R pour les géographes

Analyse de données, analyse spatiale
et cartographie avec 

Groupe ElementR
Mars 2013
Version 0.1

Groupe ElementR (2013) ***R pour les géographes : analyse de données, analyse spatiale et cartographie avec R*****, Paris, UMR Géographie-cités.**

Le présent document est réalisé par le groupe ElementR, groupe de huit auteur-e-s coordonné par Hadrien Commenges. Il est le fruit de sessions de formation au logiciel R organisées à l'UMR Géographie-cités durant l'année 2012.

Le résultat de ce travail collectif est mis à disposition sous licence Creative Commons, dans le respect de l'esprit de partage des connaissances qui est le fondement de R et la raison de son succès.

Ce manuel a pour ambition d'offrir un support technique à des étudiants, enseignants, chercheurs, ingénieurs, etc. qui partagent notre intérêt pour l'utilisation des méthodes d'analyse des données en géographie et souhaiteraient pouvoir développer leurs travaux avec R. Ce manuel est largement inspiré d'autres manuels R spécifiques ainsi que de nos différents cours pour les aspects méthodologiques. Il est accompagné de jeux de données et peut être utilisé comme support pédagogique.

Ce document est une première version où nous avons tenté d'harmoniser les différents chapitres du point de vue de leur forme et des jeux de données mobilisés. Nous avons cependant bien conscience qu'il est perfectible et nous invitons les utilisateurs de ce manuel à nous faire part de tout commentaire, critique ou suggestion, dans la perspective d'une version complétée et améliorée, à l'adresse suivante : **`< elementr@parisgeo.cnrs.fr >`**

Laurent BEAUGUITTE
UMR IDEES

Florent LE NÉCHET
UMR LVMT

Elodie BUARD
IGN - COGIT
UMR Géographie-cités

Marion LE TEXIER
UMR Géographie-cités
Université du Luxembourg

Hadrien COMMENGES
UMR Géographie-cités

Hélène MATHIAN
UMR Géographie-cités

Robin CURA
UMR Géographie-cités

Sébastien REY
UMR Géographie-cités

Table des matières

Chapitre 1

Introduction

1.1 R pour ...

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 commerciaux : SAS, SPSS et Stata. La validité des critères sur lesquels l'auteur appuie son analyse peut être discutée, mais le constat quant à lui est indiscutable : R est un langage qui gagne en importance depuis le début des années 2000, et cette croissance ne semble pas devoir s'arrêter dans les années qui viennent. 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 (voir la liste et les liens sur `http://www.r-project.org/conferences.html`) 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 grands 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 [Bivand *at al.* 2008], mais ce dernier est en anglais, il est difficile à aborder et son approche est très statisticienne. Il nous paraissait important d'adopter 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 en 2011/2012 pour un public d'étudiants 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 réalisant des études quantitatives. Mais surtout parce que la prise en compte de l'espace et la cartographie sont de plus en plus présentes dans d'autres disciplines, la sociologie, l'histoire, les sciences politiques, etc. 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 de l'ouverture des données (*open data*), sont de plus en plus utilisées pour produire des cartes de thèmes d'intérêt (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 réaliser la majeure partie de ces opérations dans un même flux de travail (*workflow*, c'est-à-dire la chaîne des traitements réalisés). 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 : il y avait au début des années 2000 quelques 30 *packages* (bibliothèques de fonctions) assez généralistes, en 2012 il y en avait plus de 3000.

1.2 Plan et fil pédagogique

L'intérêt du manuel est également de proposer un ensemble comprenant les explications, les programmes et les données. Toutes les applications sont réalisées sur le même jeu de données caractérisant le même espace d'étude : Paris et la petite couronne (départements 75, 92, 93, 94).

Il est divisé en trois sections indépendantes, contenant chacune plusieurs chapitres autonomes. Chaque chapitre est indépendant dans le sens ou l'on reprend précisément en début de chapitre les noms des fichiers de données nécessaires ainsi que les *packages* de R nécessaires au déroulement du programme. Chaque chapitre est complété par des références spécifiques permettant d'approfondir soit le travail dans un environnement R, soit les méthodes d'analyse spatiale.

La première section « Manipulation des données et programmation » comporte des éléments de langage nécessaires pour débuter avec R et manipuler les données : le chapitre 1 présente des éléments de prise en main et illustre différentes méthodes pour transformer les données. Un second chapitre présente des éléments plus avancés de programmation avec la mise en œuvre de boucles et de fonctions.

La deuxième section présente des méthodes statistique d'« Exploration des données géographiques ». Ainsi les chapitre 3 et 4 abordent les méthodes de traitements statistiques univariés et bivariés classiquement utilisés en analyse spatiale. Le chapitre 5 décline des questions nécessitant l'utilisation de méthodes factorielles multivariées. Enfin le chapitre 6 présente les méthodes de classification.

