, les codes des communes sont ajoutés en tant que nom de lignes : ils seront ainsi conservés sans affecter l'ACP.

```
library(ade4)
row.names(socEco9907) <- socEco9907$CODGEO
pca07 <- dudi.pca(socEco9907[ , 23:35],
                  center = TRUE,
                  scale = TRUE,
                  scannf = FALSE,
                  nf = 2)
```

L'objet produit par la fonction `dudi.pca()` contient tous les éléments nécessaires à une telle analyse, en particulier les coordonnées des variables et des individus sur les axes. On y accède comme d'habitude avec l'opérateur `$` (cf. Encadré p. 93, penser à utiliser la touche `Tab` pour l'autocomplétion.).

6.1.2 Examiner et interpréter les résultats

L'ACP consiste au passage d'un système d'observations dans un espace à p variables à un espace à p composantes principales. L'intérêt de cette méthode est de réduire les dimensions et de supprimer la colinéarité entre variables. Les composantes correspondent à des combinaisons linéaires de l'ensemble des indicateurs analysés. Elles se hiérarchisent en fonction de la variance des coordonnées des individus dans l'espace euclidien des variables. Les composantes principales sont des variables synthétiques qui permettent d'identifier le(s) facteur(s) principal(-aux) de différenciation au sein de la matrice initiale.

Une telle analyse produit un certain nombre d'éléments pour caractériser ces composantes, à commencer par les valeurs propres (*eigenvalues* en anglais). Celles-ci rendent compte de l'inertie du nuage de points expliquée par chaque facteur. La somme de ces valeurs propres donne la

variance totale. Pour se faire une idée de la structuration des variables, il faut examiner les parts relatives de la variance pour chaque composante ainsi que leurs parts cumulées. Ces indications sont contenues dans l'objet créé par la fonction `dudi.pca()`, elles sont regroupées ici dans un tableau et représentées sous forme graphique.

```
summaryPca <- data.frame(
  EIG = pca07$eig,
  PCTVAR = 100 * pca07$eig / sum(pca07$eig),
  CUMPCTVAR = cumsum(100 * pca07$eig / sum(pca07$eig))
)

barplot(summaryPca$PCTVAR,
        xlab = "Composantes",
        ylab = "Pourcentage de la variance (inertie)",
        names = paste("C", seq(1, nrow(summaryPca), 1)),
        col = "black",
        border = "white")
```


Dans cet exemple, le graphique montre que l'ensemble des variables est très structuré avec des corrélations fortes. Il apparaît que 65,5 % de l'information contenue dans le tableau initial est résumée par le premier facteur. Le pouvoir discriminant des axes suivants est relativement faible.

L'interprétation du rôle des variables dans la structuration de l'information apportée par l'ACP s'effectue grâce à l'observation de leurs coordon-

nées sur les deux premiers axes : celles-ci représentent le coefficient de corrélation entre une variable et un axe.

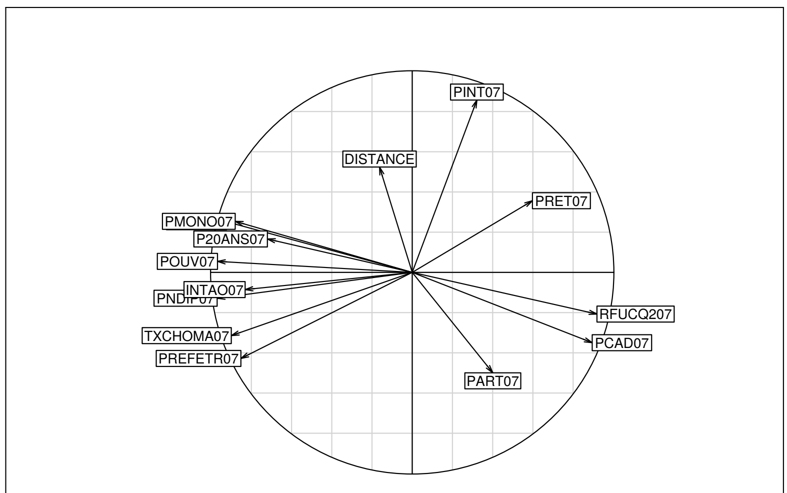
```
axisCoord <- pca07$co
```

Les coordonnées varient entre -1 et +1 : les valeurs proche de 0 correspondent à une absence de relation linéaire entre la variable et l'axe ; les valeurs absolues élevées, proches de 1, indiquent une forte corrélation entre la variable et l'axe ; le signe indique si la relation est positive ou négative.

Ici, l'axe 1 indique une opposition entre les communes aisées et les communes défavorisées, il affiche des valeurs fortement négatives pour les variables suivantes : la part des non diplômés, des chômeurs, des intérimaires, des employés et des ouvriers. Il affiche au contraire des valeurs proches de 1 pour la part des cadres et le revenu médian. L'axe 2 de l'analyse rend compte, quant à lui, d'une opposition entre communes ayant une forte proportion de professions intermédiaires au sein de la population d'actifs occupés (**PINT07**) et les autres.

La visualisation du premier plan factoriel (repère du plan défini par les deux premiers axes) permet de mieux comprendre comment les deux premiers facteurs structurent l'espace des variables. La fonction `s.corcircle()` permet de l'afficher. On y ajoute également la variable de distance au centre de l'agglomération pour voir comment elle se situe vis-à-vis des autres. Cette variable de distance est une variable dite « supplémentaire », elle est simplement projetée sur le plan mais n'entre pas en ligne de compte dans le calcul des composantes. C'est la fonction `supcol()` qui permet d'afficher cette variable, elle doit être centrée-réduite en cohérence avec les options définies plus haut pour l'analyse.

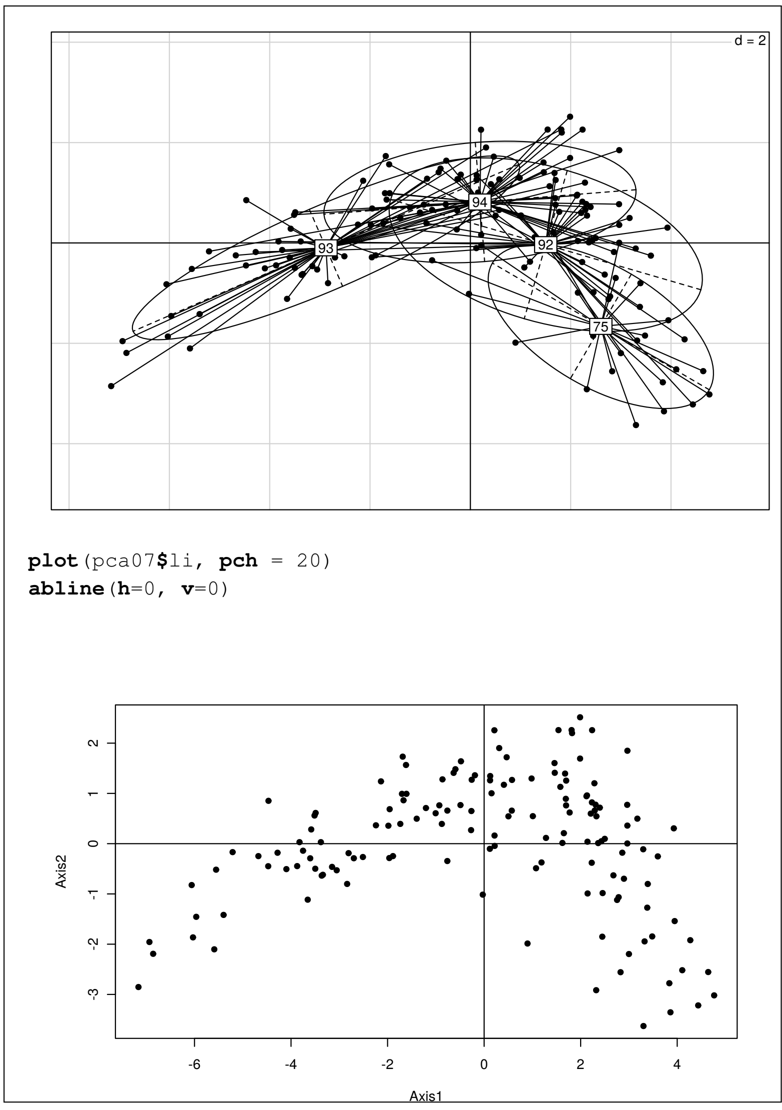
```
suplDist <- supcol(pca07, scale(socEco9907$DISTCONT))
s.corcircle(pca07$co)
s.corcircle(suplDist$cosup,
            add.plot = TRUE,
            label = "DISTANCE")
```



On remarque notamment que les communes ayant une part élevée de professions intermédiaires au sein des individus actifs sont plutôt corrélées avec une distance à Paris importante.

Une fois analysées les relations entre variables, il s'agit de s'intéresser à la façon dont les individus statistiques, ici les communes, se positionnent sur les plans factoriels. Il est souvent intéressant de surimposer à la position des individus statistiques une variable de regroupement représentée par une ellipse, ici le département d'appartenance des communes. Le centre de l'ellipse est le centre de gravité du sous-groupe et les points sont reliés au barycentre correspondant à leur sous-groupe. Il est également possible d'utiliser la fonction générique `plot()`, ou d'autres fonctions graphiques (cf. Chapitre 9), en récupérant les coordonnées des individus.

```
s.class(pca07$li,
        fac = as.factor(substr(socEco9907$CODGEO, 1, 2)))
```




```
plot(pca07$li, pch = 20)
abline(h=0, v=0)
```

6.1.3 Analyse des contributions

Une fois calculées les composantes principales, il est important de faire une analyse des contributions. Deux types de contributions peuvent être

analysées : la contribution des variables à la formation des axes ainsi que la contribution des individus statistiques. Ce type d'analyse est absolument nécessaire dans le cas d'une ACP assortie d'une pondération, elle reste importante même sur un tableau non pondéré comme c'est le cas ici.

La contribution d'une variable rend compte du rôle qu'elle a joué dans la formation de l'axe. L'ensemble de ces contributions pour un axe équivaut à 1, puisque les contributions s'obtiennent en effectuant le rapport entre le carré de la coordonnée de la variable sur l'axe et la valeur propre de ce dernier (*i.e.* la somme des carrés des coordonnées de l'ensemble des variables sur l'axe).

Ces contributions pourraient être calculées par l'utilisateur, mais le *package* `ade4` fournit une fonction qui fait le travail : la fonction `inertia.dudi()`. Les arguments `row.inertia` et `col.inertia` indiquent que les contributions des individus et des variables doivent être renvoyées.

```
contribPca <- inertia.dudi(pca07,
                           row.inertia = TRUE,
                           col.inertia = TRUE)

contribVar <- contribPca$col.abs
```

Les hauteurs des contributions des différentes variables (noter que ces contributions sont exprimées en 1/10 000) montrent que le premier facteur résulte d'une combinaison de variables plutôt que d'une spécialisation dans un domaine particulier, la dispersion autour des valeurs centrales étant relativement faible. Ceci montre également que la plupart de ces variables sont corrélées les unes avec les autres, à l'exception des professions intermédiaires et des artisans commerçants.

La contribution d'un individu à la formation d'un axe permet donc de déterminer la part qu'un individu prend dans la variance d'un axe (ici le premier axe factoriel). On peut ainsi regarder quelles sont les communes qui ont le plus contribué à la formation du premier axe à l'aide de la fonction `which()`. La première ligne donne le code Insee des communes, et la seconde restitue leurs numéros de ligne dans le tableau de données initial. Attention, la fonction `which()` classe les individus selon l'ordre des lignes et non des valeurs des contributions.

```
contribIndiv <- contribPca$row.abs
contribIndivAxis1 <- as.vector(contribIndiv[ , 1])
names(contribIndivAxis1) <- row.names(contribIndiv)
head(sort(contribIndivAxis1, decreasing = TRUE))

## 93014 93079 93027 93072 93066 93008
##   421   395   386   301   299   292
```

On note que les communes ayant le plus contribué à la formation de l'axe 1, axe distinguant globalement les communes défavorisés (chômage, revenu médian faible, etc.) et les favorisés (cadres, revenu médian élevé), sont les communes les plus paupérisées du département de Seine Saint-Denis, en premier lieu Clichy-sous-Bois et Villetaneuse.

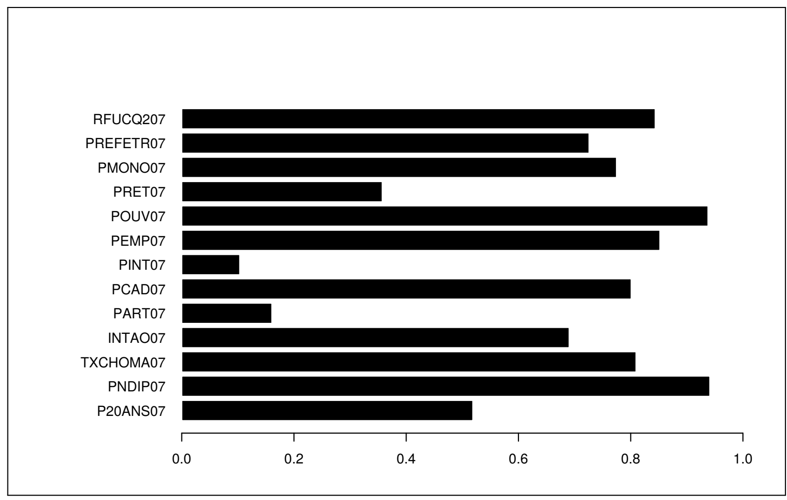
6.1.4 Analyse des qualités de représentation

La qualité de la représentation d'une variable par un axe été calculée dans la section précédente par la fonction `inertia.dudi()`. Elle renseigne sur la part de la variable qu'explique l'axe associé et correspond au carré de sa coordonnée sur l'axe, donc du coefficient de corrélation. La somme des carrés des coordonnées d'une variable sur l'ensemble des axes est égale à 1. La fonction utilisée l'exprime en 1/10 000 et conserve le signe de la contribution.

```
qualVar <- contribPca$col.rel
```

Pour faciliter l'analyse des qualités de représentation des variables, une représentation graphique peut être utile : il s'agit d'un diagramme en bâtons horizontal pour représenter les variables sur l'axe 1, seules les valeurs absolues sont conservées et celles-ci sont divisées par 10 000 pour les situer entre 0 et 1.

```
barplot(abs(qualVar[ , 1] / 10000),
        col = "black",
        border = "white",
        horiz = TRUE,
        las = 1,
        xlim = c(0, 1))
```



On remarque que les variables **RFUCQ207**, **POUV07**, **PEMP07**, **TXCHOMA07**, **PNDIP07** sont particulièrement bien représentées par le premier facteur, puisque ce dernier rend compte de plus de 80% de la dispersion autour de ces six variables.

La qualité de la représentation d'un individu sur les axes factoriels est elle aussi calculée par la fonction `inerti.dudi()`. Elle mesure la proximité d'un individu à un axe. L'analyse sur le premier axe factoriel rend particulièrement bien compte de la particularité de certaines communes du département de la Seine-Saint-Denis (93) et du département du Val de Marne (94).

6.2 Analyse factorielle des correspondances

L'Analyse Factorielle des Correspondances (AFC, *correspondence analysis* en anglais) est une analyse en composantes multiples (ACP) appliquée à un tableau de contingence. Elle est appliquée ici pour mesurer les proximités statistiques entre les communes de la petite couronne parisienne et les arrondissements de la capitale au regard de l'évolution de leurs populations entre 1936 et 2008. La manipulation des fonctions et la façon d'interpréter les sorties sont très semblables dans le cas de l'AFC

que dans le cas de l'ACP. C'est pourquoi l'analyse présentée dans cette section est moins détaillée que la précédente.

L'analyse porte sur les données contenues dans le tableau **popCom3608**. Celui-ci dénombre les individus (ici la population résidente) correspondant au croisement entre deux modalités de variables qualitatives ; l'une définie en lignes (ici la commune) et l'autre en colonnes (ici les années d'observation, considérées comme les modalités d'une même variable de population).

6.2.1 Réaliser l'analyse

En explorant les variables de population à toutes les dates avec les outils présentés dans les chapitres précédents, on remarque que la population moyenne des communes de l'agglomération parisienne n'a pas augmenté de façon linéaire entre 1936 et 2008. Trois phases peuvent être définies : une phase de croissance allant de 1936 à 1968, puis une phase de décroissance entre 1968 et 1982, et enfin une nouvelle phase de croissance entre 1982 et 2008. En outre, alors que la population la plus faible observée a eu tendance à augmenter au cours du temps, passant de 276 habitants en 1936 à 1680 habitants en 2008, le nombre d'habitants de la commune la plus peuplée a décru, passant de 258 599 habitants en 1936 à 234 091 habitants en 2008.

La construction d'une AFC va permettre de caractériser plus en détails les dynamiques démographiques des communes. En effet, en tant que méthode d'analyse de données multivariées, l'AFC donne lieu à une hiérarchisation de l'information contenue dans le tableau de contingence initial. Elle permet de mesurer les proximités statistiques entre modalités d'une variable et individus, ainsi que de comparer les profils-lignes et profils-colonnes entre eux. Il faut noter que les résultats de l'AFC ne sont pas dépendants de l'analyse de l'un ou de l'autre, les transformations opérées sur les individus et les variables du tableau étant parfaitement symétriques.

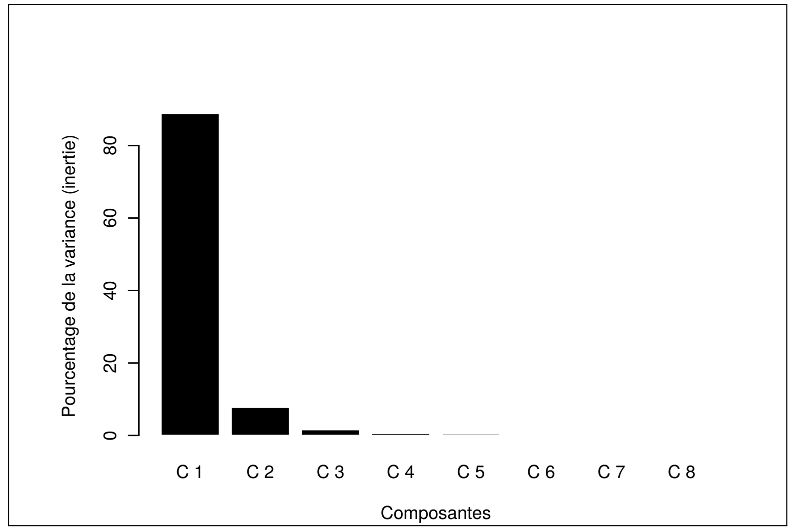
L'analyse est effectuée avec la fonction `dudi.coa()` du *package* `ade4`. Comme dans l'ACP, l'objet créé par la fonction contient tous les éléments nécessaires à la visualisation des résultats et à leur interprétation.

```
row.names(popCom3608) <- popCom3608$CODGEO
coaPop <- dudi.coa(popCom3608[ , 3:11],
                   scannf = FALSE,
                   nf = 2)
```

6.2.2 Interpréter les résultats

Tout comme dans le cas d'une ACP, on observe les valeurs propres associées aux différents axes factoriels afin de déterminer lesquels d'entre eux vont être analysés. Le graphique des valeurs propres apporte une aide visuelle à l'interprétation.

```
summaryCoa <- data.frame(
  EIG = coaPop$eig,
  PCTVAR = 100 * coaPop$eig / sum(coaPop$eig),
  CUMPCTVAR = cumsum(100 * coaPop$eig / sum(coaPop$eig))
)

barplot(summaryCoa$PCTVAR,
        xlab = "Composantes",
        ylab = "Pourcentage de la variance (inertie)",
        names = paste("C", seq(1, nrow(summaryCoa), 1)),
        col = "black",
        border = "white")
```



L'AFC rend compte d'une structuration particulièrement marquée des données. En effet, le premier axe factoriel explique la plus grande part de la variance totale contenue dans le tableau de contingence (88,9 %). Le second facteur résume 7,8 % de l'inertie totale du nuage de points, ainsi 96,68% de l'information sont concentrés sur les deux premiers axes.

Pour aller plus loin dans l'analyse des résultats de l'AFC, il faut étudier plus en détails les coordonnées des individus et des variables sur les axes factoriels. Les coordonnées des individus et des variables sur les axes factoriels sont mesurées par les valeurs de leurs projections sur l'axe et rendent compte des positions relatives des individus et des variables les uns par rapport aux autres et par rapport au centre de gravité du nuage (*i.e.* le profil ligne moyen). La moyenne des valeurs des projections des individus sur l'axe est nulle et la variance associée à ces coordonnées équivaut à la valeur propre associée au facteur. En règle générale, le premier axe factoriel exprime une opposition entre groupes d'individus et/ou variables (coordonnées positives *versus* neutres *versus* négatives), alors que les axes suivants définissent des différenciations internes à ces groupes.

Afin de faciliter l'interprétation visuelle des projections des deux variables, il est préférable de ne pas utiliser le graphique par défaut du *package* `ade4` mais de représenter avec la fonction générique `plot()` les

coordonnées des communes sur les deux premiers axes (cercles gris, arguments pch et col). Les projections des années d'observation sur le plan sont également ajoutées avec la fonction points() et la fonction text() par des losanges noirs reliés les uns aux autres (arguments col, pch et type). L'argument pos précise la position du label (1-Bas ; 2-Gauche ; 3-Haut ; 4-Droite) : ici on donne à cet argument un vecteur de longueur 9 pour préciser la position de chaque label et éviter les superpositions.

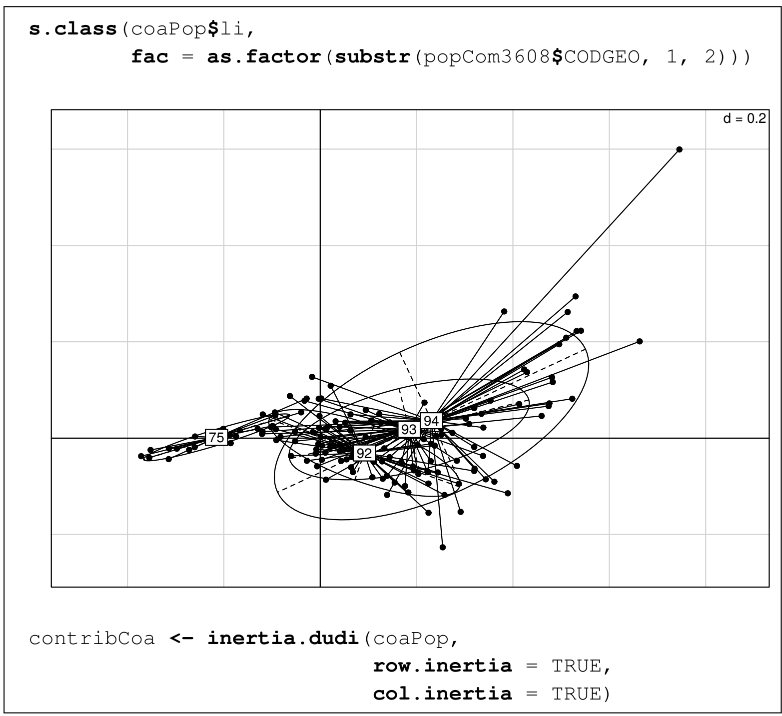
```
plot(coaPop$li, pch = 20, col = "grey40")
abline(h=0, v=0)
points(coaPop$co, type = "o", pch = 18, col = "black")
text(coaPop$co,
     labels = row.names(coaPop$co),
     cex = 0.8,
     pos = c(rep(4, times = 3), 1, rep(4, times = 4), 3))
```

L'observation du premier plan factoriel indique que le premier axe oppose les communes ayant connu une croissance relative de leur population plus forte en début de période à celles dont cette croissance relative de la

population a été plus élevée en fin de période. Le second axe factoriel oppose quant à lui, et ce assez logiquement, les communes ayant connu une croissance relative plus importante en milieu de période aux autres. Au centre du plan, on retrouve des communes affichant un profil de croissance de population relativement moyen, ou ayant connu de nombreuses bifurcations dans la croissance de leur population au cours de la période.

La position des communes sur les deux axes pourra être étudiée plus en détail par l'analyse de leurs coordonnées sur les deux facteurs, la visualisation des ellipses peut également être utile. Comme pour l'ACP, il est intéressant d'examiner les contributions, ce qui se fait, comme dans la section précédente, avec la fonction `inertia.dudi()`.

```
s.class(coaPop$li,
        fac = as.factor(substr(popCom3608$CODGEO, 1, 2)))
```




```
contribCoa <- inertia.dudi(coaPop,
                           row.inertia = TRUE,
                           col.inertia = TRUE)
```

Les dynamiques de peuplement dans l'agglomération parisienne sont typiques d'un modèle monocentrique : la temporalité de l'étalement dépend très directement de la distance au centre, ou plus exactement de l'ac-

cès au centre par les réseaux de transport. Les communes qui se sont peuplées dans le début de période, des années 1930 aux années 1960 sont principalement les arrondissements parisiens. En banlieue, ce sont d'abord les communes les plus proches de Paris et les mieux desservies qui ont vu leur population augmenter, c'est-à-dire principalement les communes des Hauts-de-Seine. Finalement les communes les plus lointaines se sont peuplées sur la dernière période, en particulier dans les département de Seine Saint-Denis et du Val-de-Marne.

Cette analyse factorielle pourrait utilement être suivie d'une procédure de classification qui viserait à dresser des profils de communes en fonction de l'évolution de leur population. Cette analyse est présentée à la Section 7.3.

CHAPITRE 7

Méthodes de classification

Objectifs : *Ce chapitre présente des méthodes de classification automatique utiles pour créer des groupes d'individus statistiques homogènes. Il se concentre sur la classification ascendante hiérarchique, simple puis couplée à une analyse factorielle.*

❊

Prérequis Notions théoriques sur l'analyse géométrique des données : classification ascendante hiérarchique et analyses factorielles présentées dans le chapitre précédent.

Description des *packages* utilisés Ce chapitre utilise le *package* `cluster` pour la classification, le *package* `ade4` pour produire l'analyse factorielle, le *package* `FactoClass` pour faire une classification prenant en compte une variable de pondération. Il utilise également les *packages* `ggplot2` et `scales` pour les représentations graphiques.

7.1 Rappels théoriques

Les algorithmes de classification visent à constituer des groupes différenciés d'individus homogènes au regard de leurs attributs statistiques. Les groupes sont homogènes si les individus statistiques qui les constituent sont les plus semblables possibles au sein de chaque groupe, ils sont différenciés si les individus de groupes différents sont aussi dissemblables que possible.

Le schéma suivant montre un exemple simple comprenant 12 individus statistiques décrits par 2 variables. L'inertie totale du nuage de points est la somme du carré des distances de chaque point au centre. Cette inertie totale peut être décomposée en une inertie intergroupe et une inertie intragroupe : la classification aura pour objectif de minimiser l'inertie intragroupe (homogénéité au sein du groupe) et de maximiser l'inertie intergroupe (différentiation entre groupes).

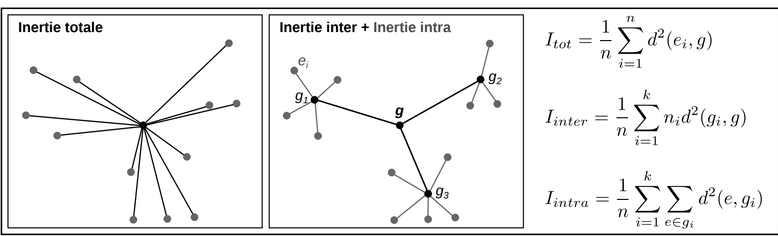


FIGURE 7.1 – Classification et décomposition de l'inertie

Les deux méthodes de classification les plus classiques sont le partitionnement direct de l'ensemble d'individus et la classification ascendante hiérarchique. La première méthode consiste à assigner un nombre de centres fixé *a priori* puis à localiser et re-localiser les centres ou affecter et ré-affecter les individus à ces centres par un processus itératif qui

optimise un critère de qualité de la partition. Ces algorithmes de « réallocation dynamique » (centres mobiles, nuées dynamiques) sont rapides et peuvent être appliqués sur des données massives.

La classification ascendante hiérarchique (CAH) est également un algorithme itératif. Il part d'une partition en n classes, chaque classe étant constituée d'un seul individu. À chaque étape, il agrège les deux classes les plus proches. Ce processus est répété $n-1$ fois, il produit ainsi une série hiérarchique de $n-1$ partitions, de la plus fine (n classes) à la plus grossière (une classe). Cette série de partitions est représentée par un arbre hiérarchique ou dendrogramme. À la différence des méthodes de partitionnement direct, la CAH permet à l'utilisateur de choisir le nombre de classes *a posteriori*, à partir de mesures et de graphiques d'aide à l'interprétation. Cette caractéristique explique l'usage fréquent de cette méthode en sciences sociales, cependant elle est beaucoup plus gourmande en calcul et ne peut être appliquée à de gros jeux de données.

Ce chapitre se concentre sur la méthode de classification ascendante hiérarchique parce qu'elle est très souvent utilisée en géographie. Les paramètres à définir sont la définition d'une distance entre individus et d'une distance entre classes. Ce type de distance entre classes est aussi appelé « critère d'agrégation ».

La distance $d(i, i')$ est la distance entre deux individus i et i'. La classification s'appuie sur une matrice de distances entre individus que l'utilisateur doit définir : distance euclidienne, distance du χ^2 ou autre. Dans les fonctions présentées par la suite, l'utilisateur peut fournir un tableau d'individus décrits par des variables quantitatives, il peut aussi fournir une matrice de distances calculée au prélable, par exemple avec la fonction `dist()`. Pour pouvoir raisonner en termes d'inertie inter- et intraclasse, il faut travailler sur des distances euclidiennes au carré (cf. Figure 7.1).