La dernière section, « Éléments spécifiques de traitement de l'espace », regroupe trois chapitres illustrant des aspects plus spécialisés en géographie. Le chapitre 7 revient sur des fondamentaux de l'analyse de réseaux. Le chapitre 8 introduit à la cartographie avec R. Enfin le chapitre 9 présente des éléments de statistiques spatiales et, en particulier, une initiation à l'autocorrélation spatiale.

1.3 Introduction à R

Généralités

R est un logiciel créé en 1993 par Robert Gentleman et Ross Ihaka, de l'Université d'Aukland. C'est un logiciel libre, gratuit et multiplateforme (Linux, Windows, MacOS). En pleine expansion, il concurrence avec succès les logiciels commerciaux qui détenaient cette part de marché : SAS, SPSS et Stata. Robert A. Muenchen propose sur son site Internet (`http://r4stats.com/`) des analyses quantifiées de cette mise en concurrence. Il est aussi l'auteur de manuels facilitant la migration des utilisateurs des grands logiciels commerciaux vers le logiciel R [Muenchen 2008, Muenchen 2012].

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 Network (CRAN - `http://cran.r-project.org`). Cette structure modulaire, commune à de nombreux logiciels libres, fait que l'étendue des applications possibles 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.

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 statistiqes, spatiales et temporelles, de produire des tableaux, des

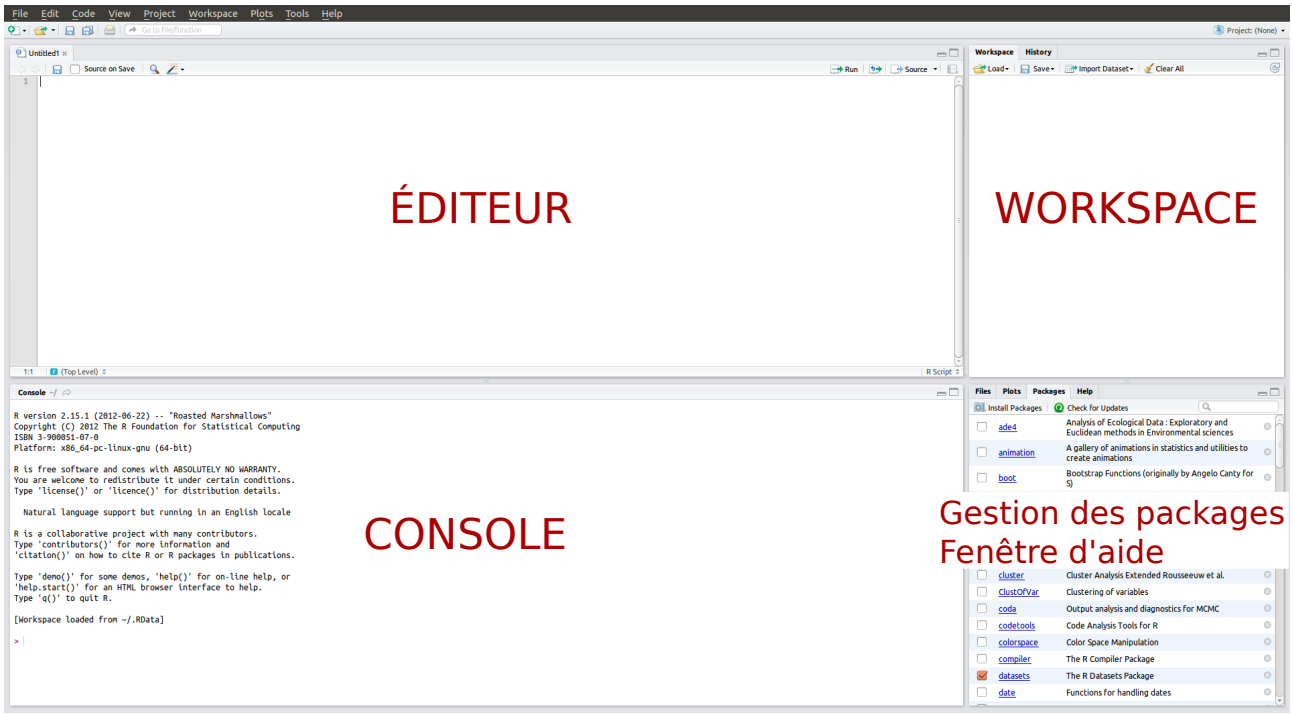


FIGURE 1.1 – Structure de l'interface graphique de RStudio

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 SIG à un logiciel de statistiques par exemple).

Interface et 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 : Résultats, Éditeur, et Journal. 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, l'accès aux *packages*.

Pour utiliser R et commencer à travailler sur ce manuel, il faut installer R (`http://cran.r-project.org`) puis RStudio (`http://rstudio.org`). L'interface graphique de RStudio se compose de quatre fenêtres (cf. Figure 1.3) : 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, actualisation, chargement) et d'accéder à l'aide. Le moteur de recherche de l'onglet « Aide » peut trouver la documentation de toutes les fonctions installées et chargées.