Le critère d'agrégation $\delta(k, k')$ définit la façon de mesurer la distance entre les classes k et k'. Plusieurs critères peuvent être utilisés :

- distance minimum,
- distance maximum,
- distance moyenne,
- distance entre centres de gravité,
- moyenne des distances à l'intérieur des classes,

— perte d'inertie minimum (critère de Ward), fondée sur la décomposition de l'inertie totale en inertie intra- et interclasses.

La Figure 7.2 montre trois de ces critères d'agrégation. Cet exemple simple ne comporte que deux variables pour permettre de bien visualiser le travail sur les distances. Au-delà de trois dimensions les distances ne sont plus visualisables, mais elles restent calculables et le principe d'agrégation reste le même.

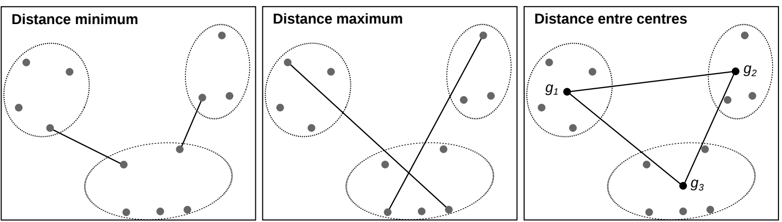


FIGURE 7.2 – Exemples de critères d'agrégation

Chacun des critères d'agrégation a ses inconvénients, certains ont tendance à créer des classes de tailles très hétérogènes, d'autres ont tendance à systématiquement créer des groupes de même taille, certains sont très sensibles aux valeurs extrèmes. Dans tous les cas, l'utilisateur doit prendre toutes les précautions nécessaires pour arriver à un résultat cohérent : faire des tests en faisant varier les paramètres, bien utiliser les aides à l'interprétation pour choisir le nombre de classes, examiner la classification obtenue.

Pour déterminer le nombre de classes optimal, l'arbre hiérarchique, ou dendrogramme, est un outil visuel efficace. Les feuilles de l'arbre sont les n classes formées chacune d'un seul individu avant la première étape d'agrégation. À chaque étape, elles sont regroupées selon le critère d'agrégation choisi et la position sur l'axe vertical du nœud formé à chaque regroupement donne une mesure de la dissimilarité entre classes.

Pour la lecture des partitions, on peut imaginer que l'on parcourt l'arbre du bas vers le haut avec une règle horizontale. La règle coupe l'arbre en 2. La partie inférieure donne les classes associées (ensemble des nœuds directement inférieurs). L'axe vertical marque les valeurs du critère d'agrégation (mesure de dissimilarité) à chaque étape de l'algorithme. Ainsi, la

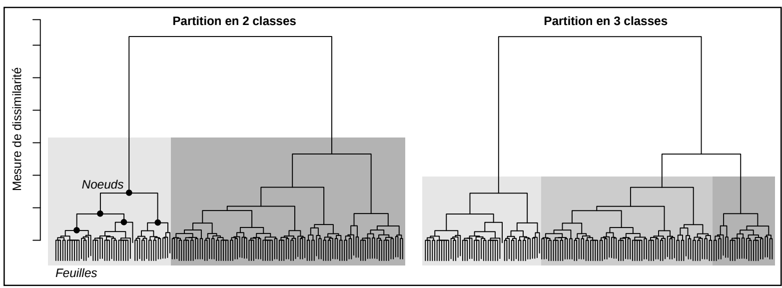


FIGURE 7.3 – Lecture du dendrogramme

compacité des classes créées se lit sur cet axe en examinant la longueur des branches associées aux nœuds inférieurs.

Au-delà de la lecture de l'arbre, il existe un grand nombre de graphiques d'aide à l'interprétation et d'indicateurs du nombre optimal de classes à adopter. Dans tous les cas, il ne faut jamais considérer qu'il existe une unique partition possible. Pour construire une classification robuste et en comprendre le résultat, il faut systématiquement examiner les classes, leur effectif, leur compacité (distance entre les individus regroupés), leur séparation des autres classes. Il est aussi utile d'examiner les individus exceptionnels, ceux qui sont très proches du centre de classe et ceux qui en sont très éloignés. Il convient enfin d'examiner au sein de chaque classe la distribution des variables utilisées pour la classification.

Pour s'assurer de la robustesse d'une classification, il existe des méthodes plus avancées : les méthodes mixtes, qui combinent la classification hiérarchique avec le partitionnement direct [1] ; le couplage de la classification avec l'analyse factorielle. Ce procédé est utile dans le cas où les variables utilisées pour la classification sont corrélées les unes avec les autres.

La suite du chapitre présente deux applications : la classification d'individus décrits par des variables quantitatives et la classification de lignes

1. Une stratégie mixte consisterait par exemple à créer une première partition avec une méthode hiérarchique puis à améliorer la partition obtenue en réaffectant les individus avec une méthode de moyennes mobiles appliquée sur les centres de classes produits par la CAH. La fonction `kmeans()` autorise ce traitement en permettant de fixer les centres de classes initiaux.

d'un tableau de contingence. La première utilisera la distance euclidienne passée au carré, la seconde est couplée à une analyse factorielle des correspondances et travaille sur une matrice de distances du χ^2.

7.2 Classification ascendante hiérarchique

Il s'agit dans cette section de classer les communes en fonction de leur composition en catégories professionnelles (CSP). Cette classification est faite sur les variables qui renseignent les proportions de ces catégories en 2007 : proportions d'artisans, de cadres, de professions intermédiaires, d'employés et d'ouvriers. Les ordres de grandeur de ces variables étant différents, elles doivent être centrées-réduites de manière à donner à chacune le même poids dans l'analyse.

```
selecVar <- c("PART07","PCAD07","PINT07","PEMP07","POUV07")
muniCSP <- socEco9907[, selecVar]
muniCSP <- scale(muniCSP)
```

La classification se fait avec la fonction `agnes()` (**ag**glomerative **nes**ting) du *package* `cluster`. La fonction permet de spécifier la distance entre individus, ici la distance euclidienne, et la distance entre classes, ici le critère de Ward précédemment décrit.

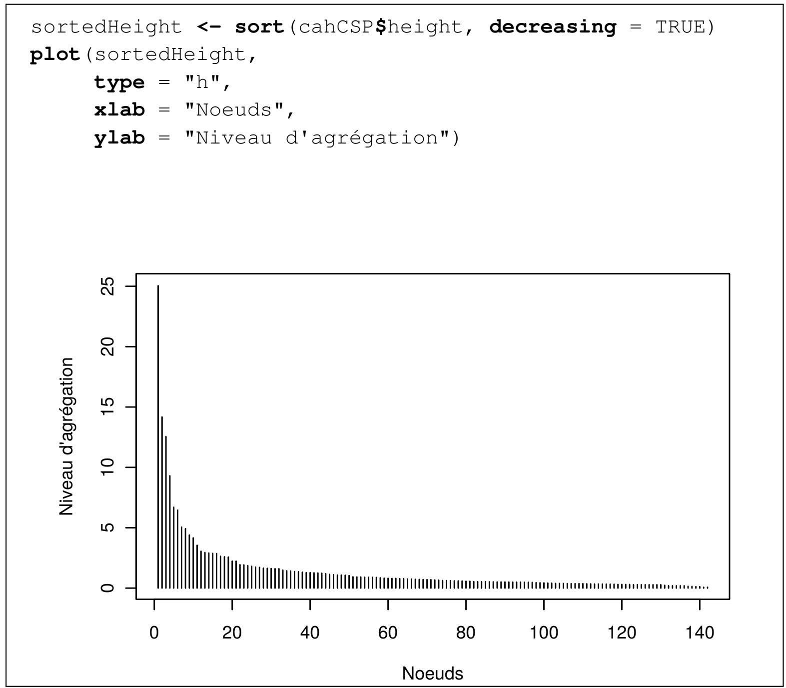
```
cahCSP <- agnes(muniCSP,
                metric = "euclidean",
                method = "ward")
```

L'objet **cahCSP** est de type *agnes*. Il contient l'ensemble des informations qui vont ensuite servir pour dessiner et analyser l'arbre, puis le couper et donner des aides à l'interprétation. Le plus simple de ces graphiques est le diagramme de niveaux : il représente la valeur de la mesure de dissimilarité associée à la création de chacun des nœuds. Cette valeur dépend du critère d'agrégation choisi, dans le cas présent, il s'agit de la part de l'inertie interclasse qui, à chaque étape d'agrégation, passe en inertie intraclasse. Au début du processus, chaque classe est composée d'un seul individu, l'inertie totale est donc égale à l'inertie interclasse. À la fin du

processus, il n'y a plus qu'une seule classe, l'inertie totale est donc égale à l'inertie intraclasse.

On accède aux niveaux par l'attribut `$height` de l'objet. Ceux-ci peuvent être triés par ordre décroissant puisque, par définition, l'algorithme agrège à chaque étape les classes les moins dissemblables.

```
sortedHeight <- sort(cahCSP$height, decreasing = TRUE)
plot(sortedHeight,
     type = "h",
     xlab = "Noeuds",
     ylab = "Niveau d'agrégation")
```



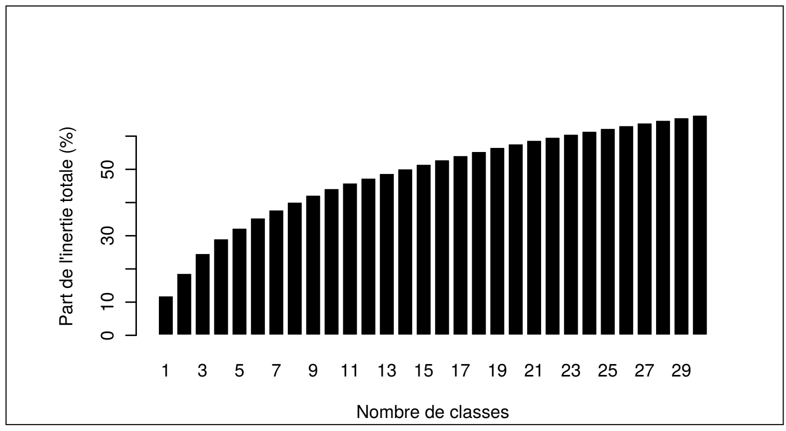
Un tel graphique permet d'ores et déjà de repérer les « sauts » et de décider à quel niveau on peut couper l'arbre. Les niveaux sont exprimés ici en absolu et, lorsque l'on utilise le critère d'agrégation de Ward, il est d'usage de les exprimer en proportion de l'inertie totale : chaque niveau est donc rapporté à la somme des niveaux. En cumulant ce taux de proche en proche, on obtient la part de l'inertie restituée par la partition correspondante. Représenter le haut de la hiérarchie est suffisant, ici les 30 derniers nœuds.

Attention, pour un telle interprétation, la distance entre classes utilisée par le critère d'agrégation de Ward doit être une distance euclidienne pas-

sée au carré. Comme mentionné précédemment, la fonction `agnes()` accepte deux types d'objets : un tableau d'individus caractérisés par des variables ou une matrice de distances calculée au préalable. Si l'utilisateur décide de calculer la matrice de distance, avec la fonction `dist()` par exemple, il faut qu'il la passe au carré avant de faire la classification. En revanche, si l'utilisateur donne comme argument un tableau individus-variables et paramètre la fonction comme fait plus haut (distance euclidienne et critère de Ward), la fonction travaille automatiquement sur des distances euclidiennes au carré.

```
relHeight <- sortedHeight / sum(sortedHeight) * 100
cumHeight <- cumsum(relHeight)

barplot(relHeight[1:30], names.arg = seq(1, 30, 1),
        col = "black", border = "white", xlab = "Noeuds",
        ylab = "Part de l'inertie totale (%)")
```

```
barplot(cumHeight[1:30], names.arg = seq(1, 30, 1),
        col = "black", border = "white",
        xlab = "Nombre de classes",
        ylab = "Part de l'inertie totale (%)")
```

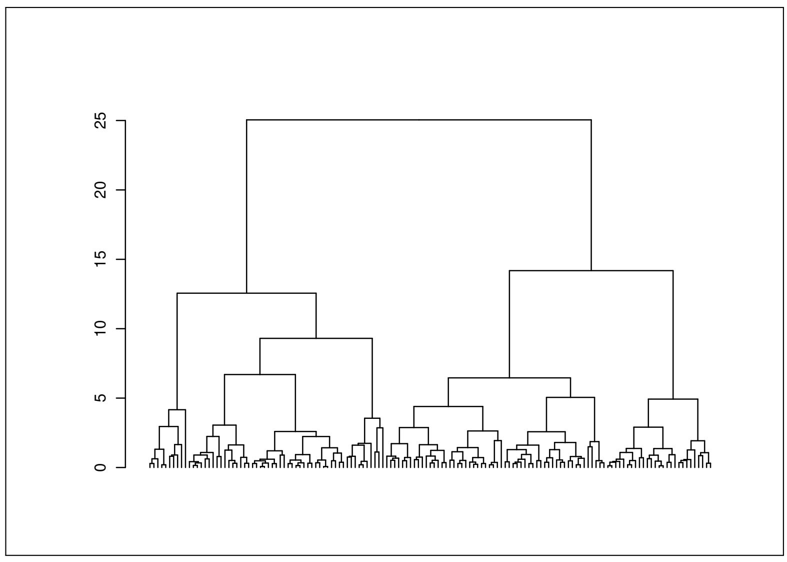


On lit ainsi à la fois les sauts (graphique du haut) et la part de l'inertie interclasse par rapport à l'inertie totale associée à la partition (graphique du bas). Une coupure en 2, en 4 ou en 5 classes semble judicieuse.

7.2.1 Coupure de l'arbre et description des classes

Il y a plusieurs façons d'afficher l'arbre hiérarchique, la première consiste à utiliser la fonction `pltree()` du *package* `cluster` directement sur l'objet créé par la classification. Pour plus de réglages graphiques, il est utile de transformer au préalable cet objet de type `agnes` en objet de type `dendrogram` puis d'appliquer à cet objet la fonction générique `plot()`.

```
dendroCSP <- as.dendrogram(cahCSP)
plot(dendroCSP, leaflab = "none")
```



La visualisation de l'arbre confirme les remarques précédentes, une partition en quatre classes semble pertinente. Pour appuyer cette décision, un grand nombre d'indicateurs existent dont les principaux sont regroupés dans le *package* `NbClust`. Celui-ci contient une seule fonction qui calcule trente indicateurs de ce type et donne le nombre de partitions optimal selon chacun d'eux.

Pour découper l'arbre et obtenir la partition voulue, c'est la fonction `cutree()` qui est utilisée. Celle-ci crée une nouvelle variable donnant pour chaque entité son numéro de classe. Cette variable peut ensuite être intégrée au tableau initial pour caractériser les classes au regard des variables utilisées pour la classification.

```
clusCSP <- cutree(cahCSP, k = 4)
muniCSP <- as.data.frame(muniCSP)
muniCSP$CLUSCSP <- factor(clusCSP,
                          levels = 1:4,
                          labels = paste("CLUS", 1:4))
```

L'analyse la plus simple consiste à visualiser la moyenne de classe de chacune des variables. Ce travail peut être mené sur les variables originelles, les proportions de chacune des catégories socio-professionnelles,

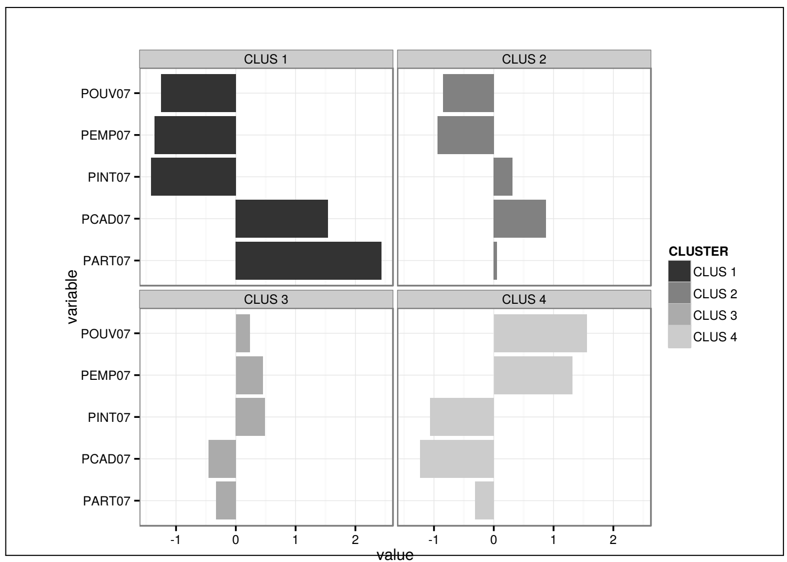
mais aussi sur les variables centrées-réduites. La première option a l'avantage de l'unité de mesure (%) qui est directement interprétable. La seconde option permet une comparaison directe des profils à partir de variables aux ordres de grandeur très différents. C'est cette seconde option qui est choisie ici.

Dans un premier temps, la moyenne des variables est calculée avec le fonction `aggregate()` sur chacun des quatre groupes créés par la classification. On cherche ensuite à faire un diagramme en bâtons (*barplot*) pour représenter graphiquement chacun des groupes. Ceci pourrait être fait avec une boucle qui afficherait successivement les graphiques. Il faudrait également modifier les paramètres graphiques pour que chaque graphique s'intègre dans la même sortie. Dans l'exemple suivant, cette représentation est produite avec le *package* `ggplot2` présenté dans le Chapitre 9. Ce *package* demande un tableau en format long (cf. Section 2.4.3), transformation effectuée avec la fonction `melt()`.

```
clusProfile <- aggregate(muniCSP[, 1:5],
                         by = list(muniCSP$CLUSCSP),
                         mean)

colnames(clusProfile)[1] <- "CLUSTER"
clusLong <- melt(clusProfile, id.vars = "CLUSTER")
```

Le graphique prend comme argument le tableau qui vient d'être transformé, les trois variables sont l'identification des classes (**CLUSTER**), l'identification des variables (variable) et les valeurs prises par les variables pour chaque classe (value). Pour construire une planche de graphiques, il faut inclure dans une même sortie graphique un ensemble de graphiques correspondant à des sous-groupes. C'est la fonction `facet_wrap()` qui permet cette mise en forme (cf. Section 9.3.4).

```
ggplot(clusLong) +
  geom_bar(aes(x = variable, y = value, fill = CLUSTER),
           stat = "identity") +
  scale_fill_grey() +
  facet_wrap(~ CLUSTER) +
  coord_flip() + theme_bw()
```



Ces profils peuvent être cartographiés [1] avec les méthodes présentées dans les chapitres 9 et 10. L'analyse conjointe de la carte et du graphique montre un Ouest parisien accompagné de quelques communes des Hauts-de-Seine (cluster 1) avec une forte sur-représentation des cadres et des artisans-commerçants. À l'opposé, le Nord de la zone d'étude, correspondant à la moitié Est de la Seine-Saint-Denis (cluster 4), montre une forte sur-représentation d'ouvriers et d'employés. Entre ces deux extrêmes, les profils 2 et 3 sont comparables aux profils 1 et 4 respectivement, mais avec des tendances moins marquées.

Certains cas particuliers pourraient être examinés par une analyse plus fine des profils de classes les comparant au profil moyen de l'ensemble de l'espace d'étude. Il s'agirait de tester, pour chaque variable, si la moyenne de la classe est significativement différente de la moyenne générale. Tous ces calculs pourraient être faits avec les fonctions présentées dans les chapitres 4 et 5. Il existe aussi une fonction tout-en-un, la fonction `catdes()` du *package* `FactoMineR`.

1. Attention, la carte en noir et blanc n'est pas satisfaisante d'un point de vue sémiologique, il serait plus pertinent d'utiliser ici une palette qualitative. Voir à ce propos le Chapitre 9 et en particulier la Section 9.2.

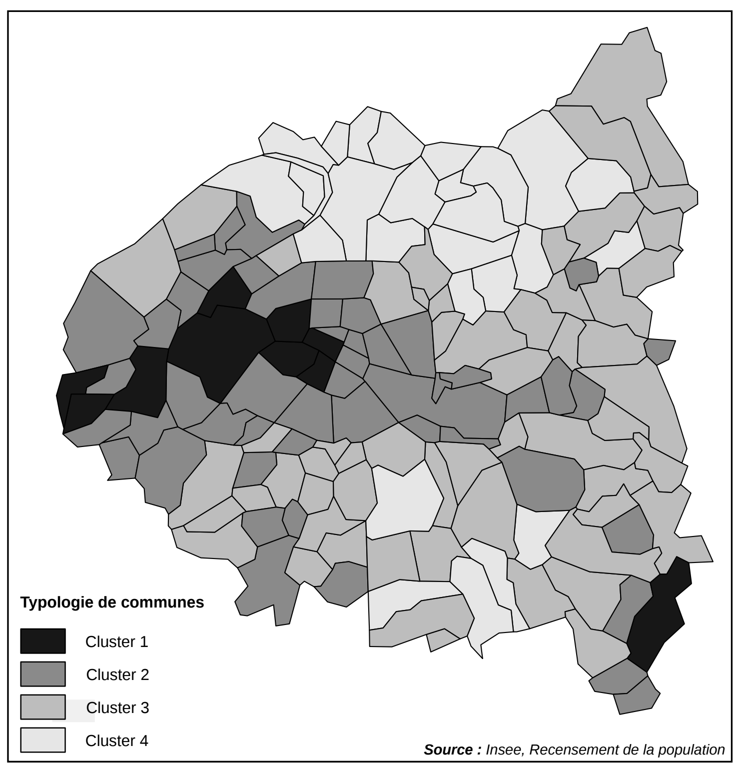


FIGURE 7.4 – Typologie de communes selon les CSP

Cet exemple simple de classification montre aussi l'intérêt de coupler les méthodes, en particulier l'analyse factorielle et la classification. Il arrive en effet souvent que les variables d'intérêt soient corrélées les unes avec les autres, cas systématique lorsque les variables sont des pourcentages qui somment 100 % pour chaque individu statistique. Ici, les communes avec une forte proportion de cadres et d'artisans-commerçants auront mécaniquement une faible proportion des autres catégories. Dans un tel cas, il est judicieux d'effectuer d'abord une analyse factorielle sur les variables d'intérêt, analyse qui élimine les problèmes de colinéarité, puis de faire une classification sur les axes factoriels.

7.3 Couplage analyse factorielle - classification

7.3.1 Classification sur une métrique du χ^2

Il s'agit ici de dresser des profils de communes en fonction de l'évolution de leur population entre 1936 et 2008. Cette classification constitue la suite logique de l'analyse factorielle des correspondances présentée dans la Section 6.2. Elle débute donc de la même façon par une analyse factorielle sur un tableau de contingence. Comme dans la Section 6.2, le tableau de contingence est le résultat du croisement de deux variables qualitatives, ici les communes (lignes du tableau) et les années de recensement (colonnes).

Dans une telle analyse menée sur les modalités de variables qualitatives, il est d'usage de travailler non pas sur une distance euclidienne mais sur une distance du χ^2. En effet, celle-ci est moins sensible au découpage en modalités du phénomène observé, c'est-à-dire à la définition de typologies ou nomenclatures (propriété d'équivalence distributionnelle). Pour un tableau de n lignes et p colonnes, la distance entre les lignes i et i' s'écrit de la façon suivante :

$$d^2(i,i') = \sum_{j=1}^{p} \left(\frac{1}{f_{\cdot j}}\right) \left(\frac{f_{ij}}{f_{i\cdot}} - \frac{f_{i'j}}{f_{i'\cdot}}\right)^2$$

À la différence d'une classification effectuée sur un tableau d'individus caractérisés par des variables, la classification des lignes ou des colonnes du tableau de contingence travaille sur des effectifs. Il est donc utile de prendre ce fait en compte en pondérant les lignes par l'effectif. Ce type de classification sur des distances du χ^2 prenant en compte une variable de pondération est implémenté dans les logiciels classiques comme SAS ou SPAD, mais pas dans les fonctions R utilisées jusqu'à présent. C'est pour cette raison que ce calcul est effectué avec la fonction `ward.cluster()` du *package* `FactoClass`.

La première étape consiste à faire l'analyse factorielle telle que présentée à la Section 6.2.

```
popEvol <- popCom3608[ , 3:11]
row.names(popEvol) <- popCom3608$CODGEO
coaPop <- dudi.coa(popEvol,
                   scannf = FALSE,
                   nf = ncol(popEvol))
```

Ensuite, la matrice de distances du χ^2 est produite grâce à la fonction `dist.dudi()` du *package* `ade4`. L'argument `amongrow` précise si la distance doit être calculée entre les lignes du tableau ou entre ses colonnes. Cette matrice est enfin utilisée par la fonction `ward.cluster()`. L'argument `peso`[1] permet de pondérer les lignes par l'effectif marginal, c'est-à-dire la somme des effectifs de chaque ligne. Cette somme est calculée directement par la fonction `apply()` présentée à la Section 3.4. L'objet qui stocke les résultats de la classification est de type `hclust` et le dendrogramme peut être découpé comme précédemment avec la fonction `cutree()`.

```
library(FactoClass)
chiDist <- dist.dudi(coaPop, amongrow = TRUE)
cahPop <- ward.cluster(chiDist,
                       peso = apply(popEvol, 1, sum),
                       plots = FALSE, h.clust = 1)
popEvol$CLUSTER <- paste("CLUS", cutree(cahPop, k = 4))
```

7.3.2 Description des classes

Le tableau de contingence représente ici les trajectoires des communes en termes de peuplement. La description des classes proposée dans cette section se concentre sur l'analyse des groupes de communes pour lesquelles les trajectoires de peuplement sont semblables. Trois types de tableaux sont créés à partir de résumés numériques calculés sur les classes de communes : un tableau qui donne la somme des populations communales des classes à chaque pas de temps, un tableau qui rapporte cette

1. Parmi les quelques 5 000 *packages* existant à la date de rédaction de ce manuel, un certain nombre d'entre eux sont peu attentifs aux noms d'arguments, mélangeant des noms d'arguments en anglais avec des noms dans la langue des auteurs. Il est donc possible de trouver un argument `peso` (poids) dans un *package* implémenté par des hispanophones, ou un argument `poids` dans un *package* implémenté par des francophones.

somme à la population totale de l'espace d'étude, un tableau qui donne la population communale moyenne pour chaque classe et à chaque pas de temps.

La fonction `aggregate()` est d'abord utilisée pour calculer les résumés numériques selon la variable de regroupement **CLUSTER** (cf. Section 2.4.2). Les effectifs sont divisés par 1000 pour une meilleure lisibilité (population exprimée en milliers). Le calcul des populations en pourcentage du total à chaque date est fait avec la fonction `prop.table()` présentée précédemment (cf. Section 5.4.2). Cette fonction demande une transformation du tableau en objet de type `matrix`, puis une transformation du résultat en objet de type `data.frame`. Le calcul des populations moyennes se fait comme le premier avec `aggregate()` mais en utilisant la fonction `mean()` au lieu de `sum()`.

```
# Somme des populations (absolue)
sumClass <- aggregate(popEvol[ , 1:9],
                      by = list(popEvol$CLUSTER),
                      FUN = sum)
sumClass[ , 2:10] <- sumClass[ , 2:10] / 1000
colnames(sumClass)[1] <- "CLUSTER"

# Somme des populations (relative)
sumClassMat <- as.matrix(sumClass[ , 2:10])
propClass <- as.data.frame(prop.table(sumClassMat, 2))
propClass$CLUSTER <- sumClass$CLUSTER

# Moyenne des populations
meanClass <- aggregate(popEvol[ , 1:9],
                      by = list(popEvol$CLUSTER),
                      FUN = mean)
meanClass[ , 2:10] <- meanClass[ , 2:10] / 1000
colnames(meanClass)[1] <- "CLUSTER"
```

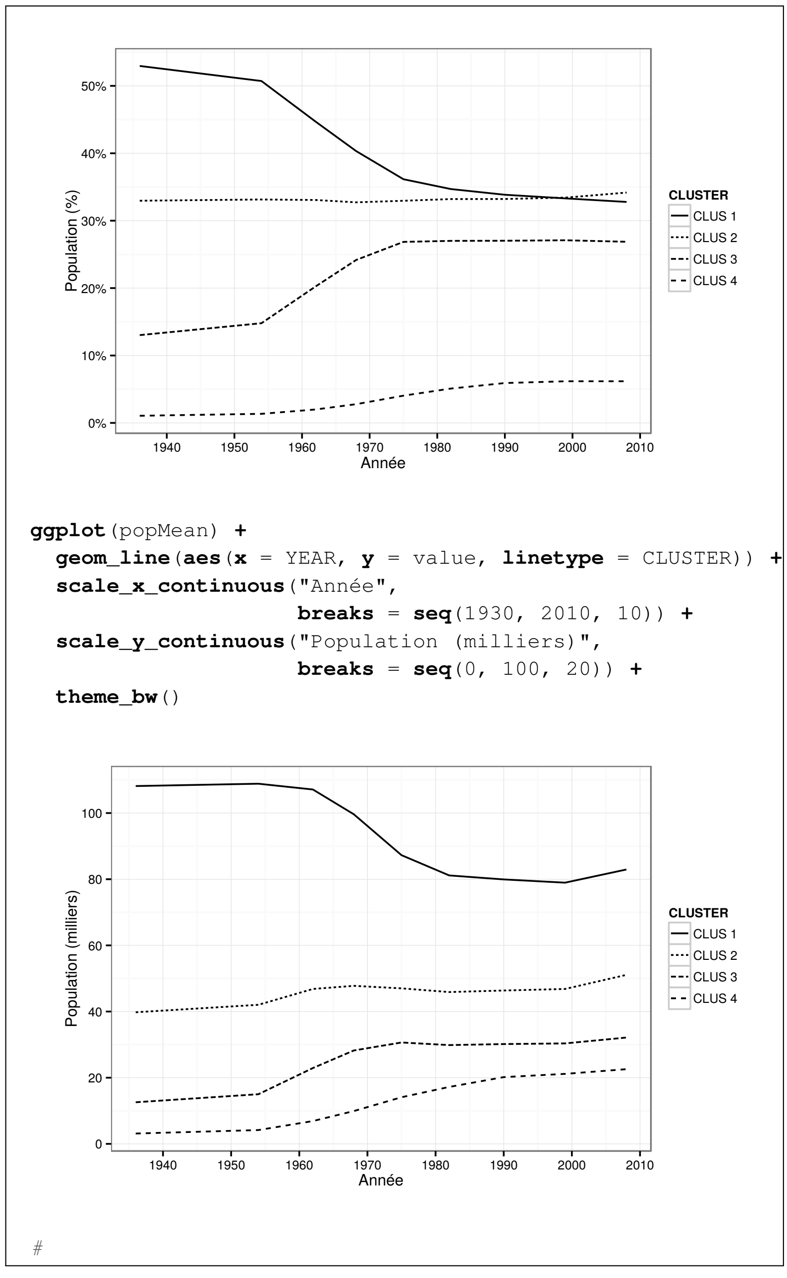
Comme dans la section précédente, il est possible de produire des représentations graphiques avec les fonctions génériques ou avec le *package* `ggplot2`. C'est cette option qui est choisie et les tableaux doivent donc être transformés en format long avec la fonction `melt()`. Ensuite, une variable temporelle est créée à partir des noms de variables (**POP1936**, **POP1954**, etc.) : les quatre derniers caractères sont extraits avec la fonction `substr()` et ils sont transformés en nombres entiers.

```
absSum <- melt(sumClass, id.vars = "CLUSTER")
relSum <- melt(propClass, id.vars = "CLUSTER")
popMean <- melt(meanClass, id.vars = "CLUSTER")
absSum$YEAR <- as.integer(substr(absSum$variable, 4, 7))
relSum$YEAR <- as.integer(substr(relSum$variable, 4, 7))
popMean$YEAR <- as.integer(substr(popMean$variable, 4, 7))
```

Ces tableaux servent à produire les trois graphiques : somme des populations (valeur absolue), somme des populations (valeur relative) et moyenne des populations par classes.

```
ggplot(absSum) +
  geom_line(aes(x = YEAR, y = value, linetype = CLUSTER)) +
  scale_x_continuous("Année",
                     breaks = seq(1930, 2010, 10)) +
  scale_y_continuous("Population (milliers)",
                     breaks = seq(0, 2500, 500)) +
  theme_bw()
```

```
ggplot(relSum) +
  geom_line(aes(x = YEAR, y = value, linetype = CLUSTER)) +
  scale_x_continuous("Année",
                     breaks = seq(1930, 2010, 10)) +
  scale_y_continuous("Population (%)",
                     labels = percent) +
  theme_bw()
```




```
ggplot(popMean) +
  geom_line(aes(x = YEAR, y = value, linetype = CLUSTER)) +
  scale_x_continuous("Année",
                     breaks = seq(1930, 2010, 10)) +
  scale_y_continuous("Population (milliers)",
                     breaks = seq(0, 100, 20)) +
  theme_bw()
```

#

Ces graphiques renvoient une image assez claire des dynamiques de peuplement en distinguant ces quatre classes de communes : la première classe se caractérise par une forte diminution, particulièrement marquée entre les années 1960 et 1980, les trois autres se caractérisent par une augmentation continue, assez faible pour la classe 2, et forte pour les classes 3 et 4. Une représentation cartographique des classes de communes [1] est utile pour spatialiser ce constat : les quatre classes s'organisent, à peu de choses près, en cercles concentriques autour de Paris.

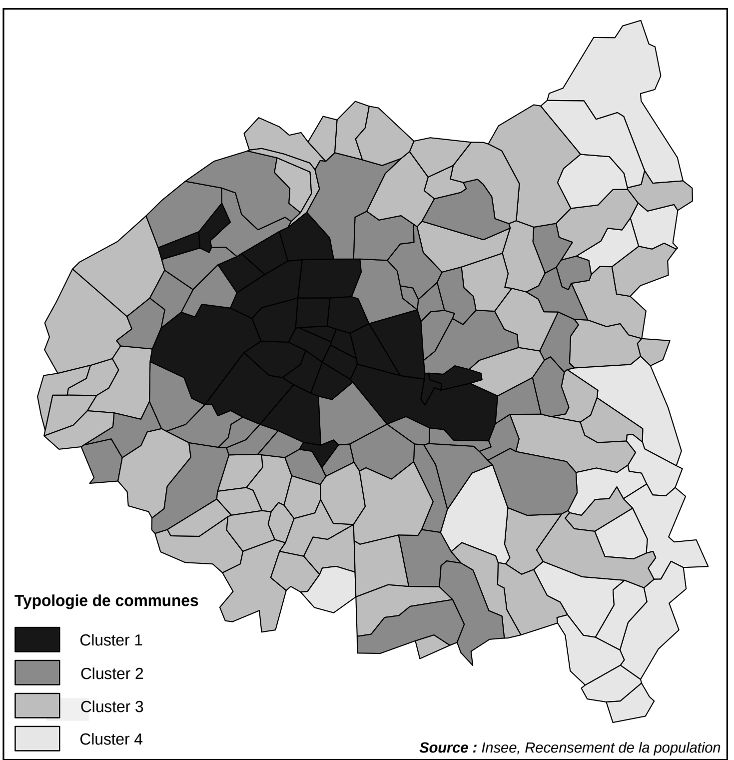


FIGURE 7.5 – Typologie de communes en fonction des dynamiques de peuplement

1. Attention, la carte en noir et blanc n'est pas satisfaisante d'un point de vue sémiologique, il serait plus pertinent d'utiliser ici une palette qualitative. Voir à ce propos le Chapitre 9 et en particulier la Section 9.2.

Ce chapitre a présenté plusieurs techniques de classification permettant de créer des groupes différenciés d'individus homogènes. La classification ascendante hiérarchique est une technique fréquemment utilisée en analyse spatiale, elle peut être mobilisée dans un approche descriptive, exploratoire et/ou modélisatrice.

CHAPITRE 8

Analyse de graphes

Objectifs : *Ce chapitre présente une initiation à l'analyse de graphes avec R : importer les données, les transformer pour créer des graphes, calculer des mesures sur les graphes et les visualiser. Les graphes sont des objets mathématiques composés de sommets (nœuds,* vertices*) et de liens (arcs,* edges*). Selon les disciplines, les graphes peuvent représenter des réseaux sociaux, des réseaux de lieux, des réseaux de transport, etc.*

Prérequis Connaissance des fonctions R de base, du vocabulaire de l'analyse de graphes et des principales mesures calculées sur les graphes.

Description des *packages* utilisés Les deux principaux *packages* d'analyse de graphe sont présentés : `statnet` et `igraph`. Le premier est en réalité un métapackage incluant divers *packages* spécifiques tels `sna`, `network`, `ergm` etc. Cet ensemble est développé par des sociologues, il utilise des objets de type *network*. Le second, `igraph`, est développpé par des physiciens, il utilise des objets de type *igraph*.

Ces deux *packages* sont très utilisés et très bien documentés avec des sites dédiés [1]. Le choix d'utiliser l'un ou l'autre, ou les deux, dépend des besoins et des préférences de l'utilisateur. Toutes les fonctionnalités de base sont implémentées dans les deux *packages*, en revanche certaines fonctionnalités spécifiques de l'analyse de réseaux sociaux (*social network analysis*) ne sont implémentées que dans `statnet` alors que d'autres, plutôt issues de la physique, ne sont implémentées que dans `igraph`.

Il est déconseillé d'utiliser les deux *packages* en même temps. En effet, il y a un certaine redondance entre les deux *packages* qui peut être source d'erreur : certaines fonctions différentes de chacun des deux *packages* ont des noms identiques (cf. Encart p. 43).

8.1 Aperçu et préparation des données

Le fichier `MobResid08.txt` est utilisé ici, il s'agit du fichier de mobilités résidentielles entre communes de la petite couronne produit par l'Insee.

```
mobResid <- read.csv2("data/MobResid08.txt",
                      stringsAsFactors = FALSE)
```

Le fichier de départ liste les flux résidentiels entre les communes de Paris et petite couronne entre 2003 et 2008. Il comprend 5 colonnes et 15 178 lignes, ses variables sont décrites à la Section 1.6. Il faut d'ores et déjà noter que le tableau est en format long : une ligne du tableau est un couple de lieux assorti d'un flux. Avec les techniques présentées à la

1. Le site `http://statnet.csde.washington.edu/index.shtml` pour `statnet` et le site `http://igraph.org` pour `igraph`.

Section 2.4.3, il serait facile de passer à un format large, qui serait ici une matrice origine-destination.

Dans le langage de l'analyse de graphe, le format long est considéré comme une liste de liens (*edges list*) et le format large comme une matrice d'adjacence (*adjacency matrix*). L'exemple ci-dessous montre trois représentations d'un même graphe : une liste de liens, une matrice d'adjacence et une visualisation du graphe correspondant. Ce graphe de trois nœuds a les mêmes caractéristiques que le graphe des communes de l'agglomération parisienne sur lequel porte ce chapitre : c'est un graphe orienté et valué. Il est orienté parce qu'un flux de i vers j n'est pas équivalent à un flux de j vers i, ce qui se traduit par une matrice d'adjacence non symétrique. Il est valué parce que les liens sont assortis d'un attribut numérique qui associe une quantité au lien, par exemple le nombre d'individus qui migrent d'une commune vers une autre.

couple de lieux
edges list

ori	des	flux
A	B	2
A	C	5
B	A	3
B	C	4
C	A	0
C	B	0

matrice origine-destination
adjacency matrix

	A	B	C
A	0	2	5
B	3	0	4
C	0	0	0

visualisation du graphe

FIGURE 8.1 – Trois représentations d'un graphe

Les *packages* utilisés ici permettent de créer un graphe à partir des deux types de tableaux : liste de liens ou matrice d'adjacence. Cependant, avant de faire cette transformation du tableau en graphe, il convient de remarquer que bon nombre de mesures pourraient être calculées directement sur les tableaux, sans passer par la transformation en graphe. L'exemple ci-dessous montre un tel traitement qui vise à calculer un solde résidentiel relatif, mesure qui met en relation les flux entrant et sortant d'une commune.

Dans un premier temps, les flux intracommunaux sont supprimés pour simplifier le traitement et son interprétation [1]. Seuls trois champs sont conservés et réordonnés : la commune d'origine (**DCRAN**), la commune de destination (**CODGEO**) et le flux (**NBFLUX**).

Dans un deuxième temps, le tableau long (liste de liens) est transformé en tableau large (matrice d'adjacence) avec le *package* `reshape2` (cf. Section 2.4.3). Ce tableau est un objet de type *data.frame*, il est transformé en objet de type *matrix* et les codes des communes d'origine sont ajoutés en tant que noms de lignes (avec `row.names()`) et non en tant que variable du tableau. Enfin, les flux entrants (*in*) et sortants (*out*) sont calculés grâce à la fonction `apply()` (cf. Section 3.4).

```
library(reshape2)
edgesList <- mobResid[mobResid$CODGEO != mobResid$DCRAN,
                      c("DCRAN", "CODGEO", "NBFLUX")]

adjMat <- dcast(data = edgesList,
                formula = DCRAN ~ CODGEO,
                value.var = "NBFLUX")

adjMat <- as.matrix(adjMat[ , -1])
row.names(adjMat) <- colnames(adjMat)

flowsOut <- apply(adjMat, 1, sum, na.rm = TRUE)
flowsIn <- apply(adjMat, 2, sum, na.rm = TRUE)
summary(flowsOut)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     168    2370    3850    5940    7010   34200
```

Une fois les flux entrants et sortants calculés, ils sont regroupés dans un nouveau tableau. Dans le vocabulaire de l'analyse de graphes, ce qui vient d'être calculé est le degré des nœuds pondéré par le flux (ce même calcul sera fait sur le graphe à la Section 8.3). Ici la commune qui présente le flux sortant maximum envoie 34 200 individus vers les autres communes de l'espace d'étude. Enfin, les soldes résidentiels sont calculés : solde absolu d'abord, qui est la différence entre les flux entrants et les flux sortants ;

1. Les flux intracommunaux sont les flux d'une commune vers elle-même, c'est-à-dire le nombre d'individus qui n'ont pas changé de commune de résidence. Dans le vocabulaire de l'analyse de graphes, le lien qui relie un noeud à lui-même est appelé « boucle » (*loop*).

solde relatif ensuite, qui est un rapport entre le solde absolu et la somme des flux.

```
flowsSum <- data.frame(CODGEO = names(flowsIn),
                       FLOWSOUT = flowsOut,
                       FLOWSIN = flowsIn,
                       stringsAsFactors = FALSE)

flowsSum$ABSDELTA <- flowsSum$FLOWSOUT - flowsSum$FLOWSIN
flowsSum$RELDELTA <- with(flowsSum, ABSDELTA /
                            (FLOWSOUT + flowsSum$FLOWSIN))
```

Ces variables qui mesurent les soldes résidentiels absolu et relatif peuvent être cartographiées avec les techniques présentées dans le Chapitre 10. Voici le résultat obtenu [1] : sans surprise, les soldes négatifs correspondent aux arrondissements parisiens et à quelques communes limitrophes, les soldes positifs correspondent au reste des communes de la petite couronne. Ce résultat ne concerne que le système de flux entre Paris et la petite couronne, pour une analyse plus complète il faudrait élargir l'analyse à l'ensemble de la région Île-de-France, de la France et/ou du monde.

8.2 Création et exploration du graphe

8.2.1 Importation et exportation de graphes

Dans l'exemple ci-dessous, un graphe est créé à partir d'un tableau de flux entre des communes. La manipulation du tableau et sa transformation en graphe sont intégrées dans un flux de travail entièrement effectué avec le logiciel R. Cependant, il arrive que l'utilisateur récupère des données sous un format spécifique, propre aux logiciels d'analyse de graphes, ou bien qu'il veuille exporter un graphe créé avec R vers un autre logiciel de ce type, Ucinet ou Gephi par exemple.

Les deux *packages* présentés ici, `statnet` et `igraph`, disposent de fonctions pour importer (*read*) et exporter (*write*) les graphes dans des

1. Attention, la carte en noir et blanc n'est pas satisfaisante d'un point de vue sémiologique, il serait plus pertinent d'utiliser ici une palette divergente. Voir à ce propos le Chapitre 9 et en particulier la Section 9.2.

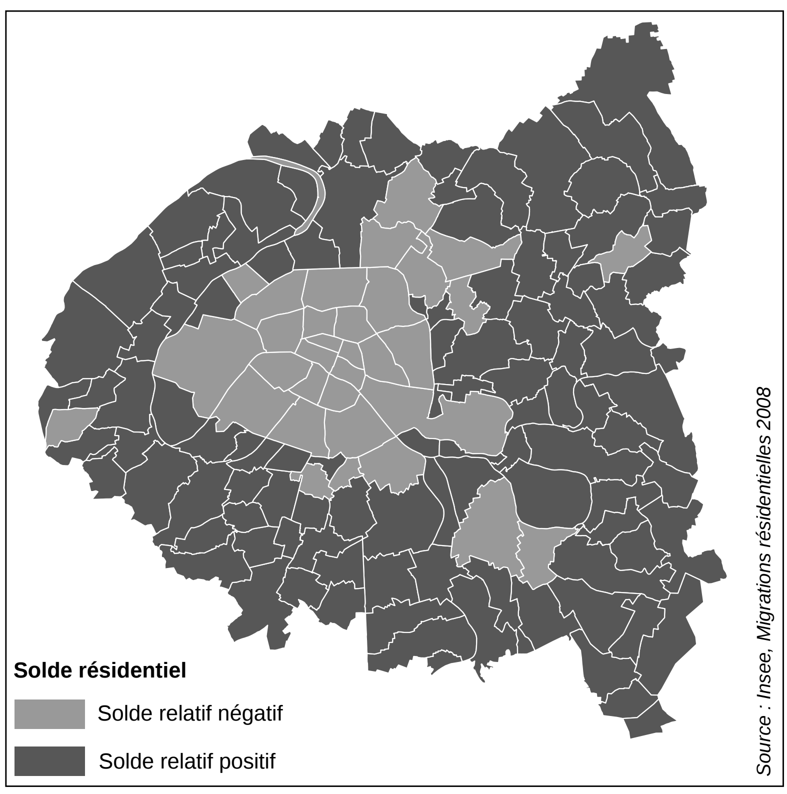


FIGURE 8.2 – Solde résidentiel des communes de Paris et petite couronne

formats de données spécifiques. Avec `statnet`, ces fonctions prennent le nom du format à importer, par exemple `read.paj()` pour un format Pajek. Avec `igraph`, les fonctions sont `read.graph()` et `write.graph()` et le format est précisé dans les arguments des fonctions (pajek, gml, graphml, etc.).

En termes de calculs, R n'a rien à envier aux logiciels spécialisés tels que Ucinet ou Gephi. La principale raison pour laquelle un utilisateur pourrait vouloir utiliser l'un de ces logiciels réside dans la visualisation statique et/ou interactive du graphe.

8.2.2 Création d'un graphe avec R

La liste de liens peut être utilisée comme argument pour créer un graphe. Cet objet sera de type *network* avec le *package* statnet et de type *igraph* avec le *package* igraph. Avec statnet, il faut d'abord transformer le tableau de liens en matrice, puis ajouter l'attribut, ici le flux, une fois le graphe créé.

```
library(statnet)
edgesMat <- as.matrix(edgesList[ , 1:2])
residNet <- network(edgesMat, directed = TRUE)
set.edge.attribute(residNet,
                   attrname = "NBFLUX",
                   value = edgesList$NBFLUX)
```

La transformation du tableau en graphe se fait de façon semblable avec igraph. La fonction utilisée prend directement un *data.frame* comme argument.

```
library(igraph)
residGraph <- graph.data.frame(edgesList, directed = TRUE)
```

Dans la suite de ce chapitre, les principales fonctions sont données pour les deux *packages*, mais les traitements sont faits uniquement avec igraph. Ils seraient bien sûr réalisables avec statnet en utilisant les fonctions indiquées.

8.2.3 Calcul de mesures globales

Pour explorer la structure du graphe, les deux *packages* disposent d'un ensemble de fonctions permettant de calculer des mesures dites « globales », c'est-à-dire portant sur le graphe dans son ensemble. Les plus couramment utilisées sont les suivantes :

- l'ordre : nombre de sommets,
- la taille : nombre de liens,
- la densité : nombre de liens présents sur nombre de liens maximum,
- le nombre de composantes connexes,

— le diamètre : longueur du plus long des plus courts chemins,
— la transitivité : proportion de triangles fermés.

Ces mesures sont particulièrement intéressantes pour comparer des graphes entre eux. Le tableau suivant donne la syntaxe à utiliser dans les *packages* étudiés ici, pour un graphe nommé `g`.

Mesure	Avec statnet	Avec igraph
Ordre	`g`	`g`
Taille	`g`	`g`
Densité	`gden(g)`	`graph.density(g)`
Comp. connexes	`components(g)`	`clusters(g)`
Diamètre	? *	`diameter(g)`
Diades	`dyad.census(g)`	`dyad.census(g)`
Triades	`triad.census(g)`	`triad.census(g)`
Transitivité	`gtrans(g)`	`transitivity(g)`

* Il ne semble pas y avoir de fonction pour calculer directement le diamètre avec `statnet`. Une solution est de calculer les plus courts chemins avec la fonction `geodist()` et de choisir le plus long des plus courts chemins.

TABLE 8.1 – Mesures globales d'un graphe `g`

Une rapide exploration du graphe est nécessaire pour se faire une idée de ses caractéristiques très particulières : un nombre de liens très importants (14 935) par rapport au nombre de nœuds (143), ce qui se traduit par une densité très forte (0,74). En effet le nombre de liens existants est proche du nombre de liens maximum, égal à $n(n-1)$ dans le cas d'un graphe orienté et non planaire. Le graphe ne comporte qu'une seule composante connexe et son diamètre est très faible (2).

```
residGraph
graph.density(residGraph)
clusters(residGraph)
diameter(residGraph)
```

8.3 Mesures locales et manipulation du graphe

Les mesures locales sont les mesures qui renseignent sur chaque sommet ou chaque lien du graphe, les plus courantes sont les suivantes :

— le degré : nombre de liens incidents pour un sommet. Dans un graphe orienté, on distingue le degré entrant et le degré sortant ;
— le degré pondéré : nombre de liens incidents pondéré par le poids de ces liens ;
— la famille de mesures (centralité, accessibilité) qui s'appuient sur le calcul des plus courts chemins.

Les mesures qui s'appuient sur le calcul des plus courts chemins peuvent être calculées aussi bien pour les nœuds que pour les liens. L'utilisateur peut souvent choisir de calculer ces mesures sur le graphe en prenant en compte une pondération des liens, c'est-à-dire un attribut quantitatif attaché aux liens. Les fonctions permettant de calculer ces différents indicateurs sont listées dans le tableau suivant.

Mesure	**Avec statnet**	**Avec igraph**
Degré	`degree(g)`	`degree(g)`
Proximité	`closeness(g)`	`closeness(g)`
Intermédiarité	`betweenness(g)`	`betweenness(g)`
Plus court chemin	`geodist(g)`	`shortest.paths(g)`

TABLE 8.2 – Mesures locales d'un graphe `g`

Le calcul des degrés pondérés et non pondérés rejoint le calcul fait dans la première section de ce chapitre. Le degré pondéré donne une idée de la masse qui est envoyée ou reçue alors que le degré non pondéré donne une idée de la variété des destinations ou des origines. La commune qui a le flux sortant maximum envoie 34 200 individus vers les autres communes de l'espace d'étude, la commune qui a la plus grande variété de destinations envoie des individus vers 138 autres communes distinctes, c'est-à-dire vers la presque totalité de cet espace.

```
degOut <- degree(residGraph, mode = "out")

wtdDegOut <- graph.strength(residGraph,
                             mode = "out",
                             weights = E(residGraph)$NBFLUX)
summary(degOut)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    23.0    93.5   111.0   104.0   124.0   138.0

summary(wtdDegOut)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     168    2370    3850    5940    7010   34200
```

Pour manipuler les attributs du graphe, la syntaxe est différente selon le *package* utilisé, `statnet` ou `igraph`. Avec `igraph`, il est possible de lire et d'écrire des attributs de nœuds et des liens avec les fonctions `V()` (V pour *vertices*) et `E()` (E pour *edges*), suivies de l'opérateur `$`, suivi du nom de l'attribut. C'est cette syntaxe qui vient d'être utilisée pour pondérer les liens par la variable de flux : `E(residGraph)$NBFLUX`.

Ces fonctions peuvent être utilisées pour faire appel à un attribut du graphe (lire l'attribut) mais aussi pour enrichir le graphe de nouveaux attributs (écrire l'attribut dans l'objet). Dans cet exemple, un attribut est ajouté aux nœuds, le degré sortant. En exécutant l'objet ou avec la fonction `summary()`, on vérifie que l'ajout a été fait : il y a bien un nouvel attribut nommé **DEGOUT**, qui caractérise les nœuds (v) et qui est numérique (n).

```
V(residGraph)$DEGOUT <- degree(residGraph, mode = "out")
residGraph

## IGRAPH DN-- 143 14935 --
## + attr: name (v/c), DEGOUT (v/n), NBFLUX (e/n)
```

Si l'utilisateur enrichit le graphe sur lequel il travaille avec de nouveaux attributs et qu'il veut récupérer l'ensemble pour le traiter sous forme de tableau, la fonction `get.data.frame()` permet cette transformation.

8.4 Cliques et communautés

C'est une pratique fréquente de chercher à l'intérieur d'un graphe des sous-groupes cohérents, c'est-à-dire des sous-graphes fortement connexes : des cliques (sociologues), des communautés (physiciens) ou des *clusters* (informaticiens). De nombreuses méthodes existent pour détecter ces sous-graphes et seules les principales sont évoquées.

Une première famille de méthodes s'appuie sur la notion de clique, terme qui désigne un sous-graphe maximal complet comprenant au moins trois sommets. En d'autres termes, une clique est un ensemble de sommets (au moins 3), entre lesquels tous les liens possibles sont présents (graphe complet) et il n'est pas possible d'ajouter un sommet sans que la propriété précédente ne disparaisse (graphe maximal). Cette définition est rarement opérationnelle, elle peut être trop restrictive (cas d'un réseau de connaissances dont les liens sont ténus) ou pas assez restrictive. C'est le cas ici sur le graphe des mobilités résidentielles dont la densité est très forte. Plusieurs paramètres permettent de moduler la définition de la clique pour l'assouplir ou au contraire la restreindre.

Il y a principalement deux façons d'assouplir la définition de la clique : jouer sur la distance et jouer sur les degrés. Au lieu de considérer un ensemble de sommets entre lesquels tous les liens possibles sont présents (distance de 1), il est possible de considérer une distance plus grande : par exemple, chaque sommet doit être relié à tous les autres avec une distance de 2. Ceci revient, dans un réseau social, à considérer que deux individus non reliés directement mais qui connaissent un même individu tiers sont effectivement reliés à une distance 2 (l'ami d'un ami est mon ami). Le type de cliques formées de cette façon sont appelés des *n-cliques*.

En assouplissant le critère du degré, on peut créer des sous-ensembles dans lesquels tous les sommets sont connectés à *k* sommets de la clique (*k-core*) ou encore des sous-ensembles dans lesquels tous les liens possibles moins *k* sont présents (*k-plex*).

Les fonctions disponibles pour calculer ces sous-ensembles sont : `clique.census()` et `kcores()` pour le *package* `statnet` ; `cliques()` et `graph.coreness()` pour le *package* `igraph`. Avec `statnet`, la fonction `clique.census()` permet d'obtenir la taille

des différentes cliques présentes, leur composition, le nombre de cliques auxquelles appartient chaque sommet et enfin les co-appartenances de sommets entre les différentes cliques. Avec `igraph`, il y a des fonctions qui renvoient uniquement le nombre de cliques, `cliques.number()` par exemple, et des fonctions qui renvoient l'appartenance des sommets aux cliques, `cliques()` par exemple.

Ces approches sont plus adaptées pour des réseaux sociaux que pour le cas étudié ici. En effet, le graphe utilisé est très dense et le poids des liens (i.e. le flux résidentiel) porte une information très importante qui doit nécessairement être prise en compte dans le calcul de sous-ensembles. Il existe un grand nombre de méthodes pour créer des sous-ensembles cohérents qui s'appuient sur la modularité. Ces méthodes sont plutôt utilisées par les physiciens, on les retrouvera donc dans le *package* `igraph` plutôt que dans le *package* `statnet`.

Pour un graphe dans lequel on distingue plusieurs communautés, la modularité est forte quand les liens intracommunautés sont forts et les liens intercommunautés sont faibles. Cette mesure s'applique à des graphes non orientés mais elle peut prendre en compte le poids des liens. Elle est définie comme la différence entre la proportion observée de liens intracommunauté et la proportion que l'on observerait dans un graphe aléatoire conservant la distribution de degrés du graphe originel :

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{w_i w_j}{2m} \right] \delta(c_i, c_j)$$

Où m est la somme de la matrice de poids, où A_{ij} est le poids du lien entre i et j et où la fonction δ est une fonction qui renvoie 1 si i et j font partie de la même communauté et 0 sinon. La modularité peut être utilisée comme simple mesure de la qualité d'une partition, c'est-à-dire pour évaluer *a posteriori* une partition obtenue par une méthode quelconque. Cependant, la modularité peut aussi être utilisée comme méthode de partionnement dans un algorithme qui cherche à la maximiser pour produire une partition cohérente. C'est sur ce principe que fonctionne l'algorithme utilisé, dit « méthode de Louvain » : il produit une partition du graphe à deux niveaux qui correspond à un optimum de modularité.

Le graphe est d'abord transformé en graphe non orienté, les arguments de la fonction (`mode` et `edge.attr.comb`) précisent que s'il y a un lien de i vers j et un lien de j vers i, il faut les confondre et faire la somme des attributs des deux liens. Ainsi, si Créteil envoie 1 000 individus vers Melun et Melun 500 individus vers Créteil, le sens et l'asymétrie de la relation seront perdus : on considérera qu'il y un seul lien non orienté entre ces deux communes représentant un flux de 1 500 individus.

La fonction `multilevel.community()` renvoie une liste d'éléments qui contient les noms des sommets et leur appartenance aux communautés créées. On accède à ces éléments avec l'opérateur `$` et on les regroupe dans un tableau.

```
undirGraph <- as.undirected(residGraph,
                            mode = "collapse",
                            edge.attr.comb = sum)

mltcomResid <- multilevel.community(
  undirGraph,
  weights = E(undirGraph)$NBFLUX
)

mltClust <- t(rbind(mltcomResid$memberships,
                    mltcomResid$names))

V(undirGraph)$CLUST <- mltClust[, 1]
```

Là encore il sera intéressant de cartographier l'appartenance des communes à ces communautés avec les techniques présentées dans le Chapitre 10. Le résultat est affiché sur la carte suivante [1].

La fonction propose deux niveaux d'agrégation, c'est le plus agrégé qui est cartographié ici. Cinq sous-ensembles sont distingués qui correspondent presque exactement au découpage départemental. Seul le département 75 (Paris) est divisé en deux : les arrondissements de l'Ouest forment une communauté avec l'ensemble des communes des Hauts-de-Seine alors que les arrondissements de l'Est forment une communauté avec quelques communes limitrophes : Bagnolet, Montreuil ou Pantin.

1. Attention, la carte en noir et blanc n'est pas satisfaisante d'un point de vue sémiologique, il serait plus pertinent d'utiliser ici une palette qualitative. Voir à ce propos le Chapitre 9 et en particulier la Section 9.2.

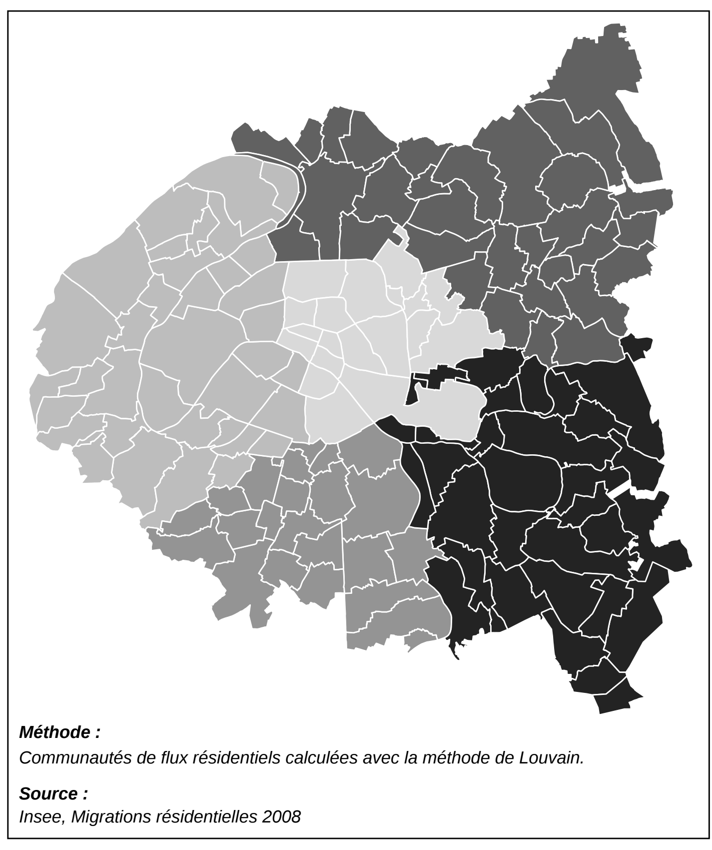


FIGURE 8.3 – Communautés de flux résidentiels (2008)

Cette division Est-Ouest de Paris se retrouve dans bon nombre de variables sociales, économiques ou politiques, la plus fameuse étant la division du vote PS-UMP.

8.5 Visualisations

Visualiser les graphes est l'un des réflexes les plus courants, et peut-être l'un des plus trompeurs, en analyse de graphes. En effet, il faut être attentif aux paramètres de visualisation (épaisseur des liens, taille des nœuds) et aux principes de l'algorithme utilisé pour la visualisation. Les deux *packages* proposent des outils et options comparables. Le résultat par défaut est plus lisible avec `statnet` qu'avec `igraph` mais les arguments des fonctions permettent d'obtenir des visualisations personnalisées de grande qualité quel que soit le *package* utilisé.

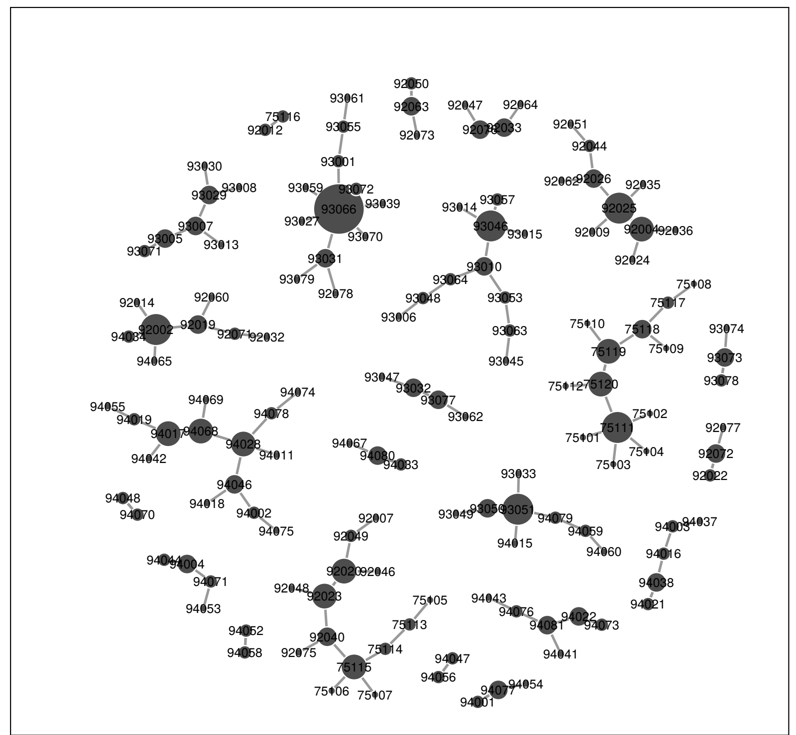
La taille et la densité du graphe des mobilités résidentielles le rendent difficile à visualiser directement. Pour explorer sa structure, il faudrait travailler sur des sous-graphes, par exemple sur les communautés créées dans la section précédente, ou encore sur des sélections des liens. Il existe plusieurs façons de faire émerger des structures simples de graphes très denses, la plus simple étant de sélectionner pour chaque nœud le flux maximum envoyé.

Dans un premier temps, on récupère à partir de la matrice origine-destination l'index de colonne du flux maximum envoyé. Pour cela on applique la fonction `which.max()` ligne à ligne avec la fonction `apply()` (cf. Section 3.4). Cette fonction renvoie un vecteur d'indexation : par exemple, si la première valeur de ce vecteur (correspondant à la commune 75101) est 11, ceci signifie que le flux maximum est atteint à la colonne 11 de la matrice. Il suffit ensuite d'utiliser cet index pour récupérer les noms de colonnes qui correspondent aux destinations des flux. La colonne 11 correspond par exemple à la commune 75111 : le flux résidentiel maximum en provenance du 1er arrondissement se dirige vers le 11e arrondissement. Ces liens sont réorganisés dans un tableau et la fonction `graph.data.frame()` transforme ce tableau en graphe.

```
indexMax <- apply(adjMat, 1, which.max)
destMax <- colnames(adjMat)[indexMax]
maxFlows <- data.frame(ORI = names(indexMax),
                        DES = destMax,
                        stringsAsFactors = FALSE)

maxGraph <- graph.data.frame(maxFlows, directed = TRUE)
maxGraph

## IGRAPH DN-- 143 143 --
## + attr: name (v/c)
```

Le graphe ainsi créé peut être visualisé directement avec la fonction générique `plot()`. La sortie par défaut n'est pas satisfaisante, il faut spécifier un certain nombre d'arguments graphiques pour obtenir un bon résultat, en particulier la couleur et l'épaisseur des liens (`edge.color` et `edge.width`), la couleur et la taille des nœuds (ici proportionnelle au degré), la disposition du graphe (`layout`). Un grand nombre d'arguments sont disponibles aussi bien dans `igraph` que dans `statnet`. Si l'utilisateur utilise beaucoup ces visualisations avec des réglages spécifiques, il peut facilement les encapsuler dans une fonction créée à cet effet et ainsi éviter de réécrire tous ces arguments à chaque fois.

```
plot(maxGraph,
     edge.color = "grey60",
     edge.width = 2,
     edge.curved = F,
     edge.arrow.mode = "-",
     vertex.frame.color = "white",
     vertex.color = "grey30",
     vertex.label = V(maxGraph)$name,
     vertex.label.cex = 0.8,
     vertex.label.color = "black",
     vertex.size = 2 * degree(maxGraph),
     layout = layout.fruchterman.reingold(maxGraph))
```



La sélection proposée dans cet exemple est rudimentaire mais elle laisse déjà apparaître certaines structures : les grands arrondissements de l'Est parisien (11e, 18e, 19e, 20e) comme destination principale des flux intra-parisiens ; le 15e arrondissement qui fait le lien entre les arrondissements de l'Ouest et les communes des Hauts-de-Seine (92) ; certains pôles d'attraction départementaux apparaissent, comme Saint-Denis (93066) pour le département de Seine-Saint-Denis (93) ou Créteil (94028) pour les communes du Val-de-Marne (94). Apparaissent également de petits sous-systèmes avec des communes peu nombreuses, souvent limitrophes, qui envoient leurs flux principaux les unes vers les autres, par exemple le trio Tremblay (93073)-Vaujours (93074)-Villepinte (93078).

L'argument `layout` prend comme argument le résultat d'algorithmes classiques de visualisation, comme celui de Fruchterman et Reingold ou celui de Kamada et Kawai. Ces algorithmes sont implémentés aussi bien

dans `igraph` que dans `statnet`. Il est cependant possible pour l'utilisateur de proposer ses propres coordonnées sous forme d'une matrice de n lignes (nombre de nœuds) et 2 colonnes pour les coordonnées X et Y. En donnant comme argument les coordonnées géographiques des centroïdes des communes, il serait par exemple possible d'afficher le graphe dans le système de projection choisi.

CHAPITRE 9

Focus sur la visualisation graphique

Objectifs : *Ce chapitre est une mise au point sur la visualisation graphique avec R. Il traite de l'exportation des tableaux et des graphiques, il présente la gestion des couleurs et il introduit, en préambule au chapitre sur la cartographie, la manipulation du* package `ggplot2`.

❄

Avertissement Ce chapitre sur la visualisation et la couleur est destiné à être imprimé en noir et blanc, le code utilisé est reproduit mais les sorties graphiques sont rarement affichées. C'est également cette contrainte éditoriale qui amène, dans l'ensemble du manuel, à ne pas toujours respecter les règles de sémiologie graphique concernant les couleurs (cf. Section 9.2).

Prérequis Types d'objet présentés dans le Chapitre 2 ; graphiques présentés dans le Chapitre 4 et le Chapitre 5. Notions de sémiologie graphique.

Description des *packages* utilisés Plusieurs *packages* sont proposés pour l'exportation des tableaux et la création dynamique de contenus : `tables` pour exporter des tableaux dans des tableurs, `xtable` pour exporter les tableaux au format LaTeX et `knitr` pour produire des documents dynamiques.

La représentation graphique demande également une manipulation de palettes de couleurs. Il existe aussi des *packages* spécialisés dans ce domaine, en particulier `RColorBrewer`.

Plusieurs *packages* spécialisés dans la représentation graphique sont disponibles. Les deux principaux *packages* graphiques généralistes sont `lattice` et `ggplot2`, d'autres proposent des fonctions spécifiques, comme le *package* `vcd` pour la visualisation de variables qualitatives. C'est `ggplot2` qui est présenté ici, pour les raisons détaillées par la suite.

Description des données utilisées Les applications présentées dans ce chapitre sont effectuées sur les données historiques contenues dans le *package* `HistData` déjà utilisé à la Section 3.4 : les données de « statistique morale » d'André-Michel Guerry et les données sur le choléra de John Snow. Les premières comprennent un ensemble de variables touchant à la « moralité » comme le nombre de suicides, de prostituées ou de désertions dans les départements français vers 1830. Les secondes présentent des données spatialisées sur l'épidémie de choléra à Londres dans les années 1850. Les travaux de Guerry et ceux de Snow sont d'une importance majeure dans l'histoire de la statistique et de l'épidémiologie, ce pourquoi leurs jeux de données ont été mis à disposition par Michael Friendly et Stéphane Dray [1].

1. Voir en particulier la page dédiée au *package* `Guerry` : `http://cran.r-project.org/web/packages/Guerry/index.html`.

9.1 Exportation des tableaux et des graphiques

L'utilisateur peut visualiser des résultats numériques et graphiques dans l'environnement RStudio, mais il a souvent besoin d'exporter ces sorties pour les intégrer dans des documents rédigés. Plusieurs cas sont envisagés dans le tableau qui suit, selon le type de sortie (tableau ou image) et le mode de travail, en particulier le logiciel utilisé pour produire du contenu.

Deux types de flux de travail (*workflow*) peuvent être distingués. Le plus classique consiste à découpler l'écriture du texte et la production des sorties numériques ou graphiques : l'utilisateur produit un graphique avec R, l'exporte dans un format d'image et l'insère dans un logiciel de traitement de texte avec lequel il écrit le contenu. L'autre façon de procéder, de plus en plus développée, s'inscrit dans la vague de la *reproducible research*. Elle consiste à combiner l'écriture des contenus textuels et des traitements numériques et graphiques pour générer un document tout-en-un. Ce flux de travail est souvent qualifié de *dynamic report generation*.

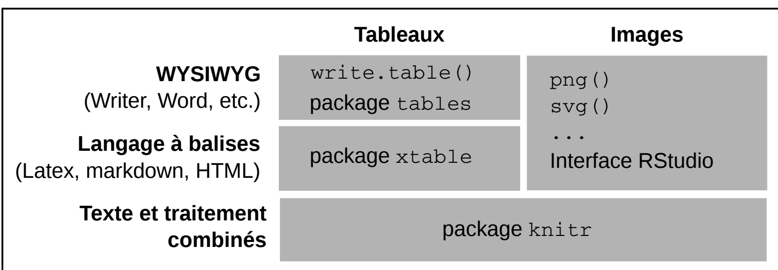


FIGURE 9.1 – Exportation de tableaux et graphiques

Si la sortie est un tableau et que le document est rédigé sur un traitement de texte de type WYSIWYG (*What You See Is What You Get*) tel que LibreOffice Writer ou MS Word, il faut nécessairement passer par une exportation dans un tableur avec les fonctions `write.table()` présentées dans le premier chapitre. Le *package* `tables`, avec ses fonction `tabular()` et `write.table.tabular()`, facilite ce travail.

Si la sortie est un tableau et que le document est rédigé avec un langage à balises – LaTeX, markdown ou HTML – le package `xtable` propose une

fonction éponyme `xtable()` qui transforme le tableau en code directement utilisable dans le document rédigé. Dans cet exemple, un tableau de dénombrement des départements selon la variable **Region** est exporté au format LaTeX :

```
library(xtable)
library(HistData)
data(Guerry)

xtable(table(Guerry$Region))
```

Si la sortie est une image (graphique ou carte), elle peut être exportée *via* l'interface graphique de RStudio. Dans l'onglet **Plots** plusieurs options d'exportation des graphiques sont proposés : il est possible de fixer la taille de l'image et de choisir parmi plusieurs formats courants, par exemple svg, png ou pdf. Pour plus de précision dans les réglages et/ou pour intégrer l'exportation dans le script, il faut utiliser fonctions graphiques qui prennent le nom du format à produire : `png()`, `svg()`, `jpeg()`, etc. Ces fonctions initialisent le dispositif graphique, il doit ensuite être fermé par la fonction `dev.off()`. L'exemple suivant exporte un fichier png qui contient l'histogramme de la variable **Crime_pers**.

```
png(filename = "chemin/fichier.png")
hist(Guerry$Crime_pers)
dev.off()
```

Dans la fonction `png()`, l'utilisateur peut fixer une taille (arguments `width` et `height`) dans une unité de longueur (`units`) comme les pouces ou les centimètres, il peut également préciser la résolution en ppi (*pixels per inch*, pixels par pouce) pour obtenir une taille en termes de nombre de pixels. L'utilisateur peut aussi fixer directement la taille en pixels en utilisant ces trois arguments. Enfin, l'argument `pointsize` est une mesure relative de la taille des caractères en fonction de la résolution.

Dans ce flux de travail, la meilleure façon d'obtenir des images prêtes à l'emploi est de les paramétrer le mieux possible avec R, de les exporter au format svg et éventuellement de les retoucher sur un logiciel de dessin vectoriel comme Inkscape.

Pour produire un document qui fourmille de traitements et de graphiques, l'arme ultime est la combinaison de l'écriture et des traitements dans un même document. R et RStudio disposent d'outils très pratiques pour cela grâce au *package* `knitr`[1]. Dans l'interface de RStudio, le menu **File > New file** propose plusieurs solutions : **R Sweave** pour une combinaison avec LaTeX (ce manuel est écrit de la sorte), **R Markdown**, **R HTML** et **R Presentation** pour produire des documents écrits en markdown et en HTML. À la date d'écriture de ce manuel, la version de RStudio la plus récente permet d'écrire un document en markdown et de compiler le texte et le code pour obtenir une sortie en pdf, en HTML et en docx (Microsoft Word).

9.2 Gestion des couleurs

Sans rentrer dans le détail du codage des couleurs pour l'impression et la visualisation numérique, il faut simplement mentionner que plusieurs conventions existent : le code RGB (*red green blue*, RVB en français) qui indique le mélange des trois couleurs primaires pour obtenir une couleur donnée ; le code CMYK (*cyan, magenta, yellow, key*, CMJN en français) issu du monde de l'imprimerie ; le code hexadécimal issu des standards des navigateurs Internet.

Avec R, tous ces codes peuvent être utilisés en cas de contrainte éditoriale forte, mais il s'agit rarement de la méthode la plus simple. Le plus courant est d'assigner des couleurs prédéfinies. Pour obtenir l'ensemble des couleurs disponibles, il faut exécuter la fonction `colors()`. Cette fonction renvoie une liste de 657 couleurs prédéfinies[2].

Les couleurs peuvent d'abord être désignées par leur nom :

```
colors()
hist(Guerry$Crime_pers, col = "red", border = "green")
hist(Guerry$Crime_pers,
      col = "springgreen",
      border = "darksalmon")
```

1. Voir le site dédié : `http://yihui.name/knitr`.
2. Voir à ce propos la page web d'Earl F. Glynn : `http://research.stowers-institute.org/efg/R`.

Les couleurs peuvent aussi être désignées par leur numéro dans le vecteur de 657 couleurs renvoyé par la fonction `colors()`. Ainsi, les deux lignes suivantes sont équivalentes :

```
hist(Guerry$Crime_pers,
     col = colors()[26],
     border = colors()[133])

hist(Guerry$Crime_pers, col = "blue", border = "firebrick")
```

Les couleurs peuvent être désignées par leur code hexadécimal :

```
hist(Guerry$Crime_pers,
     col = "#FF0000",
     border = "#FFAA00")
```

Enfin, R dispose de palettes de couleurs prédéfinies, par exemple `rainbow()` ou `heat.colors()`. Ces fonctions demandent comme argument le nombre de couleurs à extraire de la palette et renvoient le code hexadécimal correspondant :

```
rbPal <- rainbow(n = 5)
boxplot(Guerry$Crime_pers ~ Guerry$Region, col = rbPal)
```

Certains *packages* proposent des palettes de couleurs supplémentaires, le plus connu étant `RColorBrewer` qui est une implémentation des palettes de couleurs créés par Cynthia Brewer [1]. Ce *package* contient des palettes avec des variations de couleurs et de valeurs adaptées à trois grands types de variables statistiques : des variables continues, des variables continues avec une valeur seuil, des variables qualitatives.

Pour les premières, la variation statistique est traduite visuellement par une montée en valeur sur une même couleur, du vert clair au vert foncé par exemple. Pour les deuxièmes, la variation statistique est traduite par une montée en valeur sur deux couleurs distinctes divisées autour d'un seuil. Cette palette dite « divergente » est efficace pour représenter par exemple un solde qui prend des valeurs négatives et positives, les valeurs négatives

1. Voir le site dédié : `http://colorbrewer2.org`.

seront traduites dans une couleur froide (bleu par exemple) et les positives dans une couleur chaude (rouge par exemple). Enfin, les variables qualitatives sont représentées par une variation de couleur.

Toute carte statistique devrait suivre ces règles sémiologiques simples qui établissent des liens cohérents entre la variation statistique d'un caractère et la variation visuelle représentée sur la carte. Ces règles sont difficiles à respecter lorsqu'on ne dispose que de niveaux de gris, c'est pourquoi plusieurs cartes produites dans ce manuel sont signalées en note de bas de page comme ne respectant pas les règles qui viennent d'être énoncées.

Voici le code pour afficher l'ensemble des palettes disponibles, pour afficher seulement un certain type de palette (divergente dans ce cas) et pour créer une palette de *n* couleurs (6 dans ce cas).

```
library(RColorBrewer)
display.brewer.all()
display.brewer.all(type = "div")
pastelPal <- brewer.pal(n = 6, name = "Pastel2")
```

Le *package* RColorBrewer est utilisé par défaut pour les représentations graphiques produites avec `ggplot2`, *package* présenté dans la section qui suit.

9.3 Introduction à `ggplot2`

9.3.1 Intérêt vis-à-vis des fonctions graphiques classiques

C'est `ggplot2` qui est présenté ici pour plusieurs raisons. D'abord, il s'agit du *package* graphique le plus polyvalent et le plus utilisé, c'est d'ailleurs l'un des *packages* les plus téléchargés sur l'ensemble des *packages* disponibles sur le CRAN. Ensuite, ce *package* est conçu comme l'implémentation d'une « grammaire graphique » : il s'appuie donc sur une base théorique robuste qui systématise le passage des attributs statistiques d'un tableau aux attributs esthétiques d'une représentation graphique. Enfin, il est développé par Hadley Wickham, l'un des développeurs les plus productifs de la communauté R : il a vocation à s'étendre et

à communiquer avec d'autres *packages*, en particulier dans le domaine de la visualisation interactive et de la cartographie.

Le *package* `ggplot2` est décrit dans la plupart des manuels de visualisation avec R, il est présenté en détail dans un manuel dédié, il dispose enfin d'un site web dédié [1]. L'avantage de ce *package* est qu'il permet une manipulation fine des attributs esthétiques. Les fonctions graphiques de base, utilisées dans les chapitres précédents, sont simples et rapides : `boxplot()` et `hist()` permettent par exemple de faire des boîtes à moustaches et des histogrammes, de modifier quelques paramètres comme la couleur ou la taille. Cependant, leur domaine d'utilisation est limité à ces quelques paramétrages.

Avec `ggplot2`, il est possible de produire tous types de graphiques, pas seulement les grands classiques. On peut produire de nouveaux types de représentations graphiques ou encore reproduire des graphiques uniques : celui de Minard par exemple (cf. Figure 3.4) [2], certains graphiques oubliés proposés par Tufte comme le *slopegraph* par exemple.

9.3.2 Aperçu de la syntaxe

La polyvalence de `ggplot2` est possible parce qu'il permet un travail fin sur les éléments suivants, constitutifs d'une représentation graphique :

— variable visuelle (**mappings**) : variation esthétique correspondant à la variation statistique, en particulier la taille et la couleur des éléments ;
— géométrie (**geoms**) : implantation géométrique des objets graphiques, principalement des points, des lignes et des polygones ;
— échelle (**scales**) : variation dans l'espace des valeurs numériques mise à l'échelle dans l'espace des variables visuelles.
— système de coordonnées (**coord**) : projection des valeurs sur le plan, par exemple avec un système de coordonnées cartésien, polaire ou un système de projection cartographique ;

1. `http://docs.ggplot2.org`.
2. Voir à ce propos la page web de Michael Friendly : `http://www.datavis.ca/gallery/re-minard.php`.

— transformation statistique (**stats**) : élément optionnel, il est utile par exemple pour représenter graphiquement des résumés numériques (fréquences absolues ou relatives) ;
— construction d'une planche (**facet**) : élément optionnel, il est utile pour construire une planche constituée de plusieurs graphiques correspondant à des sous-groupes de données.

Pour utiliser `ggplot2`, il faut d'abord se familiariser avec une syntaxe et un fonctionnement particuliers. D'abord, les données statistiques à graphier doivent nécessairement être contenues dans un objet de type *data.frame*. Ensuite, à la différence de la plupart des fonctions graphiques de base, `ggplot2` peut créer un objet qui contient les données et les paramétrages graphiques. Pour échanger un graphique classique produit avec `plot()` par exemple, il faut mettre à disposition à la fois le code de la représentation graphique et les données nécessaires. L'objet `ggplot2`, en revanche, est autonome, il peut être stocké et échangé.

Pour créer un tel objet, le *package* propose une fonction de base `ggplot()` à laquelle on ajoute les différents éléments décrits ci-dessus à l'aide de l'opérateur +. Voici un exemple fictif qui produirait un nuage de points. Les données sont stockées dans un *data.frame* nommé **mydata**, ces données sont représentées par des points dont les coordonnées sont fixées par deux variables quantitatives **x** et **y**. Enfin, la taille et la couleur des points peuvent varier en fonctions d'autres variables présentes dans le tableau.

```
# CAS 1
ggplot(mydata) +
  geom_point(aes(x = , y = , size = , color = )) +
  scale_size(...)
  scale_color(...)

# CAS 2
ggplot(mydata) +
  geom_point(aes(x = , y = ), size = , color = )
```

La différence entre le premier et le second cas est le statut de la taille et de la couleur. Dans le premier, les arguments sont inclus dans la fonction `aes()` (*aesthetics mapping*). La taille et la couleur sont donc considérées comme des variables visuelles : elles vont traduire une variation statistique

par une variation visuelle et cette variation sera mise à l'échelle – échelle de taille et de couleur – par les fonctions de type `scale()`.

Dans le second cas, l'utilisateur fixe une couleur et une taille pour l'ensemble des points : il ne s'agit plus de variables visuelles mais de paramètres graphiques d'ensemble. Dans ce cas, les fonctions de type `scale()` ne sont pas utiles.

9.3.3 Variables visuelles

Cet exemple produit un nuage de points mettant en relation la variable **Prostitutes**, qui le nombre de prostituées à Paris par département de naissance, et la variable **Distance**, qui indique la distance du département à Paris.

```
library(ggplot2)

distProst <- ggplot(Guerry) +
  geom_point(aes(x = Distance, y = Prostitutes),
             size = 3, colour = "black")
distProst
```

Prostitutes
4000
3000
2000
1000
0
0
200
400
Distance

```
class(distProst)

## [1] "gg"     "ggplot"
```

L'objet créé est de type *ggplot*, on peut accéder à son contenu, en particulier aux données, avec l'opérateur $ comme pour les objets de type *list* et *data.frame*. La taille et la couleur sont des paramètres d'ensemble et non des variables visuelles. Le thème par défaut présente un fond gris clair carroyé de lignes blanches, les lignes principales sont fixées avec un pas de 200 pour l'axe des x et un pas de 1 000 pour l'axe des y, le titres des axes est le nom de variables représentées. Tous ces paramètres peuvent bien sûr être modifiés.

D'un point de vue thématique, la variable **Prostitutes** indique le nombre de prostituées à Paris par département de naissance. Ce nombre décroit avec la distance à Paris selon une fonction exponentielle négative, résultat cohérent avec les modèles d'interaction spatiale développés dans le domaine de la géographie et de la démogaphie des migrations.

L'exemple qui suit est un nuage de points mettant en relation les crimes contre la personne et les crimes contre la propriété. Ces deux variables sont exprimées en population par crime, c'est-à-dire l'inverse de tous les taux calculés de nos jours, par exemple un taux de criminalité exprimé en crimes par population. Le thème par défaut peut être substitué par un thème plus sobre : chaque élément du thème peut être paramétré, la couleur du cadre, l'épaisseur des lignes, etc. mais il est plus facile d'utiliser des thèmes tout faits, par exemple le thème `theme_bw()`.

La taille et la couleur sont utilisées comme variables visuelles : la taille traduit le nombre total de crimes, variable calculée au préalable, et la couleur traduit la région (Nord, Sud, Est, Ouest, Centre) [1]. La valeur de la Corse est modifiée et la région Sud lui est assignée.

1. Attention, le graphique en noir et blanc n'est pas satisfaisant d'un point de vue sémiologique, il serait plus pertinent d'utiliser ici une palette qualitative avec différentes couleurs. Voir à ce propos la Section 9.2.

```
Guerry$TotalCrimePers <- with(Guerry,
                              1000 * Pop1831 / Crime_pers)
Guerry$TotalCrimeProp <- with(Guerry,
                              1000 * Pop1831 / Crime_prop)
Guerry$TotalCrime <- with(Guerry,
                          TotalCrimePers + TotalCrimeProp)
Guerry$Region[86] <- "S"

ggplot(Guerry) +
  geom_point(aes(x = Crime_pers,
                 y = Crime_prop,
                 size = TotalCrime,
                 colour = Region)) +
  scale_size_continuous(range = c(3, 15)) +
  scale_color_grey() +
  theme_bw()
```

```
summary(Guerry$TotalCrime)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    20.3    49.2    69.2    84.3    93.7   751.0
```

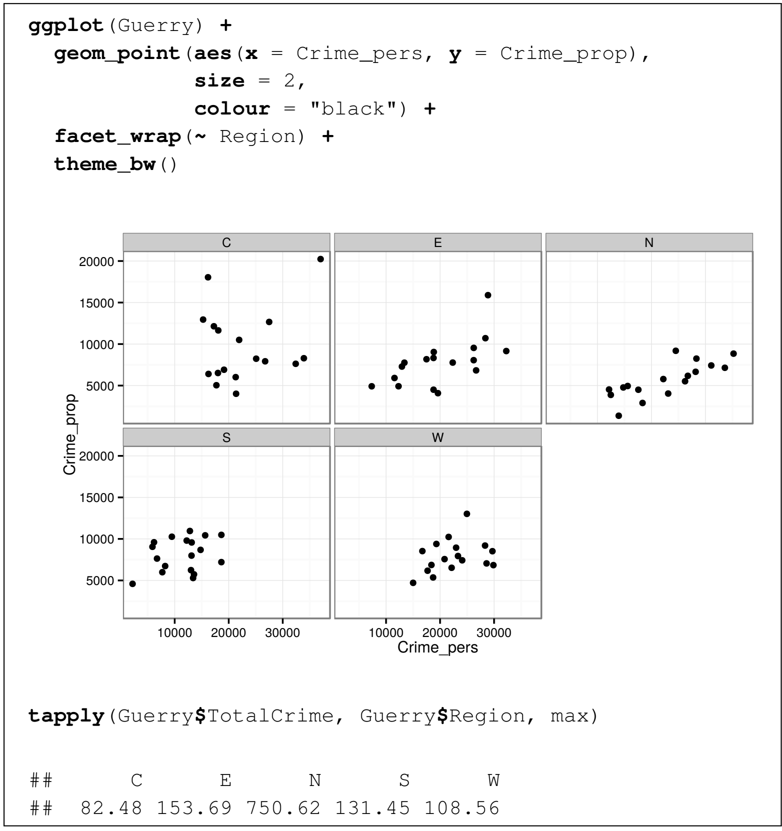
Le graphique fait apparaître de fortes disparités entre ces deux types de crimes ainsi qu'une relation presque inexistante entre les deux. Les deux variables visuelles renseignent sur le nombre absolu de crimes commis

(taille), le département de Seine en tête avec 751 crimes, et sur les grandes régions correspondant aux départements (couleur).

9.3.4 Construction d'une planche

Il est souvent utile de faire construire des planches de graphiques ou de cartes, c'est-à-dire d'inclure dans une même sortie graphique un ensemble de cartes ou de graphiques correspondant à des sous-groupes de données. Ce sont les fonctions `facet_grid` et `facet_wrap` qui permettent cela, fonctions dites de « maillage » (*latticing*). Dans cet exemple, on cherche à produire un nuage de points similaire au précédent, mais en différenciant chaque région.

```
ggplot(Guerry) +
  geom_point(aes(x = Crime_pers, y = Crime_prop),
             size = 2,
             colour = "black") +
  facet_wrap(~ Region) +
  theme_bw()
```




```
tapply(Guerry$TotalCrime, Guerry$Region, max)

##      C      E      N      S      W
##  82.48 153.69 750.62 131.45 108.56
```

Les deux axes x et y se mettent par défaut à la même échelle, ce qui facilite la comparaison des données entre tous les sous-groupes. Ce paramètre peut bien sûr être modifié.

9.3.5 Variables de regroupement

Dans les deux exemples précédents, la variable de regroupement **Region** a été utilisée de deux façons différentes, comme variable visuelle pour colorer les points et comme variable de désagrégation pour construire une planche.

L'application introduit l'usage de ggplot2 pour la cartographie. Deux points essentiels sont abordés : l'utilisation d'une variable de regroupement pour construire des objets à représenter et l'utilisation des systèmes de coordonnées. Un exemple très simple pour commencer : le tableau suivant contient 4 points avec leurs coordonnées et trois variables de regroupement, **verti**, **horiz** et **polyg**.

En utilisant la variable **verti**, on signale qu'il faut regrouper les points P1-P2 et P3-P4 pour former des segments, ce qui se traduit par deux segments verticaux. Avec la variable **horiz**, ce sont les points P1-P3 et P2-P4 qui sont regroupés formant deux segments horizontaux.

```
geomEx <- data.frame(id = c("P1", "P2", "P3", "P4"),
                     coordx = c(1, 1, 2, 2),
                     coordy = c(1, 2, 2, 1),
                     verti = c("S1", "S1", "S2", "S2"),
                     horiz = c("S1", "S2", "S1", "S2"),
                     polyg = rep("POL", 4))

ggplot(geomEx) +
  geom_line(aes(x = coordx, y = coordy, group = verti)) +
  theme_bw()

ggplot(geomEx) +
  geom_line(aes(x = coordx, y = coordy, group = horiz)) +
  theme_bw()
```

La géométrie geom_polygon() fonctionne de même : elle prend comme argument une liste de points avec leurs coordonnées et une va-

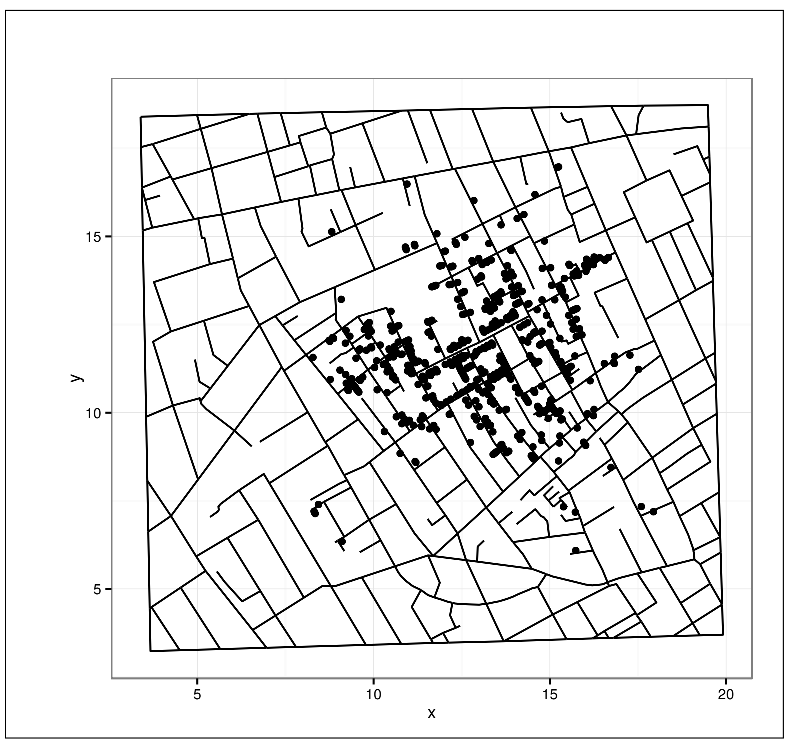
riable de regroupement qui indique quels points doivent être reliés pour former les polygones.

```
ggplot(geomEx) +
  geom_polygon(aes(x = coordx,
                   y = coordy,
                   group = polyg),
               fill = "red") +
  theme_bw()
```

Ce travail est maintenant effectué sur de véritables données spatialisées, celles de John Snow qui indiquent la localisation des morts de choléra recensés en 1854 autour de Broad Street. Le tableau **Snow.deaths** est composé de 578 points avec leurs coordonnées et le tableau **Snow.streets** donne la localisation des rues. Ces rues sont des segments composés de points : les champs **x** et **y** donnent leurs coordonnées, le champ **street** indique quels points il faut relier pour former les segments. Dans ce cas, les données nécessaires pour représenter la carte sont contenues dans deux objets différents. Ces objets sont donnés comme argument dans la fonction `geom_` et non dès la fonction `ggplot()`.

La fonction `coord_equal()` est utile dans la production de cartes : elle force la sortie graphique à respecter un ratio fixe de correspondance entre l'échelle des x et l'échelle des y. Par défaut, ce ratio est de 1, ce qui force à respecter un ratio de 1 pour 1. C'est le ratio qui sera toujours utilisé en cartographie, mais d'autres ratios seraient paramétrables.

```
data(Snow.deaths)
data(Snow.streets)

snowMap <- ggplot() +
  geom_path(data = Snow.streets,
            aes(x = x, y = y, group = street)) +
  geom_point(data = Snow.deaths,
             aes(x = x, y = y), size = 2) +
  coord_equal() +
  theme_bw()

snowMap
```



Les trois géométries qui viennent d'être présentées, `geom_point()`, `geom_path()` et `geom_polygon()`, constituent la base permettant de produire des cartes avec `ggplot2`. Elles seront utiles pour cartographier les trois grands types d'implantation d'objets géographiques : implantations ponctuelle, linéaire et zonale.

À l'issue de ce chapitre, l'utilisateur dispose de tous les éléments pour produire des sorties graphiques de qualité, pour bien gérer les palettes de couleurs et pour exporter ces sorties dans différents formats. Ces bases seront utiles pour aborder le chapitre de cartographie, tout en gardant à l'esprit que les règles de sémiologie graphique ne peuvent pas toujours être respectées en ce qui concerne les palettes de couleur vue la contrainte éditoriale d'édition en noir et blanc.

CHAPITRE 10

Introduction aux objets spatiaux et à la cartographie

Objectifs : *Ce chapitre présente les principales techniques de cartographie d'objets vectoriels (points et polygones) et d'images* raster. *Il mobilise les fonctions graphiques de base mais aussi le* package `ggplot2` *introduit dans le chapitre précédent.*

Avertissement Compte tenu de la contrainte d'impression en noir et blanc, le code utilisé est reproduit mais les sorties graphiques ne sont pas toujours affichées. C'est également cette contrainte qui amène, dans l'ensemble du manuel, à ne pas toujours respecter les règles de sémiologie graphique concernant les couleurs (cf. Section 9.2).

Prérequis Notions de cartographie thématique : discrétisation et sémiologie graphique. Connaissance des formats d'objets spatiaux.

Description des *packages* utilisés L'importation et la manipulation de données spatiales dans R s'appuient sur un certain nombre de *packages* spécialisés et de types d'objets spatiaux. Les fonctions et les objets spatiaux de base sont définis dans le *package* `sp` (*spatial*). Tous les autres *packages* spatiaux dépendent du *package* `sp`. Ce dernier, ainsi qu'un certain nombre d'autres *packages* plus spécialisés, sont développés par Roger Bivand, Edzer Pebesma et Virgilio Gómez-Rubio qui sont également les auteurs du manuel de référence en analyse spatiale avec R.

Avant de commencer cette section, il faut donc installer et charger les *packages* suivants :

- `sp` : la base de tous les autres *packages* spatiaux, dans lequel sont définis les types d'objets spatiaux ;
- `rgdal` : implémentation dans R de la bibliothèque GDAL (Geospatial Data Abstraction Library). Ce *package* permet d'importer de nombreux formats de données spatiales, *raster* et vectorielles ;
- `raster` : fonctions pour la manipulation des fichiers *raster* ;
- `rgeos` : interface avec la bibliothèque Geometry Engine-Open Source (GEOS) qui permet de manipuler la géométrie des objets spatiaux ;
- `mapproj` : fonctions de conversion des coordonnées géographiques en coordonnées projetées ;
- `fields` : ensemble de fonctions dédiées aux objets spatiaux, ce *package* est utilisé ici pour calculer des matrices de distance entre des semis de points ;
- `classInt` : permet de discrétiser des variables continues. Il est utilisé ici pour faire des cartes choroplèthes ;
- `ggplot2` : *package* utilisé pour les représentations graphiques, il est aussi utile pour faire des représentations cartographiques (cf. Chapitre 9) ;
- `scales` : *package* complémentaire de `ggplot2` ;
- `RColorBrewer` : implémentation dans R de Color Brewer, un ensemble de palettes de couleurs ;

— `animation` : permet de faire des représentations graphiques ou cartographiques animées ;
— `reshape2` : contient un ensemble de fonctions permettant de restructurer l'information d'un tableau (cf Section 2.4.3).

Description des données utilisées La distinction entre format vecteur et format *raster* est essentielle pour le traitement informatique d'images [1]. Le format *raster*, ou *bitmap*, représente l'image par une matrice de pixels dont les valeurs se traduisent par des couleurs. Les photographies en sont l'exemple le plus courant. Le format vecteur représente des objets géométriques par leurs caractéristiques (points, segments, polygones) et leurs coordonnées.

Dans un usage géographique, le fichier *raster* ou vectoriel est plus qu'une image, il contient une information géoréférencée. Dans le cas de données *raster*, cette information est contenue dans les valeurs de la matrice de pixels ; dans le cas de données vectorielles, l'information intéressante peut être la géométrie des objets elle-même (la structure radioconcentrique d'un ensemble de lignes) mais il s'agit souvent des attributs attachés à ces objets (la population d'une commune).

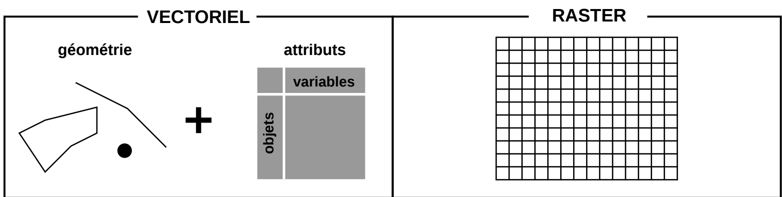


FIGURE 10.1 – Formats vecteur et *raster*

Plusieurs types de fichiers sont mobilisés dans cette section pour montrer les manipulations courantes dans un travail de cartographie : introduire des données externes stockées dans un tableau et avoir à gérer différents formats de données spatiales. Ici seuls deux formats courants sont importés, un format ESRI et un format MapInfo, mais la fonction

1. Le terme *image* est employé ici dans son acception du langage courant. Dans le domaine de la géomatique, *image* est synonyme de *raster*.

`readOGR()` permet d'importer tous les formats de la bibliothèque OGR (PostGIS, kml, Idrisi, etc.). Les fichiers nécessaires sont les suivants :

— **popCom3608.csv** : fichier des populations dans les communes de Paris et la petite couronne de 1936 à 2008 ;
— **parispc_com.shp** : limites communales de Paris et petite couronne, données vectorielles (polygones) au format ArcGIS constitué de quatre fichiers avec les extensions .shp, .dbf, .shx et .prj ;
— **parispc_bnd.shp** : contour de l'espace d'étude (Paris et petite couronne), données vectorielles (polygones) au format ArcGIS constitué de quatre fichiers avec les extensions .shp, .dbf, .shx et .prj ;
— **parispc_hop.tab** : établissements hospitaliers de Paris et petite couronne, données vectorielles (points) au format MapInfo constitué de deux fichiers avec les extensions .mif, .mid ;
— **Paris_dens.tif** : grille de population, données *raster* au format .tif.

10.1 R pour la cartographie

L'utilisation de R pour la cartographie n'est pas un choix évident. L'utilisateur qui souhaite faire une simple carte choroplèthe aura plus vite fait de la produire avec un logiciel de cartographie classique. L'utilisation de R pour faire de la cartographie se justifie si on envisage l'ensemble du flux de travail (*workflow*), c'est-à-dire l'intégration de la cartographie dans une chaîne de traitements.

On peut envisager (au moins) quatre situations dans lesquelles R s'avèrera avantageux. D'abord, pour obtenir rapidement des sorties massives : les logiciels de cartographie et de SIG sont des logiciels à interface graphique (même si certains permettent d'automatiser les traitements) et il devient rapidement coûteux de produire manuellement des ensembles de nombreuses cartes.

Ensuite, il sera intéressant d'utiliser R pour la cartographie dans le cadre d'un flux de travail unifié, c'est-à-dire pour intégrer la cartographie dans un flux de travail « classique » (analyse de données classiques, i.e. non spatiales) entièrement fait avec R et se passer ainsi d'importations et d'exportations parfois plus nombreuses que prévues, et qui sont source d'erreurs. En effet, il arrive fréquemment de faire des calculs dans un logiciel

de traitement de données, puis d'exporter les résultats pour les cartographier sur un autre logiciel, puis de refaire les calculs suite à une erreur ou une modification, puis de réexporter les résultats à cartographier, etc.

Unifier la chaîne de traitement est d'autant plus utile dans un flux de travail qui mobilise des fonctions d'analyse spatiale implémentées sous R. En effet, R dispose de plusieurs *packages* d'analyse spatiale et d'analyse de graphes (dont certaines sont abordées dans ce manuel) et il est très pratique de pouvoir manipuler des tableaux de données classiques, des objets spatiaux et/ou des graphes dans un flux de travail unifié.

Enfin, dans certains cas, l'utilisateur cherche à combiner les traitements et l'écriture des contenus : il s'agit d'intégrer des sorties numériques, graphiques et cartographiques dans des supports qui sont également écrits avec R (cf. 9.1). Dans ce cas, la cartographie, comme le reste des traitements, s'intègrera dans la production d'ensemble.

10.2 Prise en main des données spatiales

Certains *packages* spatiaux renvoient des messages d'avertissement lorsqu'on les charge, en particulier des messages sur les versions des bibliothèques externes utilisées : c'est le cas de `rgdal` qui, lors du chargement, donne une précision de ce type sur les bibliothèques GDAL et PROJ.4 [1].

```
library(sp)
library(rgdal)
library(rgeos)
library(mapproj)
library(maptools)
library(raster)
```

10.2.1 Construction d'un objet spatial

Dans cette première section, un objet spatial est créé à partir de zéro, il s'agira ici de deux sections de route (géométrie linéaire) accompagnées

1. Pour installer le *package* `rgdal` sur les systèmes Debian ou dérivés, il faut installer au préalable les bibliothèques de développement GDAL et PROJ.4, soit les paquets `libgdal-dev` et `libproj-dev`.

de deux attributs, la vitesse sur la section et le nombre de voies. Il est très rare qu'un utilisateur ait à créer lui-même des objets spatiaux, la plupart du temps il récupère les données produites par des organismes tels que l'IGN. Il est cependant important de comprendre comment un tel objet est construit.

Un objet spatial vectoriel est la combinaison d'une composante géométrique et d'une composante sémantique. La composante géométrique comporte des points avec des identifiants et des coordonnées. Ces points peuvent être accompagnés d'un attribut de regroupement s'ils forment des géométries plus compliquées, comme des lignes ou des polygones (cf. Section 9.3.5). La composante sémantique est un tableau relié aux géométries par un identifiant et qui contient des attributs correspondant aux objets géométriques.

L'exemple qui suit reproduit une structure de données fréquente pour les données spatiales linéaires telles que les routes ou les voies ferrées. La route est composée d'un ensemble de sections avec des attributs variables (type de route, vitesse, nombre de voies, etc.), chaque section est composée d'un ensemble de lignes, chaque ligne est composée d'un ensemble de points.

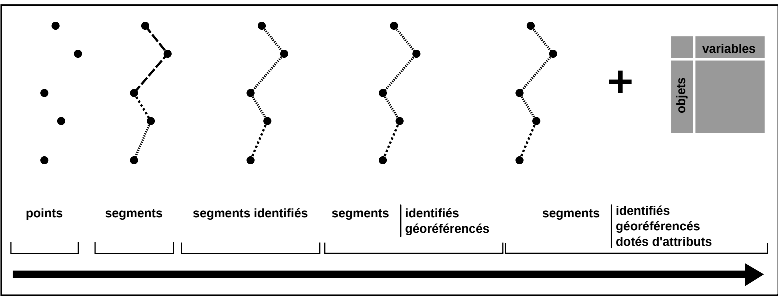


FIGURE 10.2 – Construction d'un objet linéaire

Dans un premier temps, trois tableaux sont créés qui contiennent des points avec des coordonnées.

```
coordPoints1 <- data.frame(id = c("P1", "P2", "P3"),
                           coordx = c(1, 1, 2),
                           coordy = c(1, 2, 2))

coordPoints2 <- data.frame(id = c("P3", "P4", "P5"),
                           coordx = c(2, 3, 3),
                           coordy = c(2, 2, 1))

coordPoints3 <- data.frame(id = c("P5", "P6", "P7"),
                           coordx = c(3, 4, 5),
                           coordy = c(1, 1, 1))
```

Dans un deuxième temps, les trois tableaux sont utilisés pour créer trois lignes avec la fonction `Line()` du *package* `sp`. Ces lignes ne contiennent pour le moment rien d'autre qu'un ensemble de points avec leurs coordonnées.

```
line1 <- Line(coordPoints1[ , 2:3])
line2 <- Line(coordPoints2[ , 2:3])
line3 <- Line(coordPoints3[ , 2:3])
```

Dans un troisième temps, les trois lignes sont regroupées dans des sections avec la fonction `Lines()` et un identifiant leur est attribué. Les deux objets `section1` et `section2` seront les formes géométriques (*features*) de l'objet spatial. La géométrie finale de l'objet spatial est créée avec la fonction `SpatialLines()` qui contient toutes les formes géométriques ainsi qu'un système de projection (ce point est détaillé dans la section suivante).

```
section1 <- Lines(slinelist = list(line1, line2),
                  ID = "Sec1")

section2 <- Lines(slinelist = list(line3),
                  ID = "Sec2")

geoLines <- SpatialLines(list(section1, section2),
                         CRS("+init=epsg:2154"))
```

Enfin, une table attributaire est créée avec deux champs, la vitesse et le nombre de voies. Ce tableau a deux lignes correspondant aux deux formes

géométriques. Le lien se fait entre l'identifiant des géométries et les noms de ligne du tableau. La fonction `SpatialLinesDataFrame()` crée l'objet spatial final qui combine la composante géométrique et la composante sémantique.

```
attrTable <- data.frame(SPEED = c(30, 50),
                        LANES = c(2, 3))

row.names(attrTable) <- c("Sec1", "Sec2")

geoDataLines <- SpatialLinesDataFrame(sl = geoLines,
                                      data = attrTable,
                                      match.ID = TRUE)
```

L'objet spatial sans composante sémantique (**geoLines**)peut déjà être manipulé et cartographié. L'objet spatial final contient une information sémantique (**geoDataLines**), il peut être cartographié en utilisant les variables de la table attributaire, en traduisant la variation des attributs numériques par des variations visuelles, de taille ou de couleur par exemple.

```
plot(geoDataLines,
     lwd = geoDataLines@data$SPEED,
     axes = TRUE)
```

Dans cet exemple, un objet spatial de type vectoriel et linéaire a été construit en assemblant des objets géométriques avec des attributs sémantiques. Dans un flux de travail réel, il est plus fréquent de travailler avec des objets spatiaux déjà construits sur lesquels il faut pouvoir extraire et manipuler des éléments géométriques et/ou sémantiques.

Dans les chapitres précédents, on a vu que certains types d'objets, *data.frame* ou *list* par exemple, sont composés de plusieurs éléments (*slots*) auxquels on accède avec l'opérateur `$`. Les objets spatiaux de type vectoriel comportent deux niveaux d'éléments : on accède au premier avec l'opérateur `@` et au second avec l'opérateur `$`.

Les principaux éléments de premier niveau sont : `@data` qui contient la table attributaire, `@lines` (`@points` ou `@polygons` selon le type de géométrie) qui contient la géométrie des objets et `@proj4string` qui contient l'indication du système de projection. Ensuite, il est possible avec

l'opérateur `$` d'accéder à un champ de la table attributaire de la même façon qu'à un champ de tableau classique. Lors de l'utilisation des opérateurs `@` et `$`, ne pas oublier la touche `Tab` qui complète automatiquement les noms des objets (cf. Section 1.3).

```
geoDataLines@data$TYPE <- "Highway"
attrTable <- geoDataLines@data
```

Pour récupérer et manipuler la composante géométrique, plusieurs fonctions sont disponibles, en particulier `bbox()` qui renvoie l'étendue spatiale des données (*bounding box*) et `coordinates()` qui renvoie les coordonnées des objets géométriques. Si l'objet spatial est ponctuel, la fonction renvoie les coordonnées de ces points ; si l'objet est linéaire, elle renvoie les coordonnées des points qui forment les lignes ; si l'objet est de type polygonal, elle renvoie les centroïdes des polygones.

```
bboxGeolines <- bbox(geoDataLines)
coordPoints <- coordinates(geoDataLines)
```

Cette courte introduction a montré comment construire et manipuler un objet spatial à partir d'un exemple très simple. Les sections qui suivent poursuivent l'étude des communes de Paris et de petite couronne. Deux types de géométries y sont manipulées : des points et des polygones.

10.2.2 Importation et prise en main des objets spatiaux

La fonction `readOGR()` du *package* `rgdal` est utilisée pour charger les données **Communes** (format ArcGIS) et **Hopitaux** (format MapInfo). Cette fonction importe les données et les stocke dans un objet de type *spatial*, selon leur géométrie : *SpatialPolygonsDataFrame*, *SpatialLinesDataFrame* ou *SpatialPointsDataFrame*.

La fonction `readOGR()` peut lire des formats contenant plusieurs couches, c'est pourquoi il ne suffit pas de préciser le nom du fichier (`dsn`), mais également le nom de la couche (`layer`). Dans le cas des fichiers ESRI (shp) et MapInfo (mif), ils ne contiennent qu'une seule couche (du même nom que le fichier) qu'il faut préciser avec l'argument `layer`.

L'argument (`encoding`) permet de préciser l'encodage, de la même façon que dans l'importation de simples tableaux (cf. Section 2.2.3), et l'argument `StringsAsFactors` sert à spécifier que les champs alphanumériques ne doivent pas être transformés en champs de type *factor*. Pour importer l'image *raster*, la fonction `readGDAL()` du *package* `raster` est utilisée.

```
muniBound <- readOGR(dsn = "data/parispc_com.shp",
                     layer = "parispc_com",
                     encoding = "utf8",
                     stringsAsFactors = FALSE)

regBoundary <- readOGR(dsn = "data/parispc_bnd.shp",
                       layer = "parispc_bnd",
                       encoding = "utf8",
                       stringsAsFactors = FALSE)

publicHosp <- readOGR(dsn = "data/parispc_hop.MIF",
                      layer = "parispc_hop",
                      encoding = "latin1",
                      stringsAsFactors = FALSE)

densGrid <- readGDAL("data/paris_dens.tif")
```

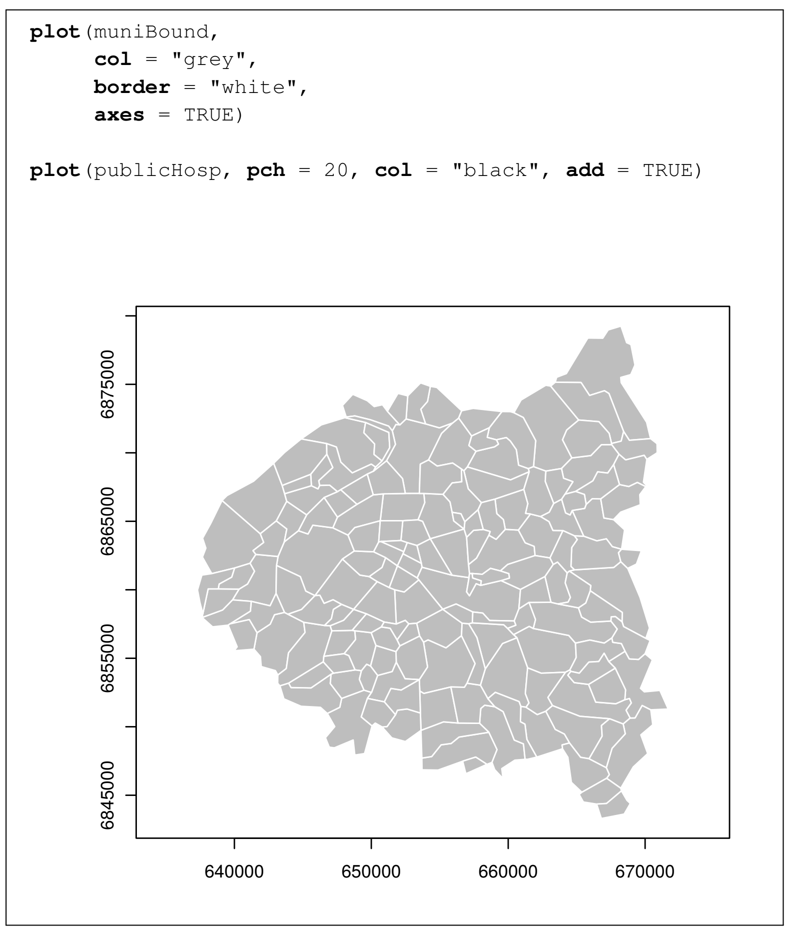
Lors de l'importation des fichiers, certaines informations sont renvoyées : le *driver* utilisé (ESRI et MapInfo dans le cas présent), le nombre d'objets (*features*) et de variables (*fields*) pour les fichiers vectoriels, le type de géométries (points et polygones dans cet exemple), la taille de la matrice de pixels dans le cas du fichier *raster* (392*380 dans ce cas).

Il est intéressant d'examiner le type et la structure de ces données spatiales grâce aux fonctions `class()` et `str()`. La fonction `str()` renvoie les attributs, le type de données spatiales, elle précise l'étendue des données (*bbox*) et le système de coordonnées (*proj4string*).

```
class(muniBound)
class(publicHosp)

str(muniBound)
str(publicHosp)
```

Une fois les données importées, elles peuvent être visualisées avec la fonction `plot()`. Cette fonction générique s'utilise de la même façon pour les données spatiales que pour les données classiques : l'argument `col` précise la couleur de remplissage, l'argument `border` la couleur du trait, l'argument `pch` le type de point, et finalement l'argument `add` pour superposer chaque nouvelle couche à la précédente.

```
plot(muniBound,
     col = "grey",
     border = "white",
     axes = TRUE)

plot(publicHosp, pch = 20, col = "black", add = TRUE)
```



Pour visualiser le *raster*, on commence par créer une palette de couleurs (cf. Section 9.2) qui est donnée comme argument de la fonction `image()` du *package* `raster`. Pour travailler directement sur les va-

leurs des pixels, il est possible de transformer l'objet *raster* (*SpatialGridDataFrame*) en une matrice de pixels avec la fonction `as.matrix()`.

```
colPalHeat <- c("white", rev(heat.colors(12)))
image(densGrid, col = colPalHeat)
plot(muniBound, add = TRUE)

pixGrid <- as.matrix(densGrid)
```

La conception de l'échelle et de la légende est détaillée dans la suite du chapitre.

10.2.3 Le système de projection

Jusqu'à présent, les données spatiales ont été importées et visualisées sans se soucier de leur système de projection qui est pourtant essentiel lorsqu'on manipule des données spatiales. Le système de projection est la transformation mathématique qui assure le passage d'une portion d'espace terrestre en trois dimensions à une représentation en deux dimensions. Cette transformation se traduit nécessairement par une déformation, ce qui a un impact en termes de visualisation (cartographie) et en termes de calculs de surfaces ou de distances (statistique spatiale).

Il existe un ensemble de normes d'identification des systèmes de projection. Le code qui identifie le système de projection suit un standard international (SRID - *Spatial Reference System Identifier*) et le standard le plus répandu est le code EPSG (*European Petroleum Survey Group*). Pour connaître le système de projection des objets spatiaux, la fonction `proj4string()` est utilisée, elle renvoie une information contenue dans les formats de fichiers classiques. Par exemple, pour un format ESRI shape, la projection est donnée dans le fichier d'extension .prj.

```
proj4string(muniBound)
```

Le résultat obtenu est une chaîne de caractères contenant plusieurs informations, principalement le système de projection et l'étendue des données (*bounding box*). Ici, le système est de type *Lambert Conformal Conic*

(lcc) et l'étendue est donnée par les coordonnées géographiques de deux points définissant la « boîte ». Il est utile de trouver le code qui identifie cette projection dans la liste de codes EPSG.

Certains sites web proposent une telle fonctionnalité [1] en récupérant l'information contenue dans le fichier d'extension .prj du fichier ESRI (*shapefile*) d'origine. Ceci dit, l'information et la modification du système de projection peuvent se faire intégralement avec R. La projection de l'objet **muniBound** semble mal référencée et sera convertie en « RGF93-Lambert93 » qui est le système de référence proposé par l'IGN [2].

Pour repérer le code correspondant à cette projection, il faut utiliser la fonction `make_EPSG()` du *package* `rgdal` qui renvoie un *data.frame* contenant l'ensemble des 4211 systèmes de projection avec leurs identifiants. Une sélection est effectuée sur ce tableau grâce à la fonction `grep()`, qui recherche les expressions régulières, c'est-à-dire les chaînes de caractères répétés (motifs). En recherchant tous les systèmes qui contiennent la chaîne de caractères « RGF » (Référentiel Géodésique Français), on obtient un tableau de 14 lignes qui contient le code EPSG correspondant au système voulu : il s'agit du code **2154**.

```
projCodes <- make_EPSG()
selecRgf <- projCodes[grep("RGF", projCodes$note), ]
```

Il s'agit maintenant de définir le système de projection des données spatiales, limites communales et hôpitaux, grâce à la fonction `spTransform()` qui ajoute au système initial le nouveau code EPSG. Avec la fonction `proj4string()` utilisée plus haut, on vérifie que le code EPSG a bien été modifié.

```
muniBound <- spTransform(x = muniBound,
                         CRSobj = CRS("+init=epsg:2154"))

publicHosp <- spTransform(x = publicHosp,
                          CRSobj = CRS("+init=epsg:2154"))
```

1. Voir par exemple le site `http://prj2epsg.org`.
2. Le site de l'IGN est riche en informations sur la géodésie : `http://geodesie.ign.fr/index.php?page=geodesie`.

10.2.4 Exportations

Au cours du travail cartographique, on peut souhaiter exporter deux types d'objets : des résultats cartographiques (des images) ou bien des objets spatiaux. Les dispositifs graphiques pour exporter les résultats sous forme d'images sont présentés dans la Section 9.1.

Pour exporter des objets spatiaux, on peut utiliser la fonction `writeOGR()`, pendant de la fonction `readOGR()` utilisée précédemment, et créer des fichiers dans les formats les plus courants, ESRI ou MapInfo par exemple. Le site de la bibliothèque GDAL/OGR [1] recense tous les formats disponibles. Dans l'exemple qui suit, le fichier de communes est exporté au format ESRI (.shp) dans un dossier intitulé « MyFolder ».

```
writeOGR(muniBound,
         dsn = "MyFolder",
         layer = "muniBound",
         driver = "ESRI Shapefile")
```

Si le dossier n'existe pas, il est créé au moment de l'exportation. Si seul le nom du dossier est donné sans préciser le chemin qui y mène, ce dossier sera créé dans le répertoire de travail (cf. Section 2.2.1).

10.3 Cartographie des objets ponctuels

Le fichier des hôpitaux est doté d'une variable quantitative de stock intitulée **Capacite**. Il s'agit de la capacité totale de l'hôpital mesurée en nombre de lits. Cette variable peut être traduite par une variable visuelle de taille pour produire une carte en cercles proportionnels.

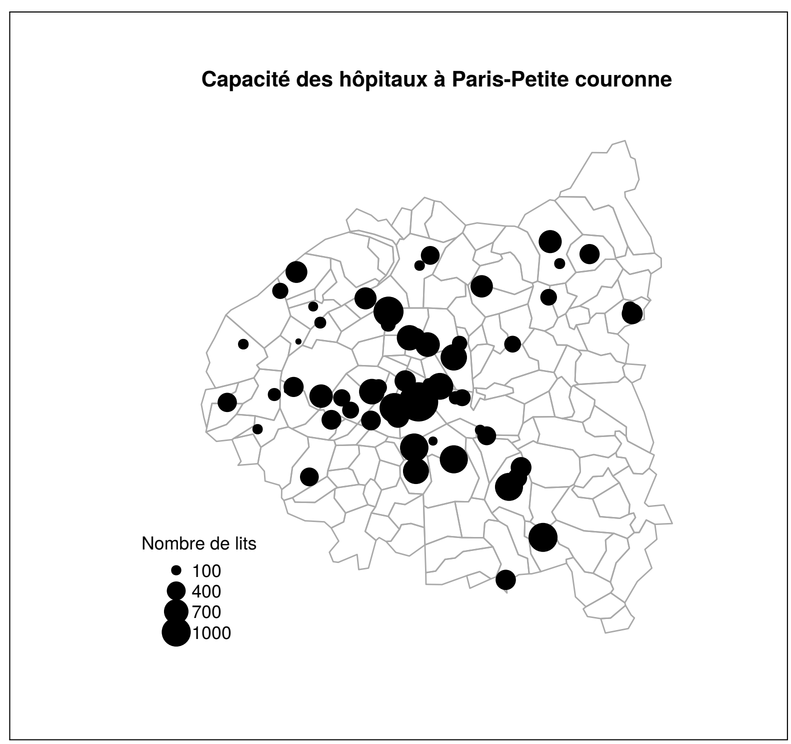
La fonction `plot()` est utilisée en faisant varier la taille des symboles (`cex`, *character expansion factor*) selon la capacité des hôpitaux. Par défaut, la variation de la valeur numérique est traduite par une variation proportionnelle du rayon du cercle. Ce paramétrage est corrigé : pour une visualisation apropriée, la proprotionnalité doit affecter l'aire du cercle et non son rayon.

1. `http://www.gdal.org`.

Il faut également ajouter une légende ainsi qu'une échelle. La légende est créée puis localisée sur le graphique soit par une localisation relative, soit en précisant les coordonnées x et y. Dans le premier cas, on indique la localisation par des arguments prédéfinis (*left*, *right*, *topleft*, etc.). Dans le second cas, on peut s'aider de la fonction `locator()` qui permet de récupérer les coordonnées d'un point sur le graphique en cliquant dessus.

Finalement, on utilise la fonction `legend()` en veillant à reprendre le style exact des symboles des hôpitaux choisis précédemment, notamment la forme, la couleur et la taille. Les arguments `pch`, `col` et `pt.cex` sont utilisées pour spécifier respectivement le type de figuré, la couleur et la taille des points, et l'argument `bty` (*box type*) permet de dessiner ou non un encadré de la légende.

```
plot(muniBound, border = "darkgrey")

plot(publicHosp,
     pch = 16,
     cex = 0.2 * sqrt(publicHosp$Capacite / pi),
     col = "black", add = TRUE)

title("Capacité des hôpitaux à Paris-Petite couronne")

legendHosp <- c(100, 400, 700, 1000)

legend("bottomleft",
       legend = legendHosp,
       pch = 16,  col = "black",
       pt.cex = 0.2 * sqrt(legendHosp / pi),
       bty = "n", title = "Nombre de lits")
```



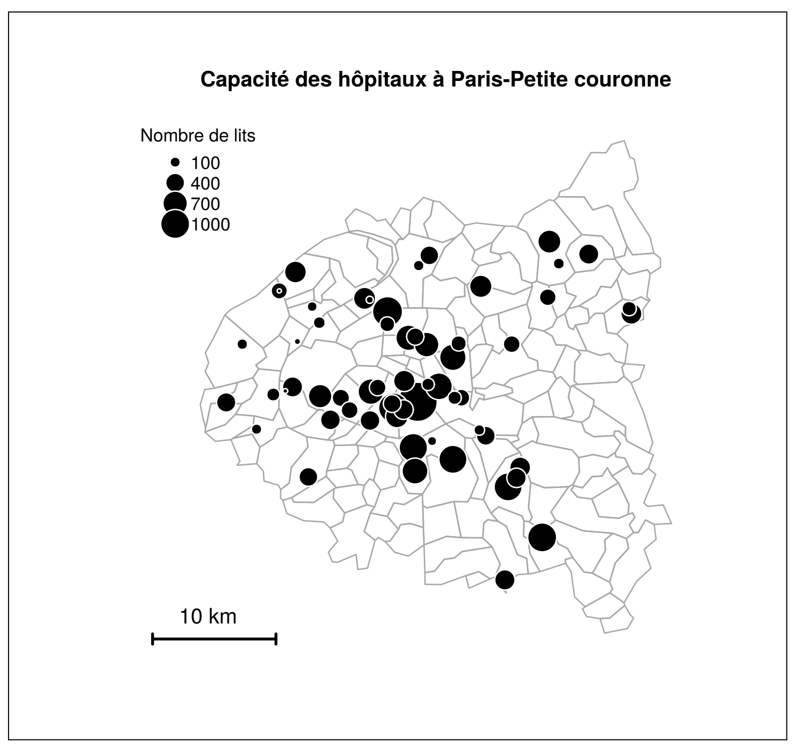
Pour ajouter une échelle, il y a deux solutions, qui demandent toutes deux de spécifier manuellement la localisation du trait, la longueur du trait, l'épaisseur et le texte. Dans le cas ci-dessous sont utilisées les fonctions `arrows()` pour le trait de l'échelle, puis `text()` pour le texte de l'échelle. Il est également possible de placer une échelle avec les fonctions `locator()` et `SpatialPolygonsRescale()`. La fonction `locator()` est interactive, elle demande de cliquer sur la carte pour obtenir les coordonnées du point où l'on souhaite placer l'échelle. Puis, la fonction `SpatialPolygonsRescale()` permet de placer l'échelle à partir de ce point.

Finalement, pour visualiser correctement les cercles proportionnels qui se chevauchent, il est nécessaire de spécifier une couleur de bordure qui les différencie (ici le blanc) et de trier les hôpitaux selon leur taille pour que les cercles les plus gros se retrouvent en arrière-plan.

Attention, l'argument `col` désigne deux choses différentes selon que l'on représente un objet zonal ou un objet ponctuel. Pour les objets zonaux, `col` désigne la couleur de remplissage des polygones et `border` la couleur du trait. Pour les objets ponctuels, `col` désigne la couleur du symbole, donc du trait, et l'argument `bg` (*background*) la couleur du fond, donc le remplissage du symbole ponctuel. Le type de point choisi (`pch`) est le numéro 21, parce qu'on cherche à distinguer la couleur de remplissage de la couleur de bordure. Il faut donc un type de symbole avec bordure, ce qui n'est pas le cas des types de symboles 15 à 20. On peut obtenir la liste des symboles ponctuels dans l'aide (`?pch`).

Le fonctionnement de la fonction `arrows()` nécessite une explication. Il s'agit de situer la droite représentant l'échelle par rapport aux marges du graphiques. La fonction `par()` permet de manipuler les paramètres graphiques généraux, l'argument `usr` est un vecteur de quatre valeurs : $x1$, $x2$, $y1$, $y2$ qui désignent les bornes de la sortie graphique. La fonction `arrows()` demande également quatre valeurs, les coordonnées x et y du premier point de la droite et les coordonnées x et y du second point de la droite. Ici le premier point de la droite a pour coordonnées $x1 + 1000$ unités (dans l'unité de mesure de l'objet spatial) et $y1 + 1000$ unités (désigne le coin en bas à gauche de la sortie graphique). Le second point de la droite a pour coordonnées $x1+10000$ unités (soit 10 km) et $y1+1000$ (bien sûr, y reste constant puisque la droite est horizontale). La fonction `text()` fonctionne de la même façon, en situant le point à partir duquel commence le texte et en précisant le contenu de ce texte (10 km).

```
publicHosp <- publicHosp[order(publicHosp$Capacite,
                                decreasing = TRUE), ]

plot(muniBound, border = "darkgrey")

plot(publicHosp,
     pch = 21,
     cex = 0.2 * sqrt(publicHosp$Capacite / pi),
     col = "white",
     bg = "black",
     add = TRUE)

title("Capacité des hôpitaux à Paris-Petite couronne")

legendHosp <- c(100, 400, 700, 1000)

legend("topleft",
       legend = legendHosp,
       pch = 21,
       col = "white",
       pt.bg = "black",
       pt.cex = 0.2 * sqrt(legendHosp / pi),
       bty = "n",
       title = "Nombre de lits")


arrows(par()$usr[1] + 1000,
       par()$usr[3] + 1000,
       par()$usr[1] + 10000,
       par()$usr[3] + 1000,
       lwd = 2, code = 3,
       angle = 90, length = 0.05)

text(par()$usr[1] + 5050,
     par()$usr[3] + 2700,
     "10 km", cex = 1.2)
```



Ce code semble exagérément long pour produire une simple carte de cercles proportionnels. Sa longueur vient du fait que chacun des éléments est paramétré de façon peu automatisée. Cependant, le code peut être encapsulé dans une fonction beaucoup plus simple pour un usage plus rapide. Le même résultat aurait aussi pu être produit avec le *package* `ggplot2` présenté dans le chapitre précédent.

10.4 Cartographie des objets zonaux

À la date de préparation de ce manuel, il existe principalement deux façons de produire des cartes avec R : avec la fonction générique `plot()` qui permet de représenter directement des objets spatiaux, ou bien avec la fonction `ggplot()` du *package* `ggplot2` très utile pour tous types de représentations graphiques. Ces deux méthodes sont expliquées ici ainsi

que leurs avantages et leurs inconvénients respectifs. L'exemple d'application est la production d'une carte choroplèthe de la densité de population par commune en 2008.

Pour chacune des deux fonctions `plot()` et `ggplot()`, deux critères de discrétisation sont testés : en quartiles et selon l'algorithme de Jenks. Dans ce dernier cas, le *package* `ClassInt`, est utilisé, comme dans le Chapitre 4.

L'algorithme de Jenks est très utilisé pour produire des cartes choroplèthes parce qu'il a été conçu dans ce but et qu'il est implémenté dans la plupart des logiciels de cartographie. Cependant, d'autres algorithmes existent pour discrétiser une variable continue selon des seuils dits « naturels » et ils sont souvent plus efficaces. Certains sont implémentés dans le *package* `ClassInt`, comme celui de Fisher ou les algorithmes de classification multivariée (k-means, CAH).

10.4.1 Cartographie avec la fonction `plot()`

On commence par créer des palettes de couleurs à partir du *package* `RColorBrewer`. Ce n'est pas la seule façon de construire une palette, mais ce *package* présente l'avantage de prédéfinir des palettes adaptées à différents types de représentations (cf. Section 9.2). La palette créée est une palette continue avec cinq couleurs de même ton et d'intensité variable (montée de valeurs).

```
library(RColorBrewer)
greyPal <- brewer.pal(n = 5, name = "Greys")
```

Ensuite, une jointure est faite entre les données attributaires de l'objet spatial et les données externes contenues dans le tableau **popCom3608**. Il y a deux précautions à prendre pour faire la jointure, l'une concerne la variable de jointure, l'autre concerne la fonction de jointure. La variable de jointure, ici le code de la commune, doit être du même type dans les deux tableaux. La fonction `class()` permet de s'en assurer. Concernant la fonction de jointure, la fonction `merge()` présentée dans la Section 2.4.1 peut être source d'erreur. Il faut en effet s'assurer que la table attributaire n'est modifiée ni dans l'ordre des lignes ni dans le nombre de lignes. Si, au

cours de la jointure, la table attributaire bouge (l'ordre des lignes change par exemple) la correspondance entre la composante sémantique (*i.e.* la table attributaire) de la couche à cartographier et sa composante géométrique n'est plus assurée.

La fonction `match()` garantit cette intégrité de la jointure. Les lignes qui suivent se lisent de la façon suivante : la table attributaire finale est la combinaison de la table attributaire initiale et du tableau externe (**popCom3608**). Les lignes du tableau externe sont ajoutées quand il y a correspondance entre les identifiants. L'ajout conserve l'ordre et le nombre de lignes du premier argument de la fonction `match()`, à savoir la table attributaire de l'objet spatial.

```
str(muniBound@data$INSEE_COM)
muniBound$INSEE_COM <- as.character(muniBound$INSEE_COM)

muniBound@data <- data.frame(
  muniBound@data,
  popCom3608[match(muniBound$INSEE_COM,
                   popCom$CODGEO), ]
)
```

Ce code nécessaire à la jointure est difficilement lisible et il sera fastidieux de le ré-écrire à chaque jointure. Il serait intéressant de l'encapsuler dans une fonction (cf. Section 3.3) pour pouvoir faire des jointures plus facilement. Cette fonction de jointure sur la table attributaire est nommée `AttribJoin()`, ses arguments sont le tableau externe (`df`), l'objet spatial (`spdf`) et les variables de jointure correspondantes (`df.field` et `spdf.field`).

```
AttribJoin <- function(df, spdf, df.field, spdf.field) {
  if(is.factor(spdf@data[ , spdf.field]) == TRUE) {
    spdf@data[ , spdf.field]  <- as.character(
      spdf@data[ , spdf.field]
    )
  }
  if(is.factor(df[ , df.field]) == TRUE) {
    df[ , df.field]  <- as.character(df[ , df.field])
  }
  spdf@data <- data.frame(
    spdf@data,
    df[match(spdf@data[ , spdf.field], df[ , df.field]), ]
  )
  return(spdf)
}
```

La fonction de jointure commence par tester le type des variables d'entrée : s'il s'agit de facteurs, ils sont transformés en champs alphanumériques. La fonction de jointure est la fonction `match()` présentée plus haut. La fonction qui vient d'être créée est simple et générique, elle peut être utilisée sur n'importe quel type de données spatiales.

```
muniBound <- AttribJoin(df = popCom3608,
                        spdf = muniBound,
                        df.field = "CODGEO",
                        spdf.field = "INSEE_COM")
```

Une fois la jointure effectuée, on peut travailler directement sur les données attributaires, les discrétiser et les représenter. Un vecteur de seuils (*breaks*) est créé avec la fonction `quantile()`. Celle-ci découpe la variable selon les probabilités données en argument avec la fonction `seq()`, qui correspondent ici à des quintiles.

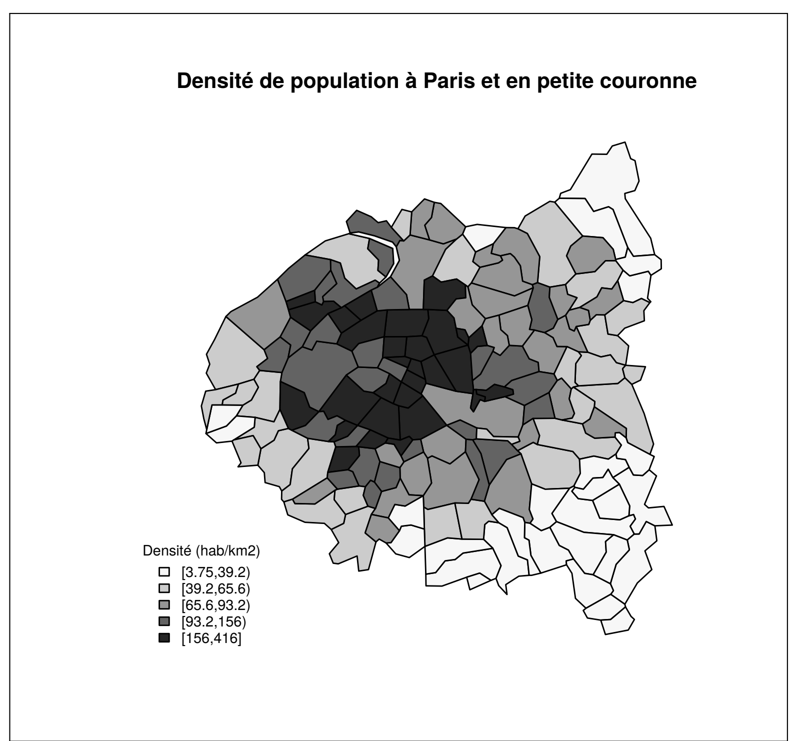
La discrétisation et l'assignation de la palette de couleurs se font dans la même étape, avec la fonction `cut()`. Cette fonction renvoie un objet de type *factor*, qui désigne chaque catégorie par un entier (ici de 1 à 5) et un label (étiquette de valeur) qui prend ici la couleur de la palette créée précédemment. L'argument `break` est utilisé pour préciser les seuils, l'argument `labels` pour préciser les étiquettes des valeurs, les arguments `include.lowest` et `right` pour créer des intervalles fermés à gauche et ouverts à droite de type [valeur1 ; valeur2[.

Finalement la fonction `as.character()` transforme le facteur en un vecteur alphanumérique qui contient la couleur correspondant à chaque catégorie. Le nouveau champ créé (**DENSQ5**) dans la table attributaire contiendra donc le code couleur utilisé par la suite avec la fonction `plot()`. La légende est construite de la même façon, la différence étant dans la fonction `levels()` qui renvoie la liste de modalités de la variable. Les modalités sont ici les intervalles de discrétisation.

```
quintBrks <- quantile(muniBound$DENSITE,
                      probs = seq(0, 1, 0.2),
                      names = TRUE)

muniBound$DENSQ5 <- as.character(cut(muniBound$DENSITE,
                                     breaks = quintBrks,
                                     labels = greyPal,
                                     include.lowest = TRUE,
                                     right = FALSE))

legendQ5 <- as.character(levels(cut(muniBound$DENSITE,
                                    breaks = quintBrks,
                                    include.lowest = TRUE,
                                    right = FALSE)))
```

La représentation cartographique proprement dite se fait comme dans l'exemple de carte en cercles proportionnels de la section précédente. La variable créée plus haut contient des étiquettes de couleurs : en précisant cette variable dans l'argument `col`, elle devient une variable visuelle.

```
plot(muniBound, col = muniBound$DENSQ5, border = "black")

legend("bottomleft",
       legend = legendQ5,
       bty = "n",
       fill = greyPal,
       cex = 0.8,
       title = "Densité (hab/km2)")

title("Densité de population à Paris et en petite couronne")
```



La même manipulation peut être effectuée avec un autre algorithme de discrétisation, celui de Jenks par exemple. Il suffira de substituer le vecteur de seuils (*breaks*) en quintiles par un autre vecteur de seuils.

```
jenksDiscret <- classIntervals(var = muniBound$DENSITE,
                               n = 5,
                               style = "jenks")

jenksBrks <- jenksDiscret$brks
```

10.4.2 Cartographie avec la fonction `ggplot()`

La fonction `ggplot()` du *package* `ggplot2` permet de personnaliser les aspects de la représentation plus finement que la fonction `plot()`. Elle a aussi certains inconvénients : elle ne prend pas comme argument un

objet de type spatial, il faut donc passer par une transformation préalable expliquée par la suite.

L'usage de ggplot() pour la cartographie se justifie quand le résultat graphique attendu est trop complexe pour pouvoir être fait avec plot() et quand les données spatiales ne sont pas trop lourdes. En effet, la transformation des données et la fonction de représentation sont plus lentes que la fonction plot(). Pour le présent jeu de données, composé de 143 communes et arrondissements, cela ne pose pas de problème, mais il est par exemple impossible de faire, sur un ordinateur de bureau, des cartes des 36 000 communes françaises avec ggplot().

En premier lieu, la variable de densité est discrétisée selon la même méthode que dans la section précédente (quintiles). En revanche, la variable est discrétisée dans le tableau externe (**popCom3608**) et non dans le tableau de données attributaires de l'objet spatial.

```
popCom3608$DENSQ5 <- cut(
  popCom3608$DENSITE,
  breaks = quintBrks,
  include.lowest = TRUE,
  right = FALSE,
  labels = c("Q1", "Q2", "Q3", "Q4", "Q5")
)
```

Pour visualiser les communes avec ggplot(), il faut transformer l'objet spatial (*SpatialDataframe*) en un objet que ggplot() puisse lire, à savoir un *data.frame* classique. La fonction fortify() du *package* ggplot2 effectue ce travail : pour un objet à géométrie polygonale, comme ici le fond communal, l'objet résultant est un tableau contenant les sommets des polygones avec leurs coordonnées (long, lat) ainsi qu'une variable de regroupement qui identifie le polygone avec les sommets le constituant. La représentation graphique dessinera les polygones en regroupant ces sommets, de la même façon que dans l'exemple présenté au chapitre précédent (cf. Section 9.3.5).

C'est à cause de cette transformation du format spatial au format « fortifié » qu'il faut nécessairement discrétiser la variable au préalable et non sur l'objet fortifié. En effet, dans cet objet un polygone est représenté par n lignes du tableau, n étant le nombre de sommets le constituant. Si la

discrétisation était faite sur ce tableau, la valeur de la variable associée à chaque polygone serait pondérée par le nombre de sommets, ce qui fausserait le résultat.

Dans un deuxième temps, une jointure est effectuée entre l'objet fortifié et le tableau externe contenant la variable discrétisée. À la différence de la jointure de la section précédente, la fonction merge() peut être utilisée ici, à condition de préciser les arguments all et sort. Cette jointure est l'équivalent d'un *left join* dans le langage SQL.

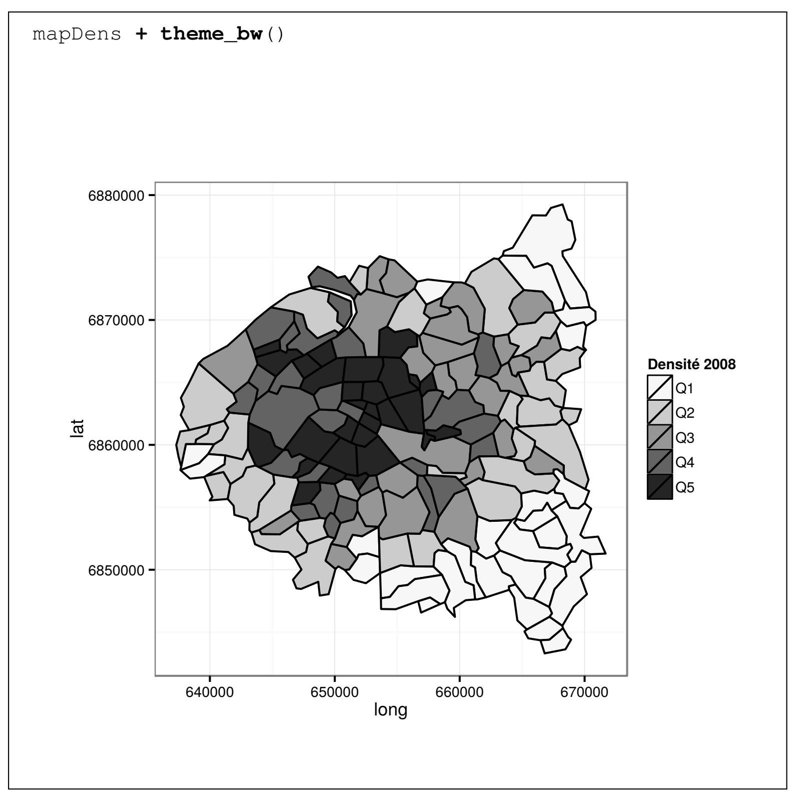
```
fortMuni <- fortify(muniBound, region = "INSEE_COM")

fortMuni <- merge(x = fortMuni,
                  y = popCom3608,
                  by.x = "id",
                  by.y = "CODGEO",
                  all.x = TRUE,
                  sort = FALSE)
```

La fonction geom_polygon() sert à graphier des objets zonaux : l'argument group désigne l'identifiant des polygones et l'argument fill désigne la couleur de remplissage, contenue dans le champ DENSQ5 créé précédemment. Finalement on modifie l'objet graphique (mapDens) avec la fonction coord_equal() qui assure la conservation de la même échelle *x* et *y* et avec la fonction scale_fill_brewer() qui permet de donner un titre à la légende et de désigner une palette de couleurs.

```
mapDens <- ggplot(fortMuni) +
  geom_polygon(aes(x = long,
                   y = lat,
                   group = group,
                   fill = DENSQ5),
               colour = "black") +
  scale_fill_manual(name = "Densité 2008",
                    values = greyPal) +
  coord_equal()
```

L'objet **mapDens** peut maintenant être visualisé, dans ce cas avec le thème défini par défaut theme_bw().

```
mapDens + theme_bw()
```



Il est également possible de définir un thème vide, sans grille, sans axes, sans titres d'axes, etc.

```
theme_update(axis.ticks = element_blank(),
             axis.text.x = element_blank(),
             axis.title.x = element_blank(),
             axis.text.y = element_blank(),
             axis.title.y = element_blank(),
             panel.grid.minor = element_blank(),
             panel.grid.major = element_blank(),
             panel.background = element_rect(),
             legend.position = "bottom")
```

10.5 Cartes *take away*

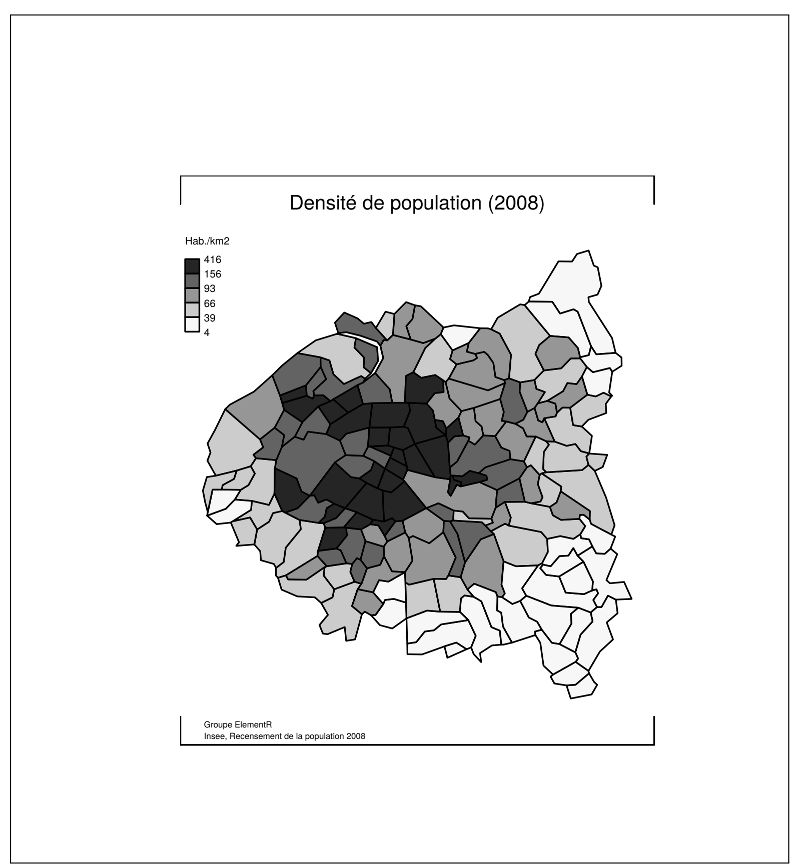
Il faut maintenant revenir aux constats posés en début de chapitre (cf. Section 10.1). Si l'on met dans la balance la longueur du code à écrire pour produire une simple carte, l'utilisation de R n'est pas toujours avantageuse vis-à-vis de logiciels à interface graphique. R est avantageux si on replace la sortie cartographique dans l'ensemble du flux de travail, et en particulier lorsqu'il faut produire des cartes à la chaîne.

Il reste cependant possible de produire rapidement une carte prête à l'emploi, prête à être intégrée dans un document par exemple. Le *package* `rCarto` présenté ci-dessous [1] travaille avec un fichier spatial au format ESRI (.shp) et un tableau (*data.frame*) existant dans l'espace de travail.

Deux fonctions permettent de reproduire les cartes présentées dans ce chapitre, `mapCircles()` pour la carte en cercles proportionnels et `mapChoropleth()` pour la carte choroplèthe. Voici en exemple qui reproduit la carte de la densité de population discrétisée en quintiles.

```
library(rCarto)
mapChoropleth(
  shpFile = "data/parispc_com.shp", shpId = "INSEE_COM",
  df = popCom3608, dfId = "CODGEO", var = "DENSITE",
  nclass = 5, palCol = "Greys", style = "quantile",
  posLeg = "topleft", lgdRnd = 0, legend = "Hab./km2",
  title = "Densité de population (2008)",
  author = "Groupe ElementR",
  sources = "Insee, Recensement de la population 2008")
```

1. *Package* développé par Timothée Giraud de l'UMS RIATE.



10.6 Cartographie à la chaîne

L'un des intérêts majeurs de faire de la cartographie avec R est de pouvoir faire des traitements automatisant la production de cartes, ce qui est coûteux à faire manuellement sur un logiciel à interface graphique. Il s'agit ici de comparer les densités à plusieurs époques (de 1936 à 2008), ce qui pose deux questions : la question du format de sortie et la question de la méthode de discrétisation. Concernant la première question, il y a au moins trois façons d'intégrer dans une seule sortie un ensemble de plusieurs cartes :

— afficher toutes les cartes sur une même page,
— afficher toutes les cartes sur plusieurs pages d'un même document,
— intégrer toutes les cartes dans une animation qui affiche chaque carte successivement.

L'exemple suivant montre la marche à suivre pour la première option. La deuxième option se fait avec les fonctions `pdf()`, `png()` ou autres suivant le format souhaité (cf. Section 9.1). Ces fonctions prendraient comme argument une boucle qui cartographie les données à chaque date du recensement.

La troisième option nécessite l'utilisation de *packages* spécifiques, par exemple `animation` qui permet d'intégrer toutes les cartes dans un format animé tel que HTML (fonction `saveHTML()`), GIF (fonction `saveGIF()`) ou autre.

Concernant la discrétisation, il faut choisir une méthode qui découpe la série selon des valeurs de centralité et de dispersion (moyenne, écart-type, quantiles, etc.) et non une méthode comme celle de Jenks qui ne s'appuie pas sur des résumés numériques fixes. On peut dès lors discrétiser chaque variable de densité de 1936 à 2008 séparément, ou bien discrétiser sur l'ensemble des valeurs de 1936 à 2008. C'est cette seconde option qui est montrée ici.

Concernant la fonction de représentation graphique, cette planche sera construite avec `ggplot()`, comme dans l'exemple présenté à la Section 9.3.4.

Dans un premier temps, les densités de population de 1936 à 2008 sont calculées. Ensuite, ces variables sont discrétisées ensemble. Il n'est pas nécessaire de joindre toutes ces données dans un seul champ pour calculer les quantiles : la fonction `quantile()` peut être appliquée à l'ensemble du tableau, à condition de le transformer en objet de type `matrix`, qui est un vecteur bidimensionnel.

```
evolDens <- popCom3608[ , 3:11] / popCom3608$SURF
evolQuint <- quantile(as.matrix(evolDens),
                      names = TRUE,
                      probs = seq(0, 1, 0.2))
```

Une fois créé le vecteur de seuils, chacune des colonnes du tableau est successivement discrétisée grâce à la fonction `cut()` appliquée colonne à colonne avec la fonction `apply()` (cf. Section 3.4). Le résultat final est transformé en tableau et une jointure est effectuée entre ce tableau et l'objet fortifié.

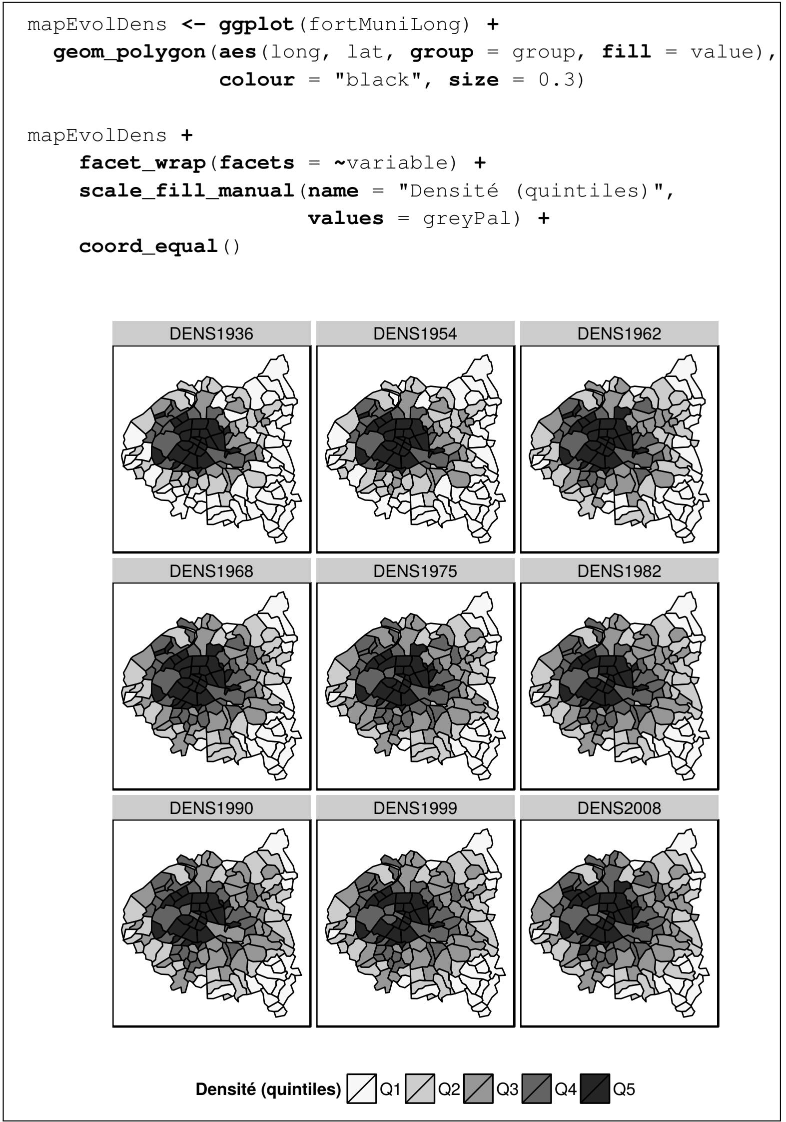
```
discrDens <- apply(evolDens, 2, cut,
                   breaks = evolQuint,
                   include.lowest = TRUE,
                   right = FALSE,
                   labels = c("Q1", "Q2", "Q3", "Q4", "Q5"))

discrDens <- as.data.frame(discrDens)
discrDens$CODGEO <- popCom3608$CODGEO

colnames(discrDens) <- c(
 "DENS1936","DENS1954","DENS1962","DENS1968","DENS1975",
 "DENS1982","DENS1990","DENS1999","DENS2008","CODGEO")

fortMuni <- merge(x = fortMuni,
                   y = discrDens,
                   by.x = "id",
                   by.y = "CODGEO",
                   all.x = TRUE,
                   sort = FALSE)
```

L'objet **fortMuni** est un tableau « large » contenant neuf champs de densité, un champ pour chaque année du recensement. Pour construire une planche en utilisant une variable désignant des sous-groupes de données (ici des années), il faudrait en faire une tableau « long » qui n'aurait que deux champs à cartographier : un champ contenant une variable qualitative indiquant l'année du recensement et un champ contenant la variable à représenter (ici des classes de densités). L'objet est transformé en format long avec la fonction `melt()` (cf. Section 2.4.3).

```
fortMuniLong <- melt(
  data = fortMuni,
  id.vars = colnames(fortMuni)[1:7],
  measure.vars = colnames(fortMuni)[22:30]
)
```

Cet objet peut être cartographié, en utilisant l'argument `facet_wrap` pour intégrer l'ensemble des cartes dans une seule sortie. Cette fonction constuire une planche de *n* cartes, une pour chaque modalité de la variable donnée comme argument (`facets`).

```
mapEvolDens <- ggplot(fortMuniLong) +
  geom_polygon(aes(long, lat, group = group, fill = value),
               colour = "black", size = 0.3)

mapEvolDens +
    facet_wrap(facets = ~variable) +
    scale_fill_manual(name = "Densité (quintiles)",
                      values = greyPal) +
    coord_equal()
```



Un rapide commentaire thématique s'impose : la population de l'agglomération parisienne, comme c'est le cas dans la plupart des grandes métropoles européennes, a tendu à s'étaler au cours du 20e. D'autres résultats concordants ont été obtenu dans les chapitres précédents : le cœur de l'agglomération, c'est-à-dire Paris *intramuros*, s'est dépeuplé durant la seconde moitié du 20e ; parallèlement, le peuplement de la proche banlieue s'accélérait.

Au terme de ce chapitre, l'utilisateur sait construire et manipuler des objets spatiaux, il sait travailler sur les tables attributaires et faire des jointures de données externes, il dispose enfin des outils pour construire différents types de cartes thématiques. Ces bases seront nécessaires pour aborder le chapitre suivant sur l'analyse spatiale.

CHAPITRE 11

Initiation aux statistiques spatiales

Objectifs : *Ce chapitre propose une initiation à des analyses fondées sur les localisations dans l'espace. Elles utilisent des calculs de statistiques spatiales utilisés dans l'analyse des localisations et organisations dans l'espace, mettant en relation proximités géographiques et ressemblances statistiques. Ces calculs peuvent également être utilisés à des fins d'exploration des relations dans l'espace.*

Prérequis Manipulation des objets spatiaux telle que présentée dans le Chapitre 9 et le Chapitre 10. Analyse des semis de points, notions d'autocorrélation spatiale.

Description des *packages* utilisés Les *packages* spécifiques pour la manipulation d'objets spatiaux sont présentés dans le chapitre précédent. Deux packages suplémentaires sont utilisés dans ce chapitre : `spdep` (*spatial dependence*) pour les statistiques spatiales et `ICSNP` (*Invariant Coordinate System - Non parametric*) pour le calcul du point médian.

Description des données utilisées Les données utilisées dans ce chapitre sont les mêmes que celles du chapitre précédent : le semis d'hôpitaux, les limites communales et les données socio-économiques attachées à ces communes.

11.1 Point moyen et point médian

Le chapitre 4 a présenté les principales mesures de centralité et de dispersion qui résument la distribution d'une variable quantitative. Parmi les mesures de centralité, la médiane et la moyenne sont les plus fréquemment utilisées. La médiane d'une variable est la valeur correspondant au deuxième quartile, elle découpe la distribution en deux sous-ensembles d'effectifs égaux. La moyenne d'une variable est la somme des valeurs divisée par l'effectif.

11.1.1 Centralité dans un espace à une dimension

La moyenne, comme la médiane, est une mesure de centralité parce que la somme des écarts entre cette valeur et toutes les valeurs prises par la variable est minimale. Plus précisément, la moyenne est la valeur qui minimise la somme du carré des écarts (1), la médiane est la valeur qui minimise la somme de la valeur absolue des écarts (2).

$$(1) \quad \arg\min \sum_{i=1}^{n} (x_i - \bar{x})^2 \qquad\qquad (2) \quad \arg\min \sum_{i=1}^{n} |x_i - \bar{x}|$$

Ces propriétés de la moyenne et de la médiane peuvent être testées sur une variable quantitative quelconque. Dans cet exemple, la variable utilisée est la capacité des hôpitaux (nombre de lits) de l'espace d'étude.

Dans un premier temps, cette variable est extraite de la table attributaire de l'objet spatial. L'histogramme, la moyenne et la médiane de la variable sont calculées. La distribution est dissymétrique à gauche, la moyenne est supérieure à la médiane.

```
hospCapacity <- publicHosp$Capacite
meanCapacity <- mean(hospCapacity)
medCapacity <- median(hospCapacity)
```

Dans un deuxième temps, une séquence de valeurs est créée qui encadre les valeurs centrales, il s'agit d'une suite d'entiers entre 0 et 1 000 avec un pas de 5. Ces 201 valeurs seront prises successivement comme valeur de référence : la somme des écarts entre chaque valeur de référence (x) et toutes les valeurs prises par la variable (x_i) sera calculée (écarts absolus et écarts au carré) avec une fonction de type `apply()` (cf. Section 3.4). L'ensemble est enfin regroupé dans un tableau, c'est-à-dire les séquences de valeurs avec un pas de 5, les 201 sommes de la valeur absolue des écarts, les 201 sommes du carré des écarts et le type correspondant à ces calculs (*absolute* et *square*).

```
seqVec <- seq(0, 1000, 5)

absDif <- sapply(
  seqVec,
  function(x, xi = hospCapacity) sum(abs(xi - x))
)

squDif <- sapply(
  seqVec,
  function(x, xi = hospCapacity) sum((xi - x) ^ 2)
)

difCurves <- data.frame(
  SUMDIF = c(absDif, squDif),
  SEQVAL = c(seqVec, seqVec),
  TYPE = c(rep("ABSOLUTE", length(absDif)),
           rep("SQUARE", length(squDif)))
)
```

Enfin, la variable est représentée graphiquement avec le *package* `ggplot2` (cf. Section 9.3) : la somme des écarts est représentée par une

courbe, les valeurs de la moyenne et de la médiane sont représentées par des lignes verticales. On vérifie ici les propriétés de ces deux mesures de centralité qui correspondent au minimum de chacune des deux fonctions.

```
ggplot() +
  geom_path(data = difCurves,
            aes(x = SEQVAL, y = SUMDIF, linetype = TYPE)) +
  geom_vline(xintercept = medCapacity, linetype = 1) +
  geom_vline(xintercept = meanCapacity, linetype = 2) +
  facet_wrap(~ TYPE, scales = "free") +
  xlab("Valeurs prises par la mesure de référence") +
  ylab("Somme des écarts (absolus ou au carré)") +
  theme_bw()
```

```
medCapacity

## [1] 395.5

seqVec[which.min(absDif)]

## [1] 395
```

Le raisonnement peut être appliqué d'un point de vue spatial dans un espace à une seule dimension. Le modèle de Hotelling en est un bon exemple : l'espace du modèle est une plage linéaire sans largeur où des

vacanciers sont répartis de façon aléatoire [1]. Sur cette plage, des vendeurs de glace cherchent à se rapprocher de leur clientèle potentielle. Le glacier qui veut maximiser ses opportunités de vente cherchera à se placer au point qui minimise la somme des distances à tous les vacanciers. Ce point est le point médian, il est facile à trouver dans un espace à une dimension mais il est plus difficile à approcher dans un espace à deux dimensions.

Dans un espace à deux dimensions, la moyenne et la médiane restent des mesures de centralité très utilisées pour décrire un semis de points. Le point moyen, ou centre de gravité, ou barycentre, est le point qui minimise le carré des distances à tous les autres points. Le point médian, quant à lui, minimise la somme des distances à parcourir. Ce point est la solution de problèmes simples de localisation, il est connu sous le nom de « point de Weber », du nom de l'économiste Alfred Weber (frère de Max Weber) qui l'a défini dans un travail sur les localisations industrielles. D'autre points centraux peuvent être utilisés, en particulier le point minimax qui minimise la distance au point le plus éloigné.

Dans la suite de cette section, le point moyen et le point médian sont calculés sur les données de Paris et de la petite couronne : le point médian sur le semis des hôpitaux et le point moyen sur le semis des communes. Le premier calcul répond à la question d'un hypocondriaque extrémiste qui veut se situer au plus près de l'ensemble des hôpitaux. Le second calcul permettra de déterminer le centre géométrique de l'espace d'étude d'abord, son centre démographique ensuite, en prenant en compte la population des communes, enfin il permettra de voir le déplacement de ce centre au cours du temps.

11.1.2 Point médian dans un espace à deux dimensions

Le point médian est d'abord approché par une méthode à la fois didactique et rudimentaire. Dans un premier temps les coordonnées des hôpitaux sont extraites de l'objet spatial grâce à la fonction `coordinates()`. Ensuite, l'étendue de l'espace d'étude (*bounding box*) est récupéré avec la fonction `bbox()`. Ces deux éléments serviront de base au calcul. La dernière ligne de code utilise la fonction

1. Il existe une description similaire de ce modèle avec des magasins à placer dans une rue, ce pourquoi il a parfois été qualifié de *linear city model*.

`fortify()` pour extraire la géométrie du polygone (contour de l'espace d'étude) et pouvoir la représenter avec le *package* `ggplot2` (cf. Section 9.3). Cette dernière étape n'est utile que pour la représentation cartographique, elle n'a pas de rôle à jouer dans le calcul du point médian.

```
coordHosp <- as.data.frame(coordinates(publicHosp))
colnames(coordHosp) <- c("COORDX", "COORDY")
bboxRegion <- bbox(regBoundary)
fortBoundary <- fortify(regBoundary)
```

Il s'agit ensuite de parcourir l'étendue de l'espace d'étude en s'arrêtant régulièrement pour prendre un point de référence et calculer la somme des distances entre ce point et l'ensemble des hôpitaux. Pour cela, une grille est créée, c'est-à-dire un semis de points régulier qui couvre l'étendue de l'espace d'étude. Pour créer cette grille, on parcourt l'étendue des x et l'étendue des y avec une séquence régulière (pas de 1 000) avec le fonction `seq()`, puis ces deux séquences sont combinées avec la fonction `expand.grid()`.

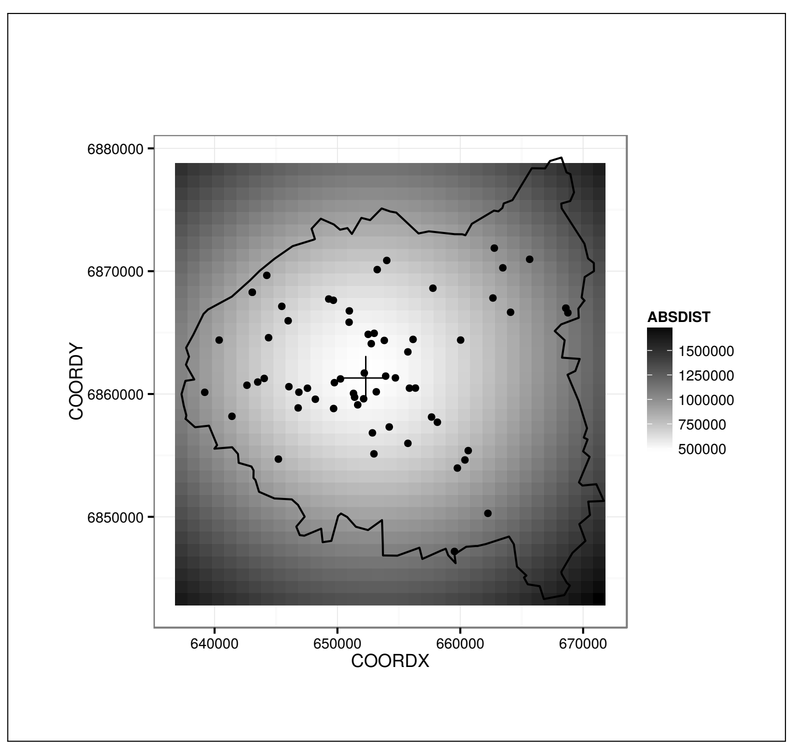
```
seqCoordX <- seq(floor(bboxRegion[1, 1]),
                 ceiling(bboxRegion[1, 2]),
                 by = 1000)

seqCoordY <- seq(floor(bboxRegion[2, 1]),
                 ceiling(bboxRegion[2, 2]),
                 by = 1000)

longGrid <- expand.grid(COORDX = seqCoordX,
                        COORDY = seqCoordY)
```

Finalement, une matrice de distances est calculée entre le semis de points des hôpitaux et le semis régulier (grille). C'est la fonction `rdist()` du *package* `fields` qui effectue ce travail. L'utilisateur pourrait envisager d'écrire lui-même une fonction qui calcule la matrice de distances, cependant une fonction de ce type est rapidement gourmande en calcul et il vaut mieux utiliser une fonction, telle que `rdist()`, qui est programmée en Fortran et qui sera bien plus rapide qu'une fonction programmée en R (cf. Section 3.3.1).

```
library(fields)
matDist  <- rdist(coordHosp, longGrid)
sumDist <- apply(matDist, 2, sum)
longGrid$ABSDIST <- sumDist
```

Le résultat est cartographié avec `ggplot2`. Quatre couches sont superposées : la première est une grille (*raster*) dont chaque pixel représente la somme des distances entre ce pixel et l'ensemble des points qui forment le semis des hôpitaux, la deuxième est une croix correspondant au point pour lequel cette somme est minimale (approximation du point médian), la troisième est le semis des hôpitaux, la quatrième est le contour de l'espace d'étude.

```
ggplot() +
  geom_raster(data = longGrid,
              aes(x = COORDX, y = COORDY, fill=ABSDIST)) +
  geom_point(data = longGrid[which.min(longGrid$ABSDIST),],
             aes(x = COORDX, y = COORDY),
             colour = "black", shape = 3, size = 10) +
  geom_point(data = coordHosp,
             aes(x = COORDX, y = COORDY),
             colour = "black", size = 2) +
  geom_polygon(data = fortBoundary,
               aes(x = long, y = lat, group = group),
               fill = NA, colour = "black") +
  scale_fill_gradientn(colours = c("white", "black")) +
  coord_equal() +
  theme_bw()
```



Il serait possible avec l'algorithme développé ci-dessus de donner une aproximation plus précise du point médian en fixant une résolution plus fine à la grille mais les calculs deviendraient rapidement très lourds. Il existe bien sûr des méthodes bien plus élégantes et plus efficientes pour calculer le point médian, l'algorithme le plus classique étant implémenté dans plusieurs *packages* de R, par exemple dans le *package* `ICSNP`.

```
library(ICSNP)
spatial.median(coordHosp)

##  COORDX  COORDY
##  652242 6861734
```

11.1.3 Point moyen dans un espace à deux dimensions

L'une des caractéristiques de la moyenne vis-à-vis de la médiane est d'être plus sensible aux variations de valeur, que ce soit dans un espace à une ou à deux dimensions. Cette caractéristique peut s'avérer intéressante lorsqu'on cherche à mettre en évidence des évolutions temporelles et le point moyen est souvent utilisé de la sorte.

Il s'agit dans l'exemple suivant de calculer le point moyen sur le semis des communes, du centroïde des communes plus précisément, mais en tenant compte du poids de chaque commune en termes de population. Ce qui est calculé est donc un point moyen pondéré par la population, et ce point moyen sera calculé pour chacune de dates auxquelles on dispose d'une variable de population entre 1936 et 2008. Les coordonnées de ce point moyen ou centre de gravité (x_g et y_g) se calculent en faisant la moyenne pondérée (w_i) des coordonnées des points (x_i et y_i)

$$x_g = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i} \qquad y_g = \frac{\sum_{i=1}^{n} w_i y_i}{\sum_{i=1}^{n} w_i}$$

Les coordonnées des centroïdes des communes sont extraites de l'objet spatial avec la fonction `coordinates()` et le code des communes est récupéré dans la table attributaire de cet objet. Ensuite, une jointure est faite avec le tableau **popCom3608** qui contient les variables de population à différentes dates.

```
coordMuni <- data.frame(X = coordinates(muniBound)[ , 1],
                        Y = coordinates(muniBound)[ , 2],
                        CODGEO = muniBound@data$INSEE_COM,
                        stringsAsFactors = FALSE)

coordPop <- merge(coordMuni, popCom3608,
                  by = "CODGEO",
                  sort = TRUE)
```

Pour calculer automatiquement les points moyens à chaque date, une boucle est mise en place. Les fonctions de type `apply()` sont souvent meilleures que les boucles pour appliquer une fonction successivement à un ensemble (cf. Section 3.4). Elles sont meilleures à la fois en termes

de vitesse d'exécution et en termes de lisibilité du code. Cependant, en termes de vitesse leur supériorité dépend en grande partie du nombre de cas auxquels la fonction est appliquée. Sur un très petit nombre comme ici (9 dates de recensement), ce différentiel de vitesse est nul. En outre, il semble plus facile dans ce cas d'écrire un code simple et lisible au travers d'une boucle.

Deux vecteurs vides sont créés pour stocker les valeurs successives des coordonnées x et y du centre de gravité. La boucle fera neuf itérations et i prendra successivement les valeurs entières comprises entre 5 et 13, ce qui correspond à l'index des colonnes contenant les populations à chaque date. À la fin de chaque itération, les valeurs des coordonnées seront ajoutées aux vecteurs de stockage.

```
xgEvol <- vector()
ygEvol <- vector()

for(i in 5:13){
 xgTemp <- weighted.mean(coordPop$X, w = coordPop[ , i])
 ygTemp <- weighted.mean(coordPop$Y, w = coordPop[ , i])
 xgEvol <- append(xgEvol, xgTemp)
 ygEvol <- append(ygEvol, ygTemp)
}
```

Les coordonnées du centre de gravité sont stockées dans un tableau qui peut maintenant être cartographié, avec la fonction générique `plot()` dans l'exemple ci-dessous. Les arguments utilisés ont déjà été vus dans les chapitres antérieurs : `type` pour le type de figuré (**p**oint, **l**ine, **b**oth), `pch` pour le type de point (*point character*) et `asp` pour le ratio d'échelle entre les axes x et y.

```
gravityEvol <- data.frame(YEAR = colnames(coordPop)[5:13],
                          XG = xgEvol,
                          YG = ygEvol)

plot(gravityEvol$XG, gravityEvol$YG,
     type = "b",
     pch = 20,
     asp = 1,
     xlim = c(652400, 654000))

text(x = gravityEvol$XG,
     y = gravityEvol$YG,
     labels = gravityEvol$YEAR,
     cex = 0.8, pos = c(rep(4, 4), rep(3, 2), 4, 1, 3))
```

Ce graphique peut être brièvement commenté : durant la première moitié du 20e siècle, l'Ouest de l'agglomération s'est peuplé plus rapidement

que l'Est, tendance qui s'est inversée durant la seconde moitié du siècle. Le centre de gravité pondéré par la population s'est donc déplacé vers l'Est durant la période 1936-2008, résultat cohérent avec les analyses faites à la Section 6.1. Ni le contour de l'espace d'étude, ni les limites municipales n'ont été ajoutées à cette carte parce que l'étendue de ce déplacement reste limitée : entre 1936 et 2008, le centre de gravité traverse le 4e arrondissement d'Est en Ouest.

11.2 Mesurer les espacements dans un semis de points

11.2.1 Notion de plus proche voisin et mesures associées

Lorsque l'on dispose d'un semis de points, la description de la structure de répartition du semis se fonde sur des mesures de l'espacement entre les points, résumés à l'ensemble du semis. Dans ces approches, la distance joue un rôle fondamental. Il y a plusieurs façons de mesurer l'éloignement entre deux points, c'est ici la façon la plus simple qui est utilisée : la distance euclidienne à vol d'oiseau. Pour certaines questions, il peut être nécessaire d'utiliser une distance plus appropriée.

À la base de la description des espacements entre points du semis, il faut calculer une matrice de distance entre tous les points, comme cela a été fait dans la section précédente. Il est courant de travailler non sur l'ensemble de la matrice mais sur la notion de distance au plus proche voisin. Cette distance peut aussi être généralisée à des ordres supérieurs comme par exemple la distance moyenne aux 3 ou aux 5 plus proches voisins.

La distance au plus proche voisin peut être utilisée dans un cadre descriptif pour décrire des contextes locaux. Elle peut aussi être utilisée dans une approche modélisatrice, par exemple comme variable explicative dans une régression multiple. Elle peut enfin être utilisée pour caractériser le semis de points en le comparant à une distance théorique obtenue par un processus ponctuel aléatoire, un processus de Poisson par exemple.

Le calcul de la distance moyenne au plus proche voisin peut être traité de deux façons :

— Calculer une matrice de distance entre tous les points ou distancier (cf. Section 3.1.2). Cette méthode a l'avantage de créer un objet complet à partir duquel un grand nombre d'indicateurs peuvent être calculés (distance au plus proche voisin, distance moyenne à tous les voisins). Elle a cependant l'inconvénient de créer un objet surdimensionné par rapport à l'utilisation que l'on peut en faire, objet qui peut être très lourd si le semis comporte un très grand nombre de points.
— Utiliser une suite de fonctions implémentées dans le *package* `spdep` spécialisé dans les statistiques spatiales. L'inconvénient de cette méthode est qu'il faut manipuler des types d'objets spécifiques avec des fonctions qui souvent ne servent qu'à passer d'un type d'objet à un autre.

11.2.2 Calcul d'une matrice de distance

Le distancier est calculé à partir des coordonnées contenues dans le tableau de données en utilisant la fonction `dist()`. Plusieurs distances sont proposées (distance de Manhattan, de Canberra, etc.), c'est ici la distance euclidienne qui est choisie. Utilisée comme suit, elle permet de construire le triangle inférieur hors diagonale d'une matrice de distance. Elle contient dans ce cas $n(n-1)/2$ éléments. Il faut ensuite transformer cet objet en un objet matrice $n \times n$ en utilisant la fonction `as.matrix()`.

```
distHosp <- dist(coordHosp, method = "euclidean")
matDistHosp <- as.matrix(distHosp)
```

On effectue ensuite le calcul de la distance au plus proche voisin pour chaque hôpital en ayant eu soin au préalable d'éliminer du calcul les éléments de la diagonale (en leur donnant la valeur `NA`), sans quoi chaque hôpital serait son plus proche voisin à une distance de 0.

```
matDistHosp[matDistHosp == 0] <- NA
minDist <- apply(matDistHosp, 1, min, na.rm = TRUE)
mean(minDist)

## [1] 1392
```

La valeur moyenne de la distance au plus proche voisin est donc de 1 392 mètres. Indépendamment de la méthode des plus proches voisins, on peut s'intéresser à différentes statistiques des distances de chaque hôpital aux autres, en effectuant des statistiques univariées sur chacune des distributions. Les lignes qui suivent permettent de créer un tableau à n lignes et 7 colonnes pour les différents indicateurs créés par la fonction `summary()`. Ce tableau peut être transposé et trié pour en faciliter la lecture.

```
summaryDist <- apply(matDistHosp, 1, summary, na.rm = TRUE)
head(t(summaryDist))

##   Min. 1st Qu. Median  Mean 3rd Qu.  Max. NA's
## 1 1180    4190   8060  8000   11300 16900    1
## 2  922    6020   8040  9130   11800 21400    1
## 3  324    4560   7510  8090   11200 18700    1
## 4 4130   13900  17500 16900   21300 26400    1
## 5 1450    5120   8110  9350   12700 19100    1
## 6 2020    5890   9440 10300   14100 18000    1
```

11.2.3 Fonctions spécifiques du *package* spdep

Le *package* `spdep` contient un ensemble de fonctions dédiées aux statistiques spatiales. Ces fonctions créent des objets spécifiques, par exemple des listes de voisins, qui doivent souvent être transformées pour alimenter d'autres fonctions du packages, par exemple pour représenter graphiquement les points et leurs voisins.

Dans un premier temps, la fonction `knearneigh()` permet de calculer de manière générique la liste des **k** plus proches (**near**) voisins (**neigh**bours) d'un ensemble. Lorsque cette fonction est appliquée à un objet spatial, la fonction récupère directement l'ensemble des informations nécessaires aux calculs et travaille sur le vecteur `$coords`. La proximité est ici entendue au sens de la distance euclidienne. Si on spécifie l'argument `longlat`, la fonction peut aussi faire le calcul sur des coordonnées géographiques (longitudes-latitudes) en degrés décimaux. Dans cet exemple, on commence par créer une liste de plus proches voisins d'ordre 5 (argument `k`).

```
library(spdep)
listNearNei <- knearneigh(publicHosp@coords, k = 5)
class(listNearNei)
```

L'objet qui vient d'être créé est une liste de type *knn* dont le premier élément est le vecteur des plus proches voisins. Ce schéma résume les types d'objets manipulés avec `spdep` et les fonctions qui assurent leur transformation.

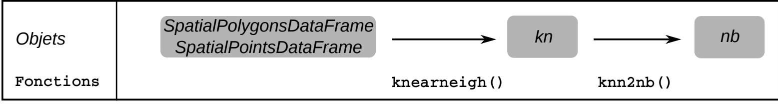


FIGURE 11.1 – Objets et fonctions du *package* `spdep`

Le premier élément de cet objet `knn` est, pour chaque hôpital, la liste des cinq hôpitaux les plus proches. À partir de cet objet, la distance est calculée entre chaque hôpital et ses cinq plus proches voisins grâce à la fonction `nbdists()`. Ce calcul ne peut être fait directement sur l'objet `knn` et il faut le transformer en objet de type `nb`. La fonction `nbdists()` demande également de préciser les coordonnées des hôpitaux contenues dans l'objet spatial (`@coords`).

```
nearNei <- knn2nb(listNearNei,
                  row.names = publicHosp$CodHop)

distNei <- nbdists(nearNei, publicHosp@coords)
class(distNei)
```

La structure de liste renvoyée par la fonction `nbdists()` est simplifiée avec la fonction `unlist()` qui crée un vecteur numérique regroupant toutes les distances. Ce vecteur donne la distance entre tous les couples d'hôpitaux voisins à un ordre 5. Cette variable peut être explorée avec les méthodes présentées dans le Chapitre 4.

```
distNei <- unlist(distNei)
summary(distNei)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0    1550    2450    2700    3890    9580

hist(distNei, breaks = 20,
     main = "Distance aux 5 plus proches voisins",
     col = "black",
     border = "white",
     xlab = "Distance",
     ylab = "Fréquence")
```

Distance aux 5 plus proches voisins

Fréquence
0 10 20 30 40 50
0 2000 4000 6000 8000 10000
Distance

Il est parfois intéressant de fixer une distance seuil et non un nombre fixe de voisins. Par exemple, en définissant un seuil de 2 km, il est possible de créer, pour chacun des points, une liste de voisins situés à une distance inférieure à deux kilomètres. La fonction `dnearneigh()` crée cette liste, à partir des coordonnées des points et des seuils de distance (arguments `d1` et `d2`), ici de 0 à 2000 mètres.

```
neiTwoKm <- dnearneigh(publicHosp@coords,
                       d1 = 0,
                       d2 = 2000,
                       row.names=publicHosp$CodHop)
```

Cette fonction renvoie directement un objet de type `nb`.

11.2.4 Visualisations des voisinages

Il est utile de compléter les mesures qui viennent d'être calculées avec des représentations cartographiques. Les objets de types nb, qui sont des listes de voisins, peuvent être représentés sous la forme de graphes et ajoutés à la cartographie du semis de points. L'exemple ci-dessous montre les réseaux de voisins à une distance inférieure à deux kilomètres.

```
plot(muniBound, axes = TRUE,
     col = "grey", border = "white")
plot(publicHosp, col = "black", pch = 20, add = TRUE)
plot(neiTwoKm, publicHosp@coords,
     pch = 20, col = "black", add = TRUE)
```

11.3 Autocorrélation spatiale et ressemblances locales

11.3.1 La notion d'autocorrélation spatiale

Les méthodes de statistique spatiale qui viennent d'être introduites s'appliquent à des semis de points. Il existe également toute une gamme de méthodes qui s'appliquent à des phénomènes observés dans des zones, par exemple dans des maillages territoriaux. L'autocorrélation spatiale en fait partie, elle sera travaillée ici sur le maillage communal.

L'autocorrélation spatiale consiste, pour un phénomène donné, à mesurer l'intensité de la relation entre la proximité des lieux et leur degré de ressemblance au regard de ce phénomène. Le fondement de ces mesures est la mise en rapport d'une variabilité locale avec la variabilité globale.

Les premiers développements de ces mesures concernaient des voisinages définis par la contiguïté sur un maillage. La variance locale peut alors s'écrire comme une variance globale pondérée par un poids w_{ij} affecté à chacun des couples (i, j) : $w_{ij} = 1$ si i et j sont des zones contiguës, $w_{ij} = 0$ sinon. Dans ce cas w est une matrice de contiguïté.

Ce principe peut être étendu à la notion de voisinage au sens large, en considérant comme voisines des unités spatiales distantes de moins de 5km, de moins d'une heure ou appartenant à la même région administrative. Dans ce cas, w_{ij} sera aussi une variable binaire et w une matrice de contiguïté. Il est enfin possible de pondérer les couples par l'inverse de la distance (ou du temps), auquel cas w_{ij} sera une variable continue et w une matrice de poids.

Les deux indices d'autocorrélation les plus anciens et les plus utilisés sont l'indice I de Moran (1950) et l'indice C de Geary (1954).

$$I = \frac{N}{\sum_i \sum_j w_{ij}} \frac{\sum_i \sum_j w_{ij}(X_i - \bar{X})(X_j - \bar{X}}{\sum_i (X_i - \bar{X})^2}$$

$$C = \frac{(N-1)}{\sum_i \sum_j w_{ij}} \frac{\sum_i \sum_j w_{ij}(X_j - \bar{X})}{\sum_i (X_i - \bar{X})^2}$$

Où N est le nombre d'individus, X_i et X_j sont les valeurs de la variable X en i et j, $\bar{X}$ est la moyenne de la variable X et w_{ij} est une pondération correspondant à la définition du voisinage.

Des développements ont ensuite été proposés pour travailler sur les contributions locales des zones à cette mesure de l'autocorrélation spatiale. Il s'agit des indicateurs locaux d'association spatiale (LISA - *local indicators of spatial autocorrelation*). L'indice local de Moran correspond ainsi à la contribution de chaque unité à l'autocorrélation globale.

Il s'agit, pour une unité spatiale i, du produit entre l'écart à la moyenne de cette unité spatiale (z_i) et la somme des écarts des unités voisines (z_j) pondérés par la matrice de contiguïté ou de poids (w_{ij}).

$$I_i = z_i \sum_j w_{ij} z_j$$

Cet indicateur permet d'identifier les situations localement homogènes (I_i faible), et les situations localement hétérogènes (I_i élevé). Selon le phénomène étudié, on cherchera à isoler l'une et/ou l'autre de ces situations.

11.3.2 Répartition des cadres et des ouvriers

L'exemple suivant cherche à caractériser la répartition spatiale des cadres et des ouvriers dans l'espace d'étude.

La fonction `moran.test()` du *package* `spdep` travaille directement sur les objets de type `nb` qui donnent la liste des unités voisines. Pour travailler sur un objet de ce type, il faut au préalable transformer l'objet spatial contenant le découpage communal (*SpatialPolygonsDataFrame*) en objet de type `nb`.

C'est la fonction `poly2nb()` qui effectue ce travail. Plusieurs arguments doivent être précisés : l'argument `queen` (déplacement de la reine au jeu d'échecs) indique qu'un seul sommet commun entre deux polygones est suffisant pour les considérer comme voisins ou qu'il faut un segment commun. L'argument `snap` permet de fixer la distance en dessous de laquelle deux points d'un polygone sont considérés comme étant

superposés. En effet, il arrive souvent de travailler avec des géométries de mauvaise qualité (erreurs de topologie), où les sommets formant des segments ne coïncident pas exactement.

Dans un premier temps, les variables du tableau externe **socEco9907** sont jointes aux données attributaires de l'objet spatial. Cette manipulation est faite avec la fonction `AttribJoin()` créée dans le chapitre précédent (cf. Section 10.4.1). Ensuite, l'objet `nb` est créé. Enfin l'indice de Moran est calculé sur les deux variables mentionnées : la proportion de cadres et la proportion d'ouvriers dans les communes. La fonction `moran.test()` prend comme argument la variable à l'étude et la matrice de contiguïté ou de poids. Celle-ci est créée directement avec la fonction `nb2listw()`.

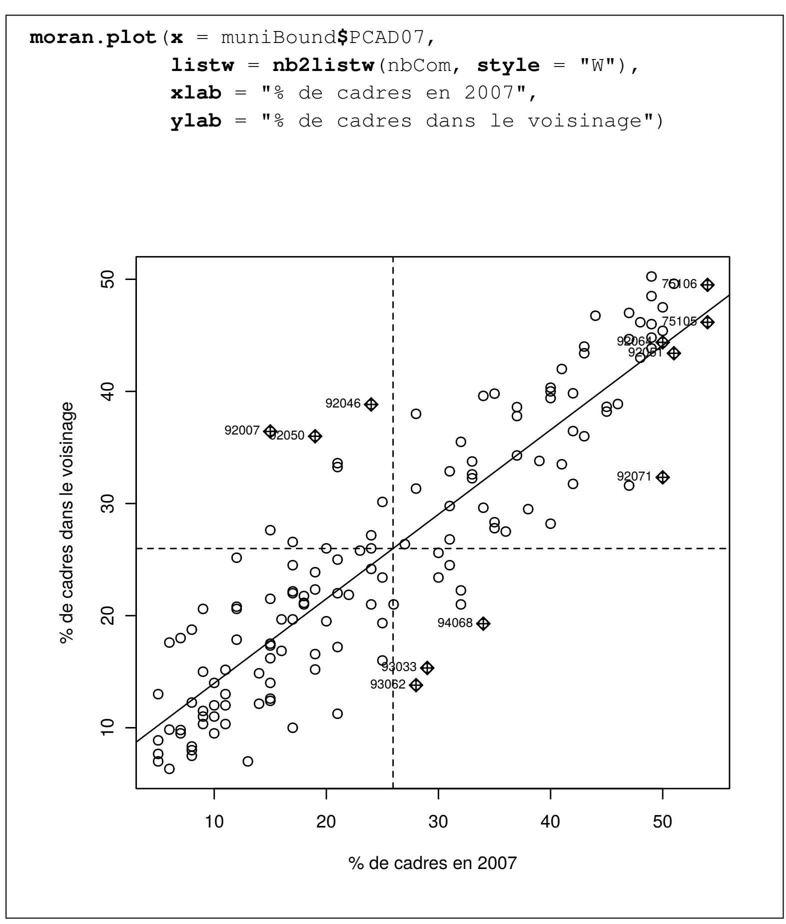
```
muniBound <- AttribJoin(df = socEco9907,
                        spdf = muniBound,
                        df.field = "CODGEO",
                        spdf.field = "INSEE_COM")

nbCom <- poly2nb(pl = muniBound,
                 row.names = muniBound$INSEE_COM,
                 snap = 50,
                 queen = TRUE)

moran.test(x=muniBound$PCAD07,
           listw = nb2listw(nbCom))

moran.test(x=muniBound$POUV07,
           listw = nb2listw(nbCom))
```

La fonction renvoie la valeur de l'autocorrélation spatiale (0,75 pour les cadres et 0,69 pour les ouvriers), la valeur de l'écart standardisé (14,59 et 13,43) ainsi que la *p-value*. La proportion de cadres et la proportion d'ouvriers présentent une autocorrélation positive, synonyme dans ce cas de ségrégation spatiale.

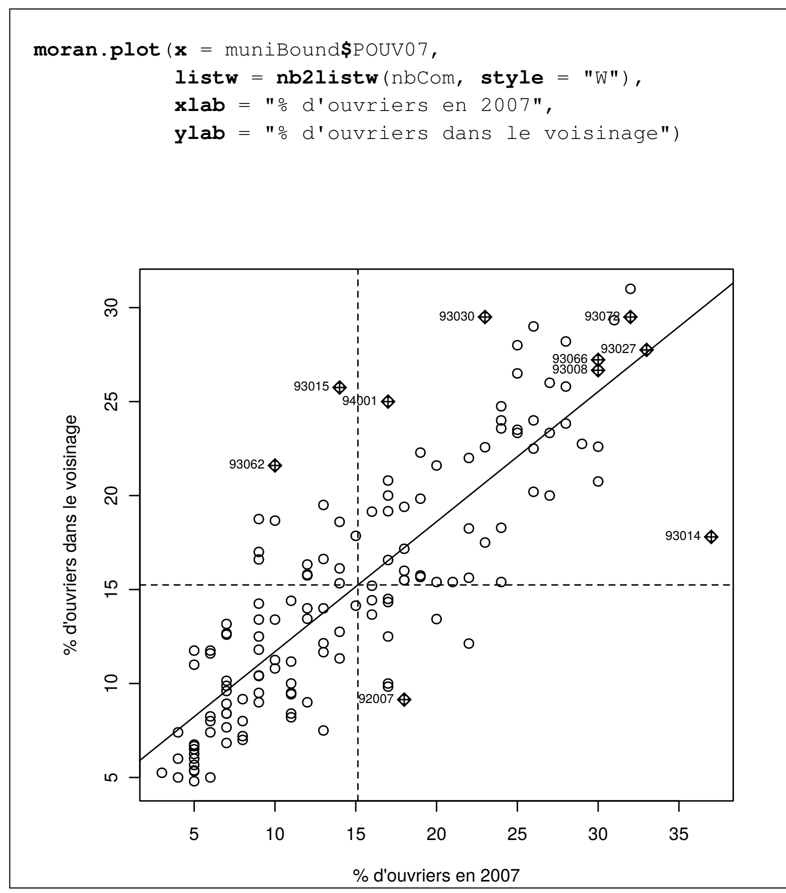
Les valeurs de l'indice de Moran peuvent être représentées graphiquement et cartographiquement. Ici la relation est représentée par un nuage de points (*Moran scatterplot*) croisant les valeurs en i avec la moyenne des valeurs du voisinage de i.

La fonction `moran.plot()` demande, comme la fonction `moran.test()`, une matrice de poids. Celle-ci est produite, comme précédemment, par la fonction `nb2listw()`. L'argument `style` est précisé même s'il s'agit de sa valeur par défaut : il indique que les poids doivent être standardisés par lignes. C'est cet argument qui permet de lire les valeurs sur l'axe y du graphique comme une moyenne pondérée par la distance de la variable à l'étude.

```
moran.plot(x = muniBound$PCAD07,
           listw = nb2listw(nbCom, style = "W"),
           xlab = "% de cadres en 2007",
           ylab = "% de cadres dans le voisinage")
```



```
moran.plot(x = muniBound$POUV07,
           listw = nb2listw(nbCom, style = "W"),
           xlab = "% d'ouvriers en 2007",
           ylab = "% d'ouvriers dans le voisinage")
```



Ce type de graphique est très riche, il permet de visualiser l'autocorrélation, de classer les unités spatiales en termes de profils et de détecter les cas exceptionnels. Les droites horizontale et verticale sont centrées sur les moyennes de chacune des deux variables : ces droites dessinent des cadrants. Là encore, il sera intéressant de cartographier ces profils [1].

Pour la proportion de cadres par exemple, ces cadrants guideront la lecture en distinguant les communes comportant peu de cadres et peu de

1. Voir par exemple la communication « R et espace » (Commenges, Cura, Mathian) présentée au *Semin-R* du Museum National d'Histoire Naturelle, disponible à l'adresse suivante : `http://rug.mnhn.fr/semin-r`.

cadres dans le voisinage, les communes comportant beaucoup de cadres et beaucoup de cadres dans le voisinage. Les cadrants opposés sont particulièrement intéressants puisqu'ils montrent les discontinuités : les communes comportant beaucoup de cadres situées dans un voisinage qui en comportent peu (Saint-Maur-des-Fossés par exemple, code 94068) et celles qui comportent très peu de cadres dans un voisinage qui en comporte beaucoup (Bagneux par exemple, code 92007).

Ce chapitre a présenté des techniques simples d'analyse spatiale, techniques permettant de caractériser la répartition spatiale d'objets et de phénomènes. Les mesures de centralité, point moyen et point médian, peuvent servir à résumer la distribution d'un semis de points dans une approche descriptive, elles peuvent être utilisées pour saisir des évolutions du semis de point, elles peuvent enfin être utilisées dans un cadre opérationnel (localisation optimale pour un hôpital). Les fonctions du *package* `spdep` sont très intéressantes pour analyser plus finement les phénomènes spatiaux, par exemple la répartition des semis de points ou l'homogénéité d'un phénomène observé dans des unités spatiales.

Annexes

ANNEXE A

Bibliographie thématique

Pour approfondir les méthodes abordées dans le manuel, voici une bibliographie indicative classée par thèmes. Cette liste reste volontairement très brève au regard de la très vaste littérature existante sur l'analyse de données et le logiciel R.

Programmation

BAATH R. (2012) "The state of naming conventions in R", *The R Journal*, Vol. 4/2, pp.74-75.

CHAMBERS J.M. (2008) *Software for data analysis. Programming with R*, Springer.

GENOLINI CH. (nd) *Petit manuel de S4. Programmation orientée objet sous R*, `http://christophe.genolini.free.fr/webTutorial/index.php`.

MUENCHEN R. (2008) *R for SAS and SPSS users*, Berlin, Springer.

MUENCHEN R., HILBE J.M. (2012) *R for Stata users*, Berlin, Springer.

WICKHAM H. (2014), *Advanced R*, http://adv-r.had.co.nz.

WICKHAM H. (2011), "The split-apply-combine strategy for data analysis", *Journal of Statistical Software*, vol.40, num.1, pp. 1-29.

Analyse statistique

CORNILLON P.-A., MATZNER-LØBER E. (2011) *Régression avec R*, Paris, Springer.

DUMOLARD P., DUBUS N., CHARLEUX L. (2003) *Les statistiques en géographie*, Presses Universitaires de Rennes.

HUSSON F., LÊ S., PAGÈS J. (2009) *Analyse de données avec R*, Presses Universitaires de Rennes.

LEBART L., MORINEAU A., PIRON M. (1995) *Statistique exploratoire multidimensionnelle*, Dunod.

SANDERS L. (1989) *L'analyse statistique des données en géographie*, Montpellier, Alidade-RECLUS.

Analyse de graphes

BARTHELEMY M. (2011) "Spatial Networks", *Physics Reports*, vol.499, pp.1-101, http://arxiv.org/abs/1010.0302.

GROUPE FMR (2010-2013) *Synthèses*, http://halshs.archives-ouvertes.fr/FMR.

HANNEMAN, R.A., RIDDLE M. (2005) *Introduction to social network methods*, Riverside, University of California, http://faculty.ucr.edu/~hanneman.

NEWMAN M. (2010) *Networks : an introduction*, Oxford, Oxford University Press.

WASSERMAN S., FAUST K. (1994), *Social network analysis. Methods and applications*, Cambridge, Cambridge University Press.

Visualisation graphique et cartographique

BÉGUIN M., PUMAIN D. (2000) *La représentation des données géographiques*, Paris, Armand Colin.

CHEN C., HÄRDLE W., UNWIN A. (dir.) (2008) *Handbook of data visualization*, Berlin, Springer.

TUFTE E. R. (1983) *The visual display of quantitative information*, Cheshire, Graphics Press.

WICKHAM H. (2011), "Graphical criticism : some historical notes", *Journal of Computational and Graphical Statistics*, vol.22, num.1, pp. 38-44, http://vita.had.co.nz/articles.html.

ZANIN CH., TRÉMÉLO M.-L. (2003) *Savoir faire une carte. Aide à la conception et à la réalisation d'une carte thématique univariée*, Paris, Belin.

Analyse spatiale

ANSELIN L. (1995) "Local indicators of spatial association', *Geographical analysis*, 27/2, pp.93-115.

BIVAND R.S., PEBESMA E.J., Gómez-Rubio V. (2008) *Applied spatial data analysis with R*, Springer.

PUMAIN D., SAINT-JULIEN TH. (2010) *L'analyse spatiale. Les localisation*, tome 1, 2e éd., Paris, Armand Colin.

PUMAIN D., SAINT-JULIEN TH. (2010) *L'analyse spatiale. Les interactions*, tome 2, 2e éd., Paris, Armand Colin.

ZANINETTI J.-M. (2005) *Statistique spatiale : méthodes et applications géomatiques*, Paris, Hermès Science Publications.

ANNEXE B

Liste des *packages* utilisés

Un certain nombre *packages* sont installés et chargés tout au long de la progression du manuel. En voici la liste exhaustive accompagnée d'une brève description. Lorsqu'un *package* est utilisé dans le manuel pour une utilisation unique et très spécifique, seule cette information est donnée, c'est le cas par exemple du package `ICSNP` :

- `ade4` : analyse géométrique des données multivariées
- `classInt` : discrétisation des variables continues
- `cluster` : méthodes de classification
- `dplyr` : manipulation des données
- `FactoClass` : analyse géométrique des données multivariées
- `fields` : outils pour le traitement de données spatiales
- `ggplot2` : représentations graphiques et cartographiques
- `HistData` : jeux de données historiques
- `Hmisc` : mélanges de fonctions courantes
- `ICSNP` : calcul du point médian
- `igraph` : analyse de graphes

- mapproj : gestion des systèmes de projection des données spatiales
- maptools : outils de cartographie
- plyr : manipulation des données
- raster : traitement et affichage de données *raster*
- rCarto : outils de cartographie
- RColorBrewer : création de palettes de couleur
- reshape2 : transposition individus-variables
- rgdal : manipulation des objets spatiaux
- rgeos : manipulation des objets spatiaux
- scales : format des axes des graphiques
- sp : manipulation des objets spatiaux
- spdep : statistiques spatiales
- sqldf : exécution de requêtes SQL

Table des matières

Introduction : R pour . . . iii

1 Prise de contact 1

1.1 R dans une coquille de noix 1
1.2 Installation 2
1.3 Utilisation de RStudio 3
1.4 Conventions d'écriture 6
1.5 Versions et mises à jour 7
1.6 L'exemple et les données 7
1.6.1 Description du fichier SocEco9907.csv 8
1.6.2 Description du fichier PopCom3608.csv 9
1.6.3 Description du fichier MobResid08.txt 9
1.6.4 Description des données cartographiques 10

2 Prise en main et manipulation des données 11

2.1 Description et manipulation des objets 12
2.1.1 Les principaux types d'objets 12
2.1.2 Le vecteur et le facteur 14
2.1.3 Déclarer des objets avec l'opérateur d'assignation 16
2.1.4 Modifier le type d'un objet 19
2.1.5 Désigner des lignes, des colonnes ou des valeurs 20
2.1.6 Traitement des valeurs manquantes 22
2.1.7 Se renseigner sur les objets et leur contenu 23
2.2 Importation et exportation 24

2.2.1 Le répertoire de travail 25
2.2.2 Les formats de données 25
2.2.3 Codage des caractères et des séparateurs 25
2.2.4 Importer des données en format texte 26
2.2.5 Exporter des données en format texte 26
2.2.6 Importation et exportation depuis/vers d'autres formats 27
2.3 Recoder et trier 27
2.3.1 Sélectionner et recoder 27
2.3.2 Trier 30
2.4 Manipulation avancée 32
2.4.1 Superposition, concaténation, jointure 32
2.4.2 Agrégations et traitements par blocs 34
2.4.3 Transposition variables-observations 37
2.5 *Packages* polyvalents 41

3 Introduction à la programmation **45**
3.1 Structures itératives 46
3.1.1 Exemple 1 : simple boucle `for` 47
3.1.2 Exemple 2 : double boucle `for` 48
3.2 La structure conditionnelle `if...else` 51
3.3 Les fonctions 52
3.3.1 Premier aperçu sur les fonctions 52
3.3.2 Discrétisation automatique 55
3.3.3 Calcul de l'équilibre de Wardrop 57
3.3.4 Calcul des vols de Syracuse 61
3.4 L'application de fonctions sur des ensembles 65

4 Analyse univariée **71**
4.1 Calculs simples et recodages 72
4.2 Résumés statistiques 74
4.3 Représentation graphique des distributions statistiques 76

5 Analyse bivariée **83**
5.1 Préambule 84
5.1.1 Types de variables 84
5.2 Relation entre deux variables quantitatives 84
5.2.1 Représentation graphique de la relation 85
5.2.2 Calcul et significativité des corrélations 88
5.2.3 Régression linéaire simple 90
5.3 Relation entre une variable quantitative et une variable qualitative 94
5.3.1 Représentation graphique de la relation 94
5.3.2 Analyse de la variance à un facteur 96
5.4 Relation entre deux variables qualitatives 99

5.4.1 Représentation graphique de la relation 99
5.4.2 Analyse de la répartition observée 101
5.4.3 Analyse des écarts à l'indépendance 103

6 Analyses factorielles 105
6.1 Analyse en composantes principales 106
6.1.1 Réaliser l'analyse 106
6.1.2 Examiner et interpréter les résultats 107
6.1.3 Analyse des contributions 111
6.1.4 Analyse des qualités de représentation 113
6.2 Analyse factorielle des correspondances 114
6.2.1 Réaliser l'analyse 115
6.2.2 Interpréter les résultats 116

7 Méthodes de classification 121
7.1 Rappels théoriques 122
7.2 Classification ascendante hiérarchique 126
7.2.1 Coupure de l'arbre et description des classes 129
7.3 Couplage analyse factorielle - classification 134
7.3.1 Classification sur une métrique du χ^2 134
7.3.2 Description des classes 135

8 Analyse de graphes 141
8.1 Aperçu et préparation des données 142
8.2 Création et exploration du graphe 145
8.2.1 Importation et exportation de graphes 145
8.2.2 Création d'un graphe avec R 147
8.2.3 Calcul de mesures globales 147
8.3 Mesures locales et manipulation du graphe 148
8.4 Cliques et communautés 151
8.5 Visualisations 155

9 Focus sur la visualisation graphique 159
9.1 Exportation des tableaux et des graphiques 161
9.2 Gestion des couleurs 163
9.3 Introduction à `ggplot2` 165
9.3.1 Intérêt vis-à-vis des fonctions graphiques classiques 165
9.3.2 Aperçu de la syntaxe 166
9.3.3 Variables visuelles 168
9.3.4 Construction d'une planche 171
9.3.5 Variables de regroupement 172

10 Introduction aux objets spatiaux et à la cartographie **175**
10.1 R pour la cartographie 178
10.2 Prise en main des données spatiales 179
10.2.1 Construction d'un objet spatial 179
10.2.2 Importation et prise en main des objets spatiaux 183
10.2.3 Le système de projection 186
10.2.4 Exportations 188
10.3 Cartographie des objets ponctuels 188
10.4 Cartographie des objets zonaux 193
10.4.1 Cartographie avec la fonction `plot()` 194
10.4.2 Cartographie avec la fonction `ggplot()` 198
10.5 Cartes *take away* 202
10.6 Cartographie à la chaîne 203

11 Initiation aux statistiques spatiales **209**
11.1 Point moyen et point médian 210
11.1.1 Centralité dans un espace à une dimension 210
11.1.2 Point médian dans un espace à deux dimensions 213
11.1.3 Point moyen dans un espace à deux dimensions 217
11.2 Mesurer les espacements dans un semis de points 220
11.2.1 Notion de plus proche voisin et mesures associées 220
11.2.2 Calcul d'une matrice de distance 221
11.2.3 Fonctions spécifiques du *package* `spdep` 222
11.2.4 Visualisations des voisinages 225
11.3 Autocorrélation spatiale et ressemblances locales 226
11.3.1 La notion d'autocorrélation spatiale 226
11.3.2 Répartition des cadres et des ouvriers 227

Annexes 235

A Bibliographie thématique **235**

B Liste des *packages* utilisés **239**