En ce qui concerne l'utilisation des *packages*, il faut bien distinguer deux choses : installer un *package* et le charger. L'installation se fait par la commande `install.packages()` ou bien par l'onglet intitulé « Packages » de l'interface RStudio. Dans cet onglet apparaissent tous les *packages* installés sur la machine. 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()`.

Versions et actualisations

R est un logiciel à structure modulaire dont la base (`r-base`) et les différentes *packages* sont actualisés 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 d'actualisation. 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, vue la dynamique qui porte ce logiciel, nous conseillons de rester le plus à la page possible et d'actualiser 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 actualisée en mars 2013 qui comprend : la version 2.15.3 de R, la version 0.97.318 de RStudio et les dernières actualisations de tous les *packages* utilisés.

1.4 Utilisation de l'aide et des raccourcis

RStudio présente deux fonctionnalités très intéressantes qui facilitent le travail : l'accès intégré à l'aide et la capacité de compléter automatiquement les termes en cours d'écriture (capacité qualifiée par la suite d'autocomplétion). Dans la fenêtre en bas à droite de l'interface, l'onglet d'aide permet d'accéder à la documentation de toutes les fonctions des *packages* installés et chargés. Pour accéder à l'aide d'une fonction, on peut également utiliser la syntaxe `?MaFonction` ou bien `help(MaFonction)`.

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 :

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

Parmi les raccourcis clavier intéressant, il faut mentionner le raccourci « `Alt + -` » qui 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*.

1.5 Conventions d'écriture

Un bon programme doit réunir au moins deux qualités : l'efficience (atteindre l'objectif avec une parcimonie de moyens) et la lisibilité. Indépendamment de la qualité et de la complexité du programme, sa lisibilité reste un impératif 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 symbole `#` : 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 [Baath 2012]. 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 », « mon.objet », « 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.

- Parenthèses et crochets : l'écriture de commandes longues, souvent des fonctions ou des ensemble de conditions (if) devraient aussi respecter certaines conventions pour faciliter leur lecture (cf. guides de style précités).

Dans ce manuel, nous avons nommé les objets en utilisant des majuscules (MonObjet) et nous avons adopté une convention d'écriture supplémentaire qui consiste à indiquer la classe de l'objet créé par une minuscule avant le nom de l'objet. Par exemple si mon objet est :

- un vecteur (v), on le nomme « `vMonObjet` » ;
- un *data.frame* (d), on le nomme « `dMonObjet` » ;
- une matrice (m), on la nomme « `mMonObjet` » ;
- une liste (l), on la nomme « `lMonObjet` ».

Cette contrainte est strictement respectée dans le premier chapitre, puis relâchée par la suite au vu de la grande diversité d'objets manipulés.

1.6 L'exemple et les données

Toutes les applications sont réalisées 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 3 jeux de données statistiques en libre accès sur le site de 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.

Nous travaillerons également sur des données cartographiques en accès libre sur le site de l'IGN, vectorielles et raster : 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.

Pour les données socioéconomiques, nous avons regroupé un ensemble de variables 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.

Description du fichier data99_07.csv :

CODGEO	Code communal
LIBGEO	Nom de la commune (arrondissement pour Paris)
X, Y	Coordonnées géographiques en Lambert 2 étendu (hectomètres)
SURF	Superficie en hectares
EMPLOI99, EMPLOI06	Nombre d'emplois
ACTOCC99, ACTOCC06	Nombre d'actifs occupés
P20ANS99, P20ANS07	Proportion de moins de 20 ans (population totale)
PNDIP99, PNDIP07	Proportion de non diplômés (population > 15 ans)
TXCHOM99, TXCHOM07	Proportion de chômeurs (population active)
INTAO99, INTAO07	Proportion d'intérimaires (population active occupée)
PART99, PART07	Proportion d'artisans (population active occupée)
PCAD99, PCAD07	Proportion de cadres (population active occupée)
PINT99, PINT07	Proportion de professions intermédiaires (population active occupée)
PEMP99, PEMP07	Proportion d'employés (population active occupée)
POUV99, POUV07	Proportion d'ouvriers (population active occupée)
PRET99, PRET07	Proportion de retraités (population active occupée)
PMONO99, PMONO07	Proportion des familles monoparentales (familles)
PREFETR99, PREFETR07	Proportion des ménages dont la personne de référence est étrangère (ménages)
RFUCQ201, RFUCQ207	Revenu médian (euro courant)

Description du fichier pop36_08.csv (le sigle RP signifie Recencement de la Population) :

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

Description du fichier dMobResid2008.txt :

CODGEO	Code INSEE de la commune d'origine
LIBGEO	Nom de la commune d'origine
DCRAN	Code INSEE de la commune de destination
L_DCRAN	Nom de la commune de destination
NBFLUX	Nombre de personnes ayant déménagé de la commune d'origine vers la commune de destination

Concernant les données cartographiques, le fichier des limites communales ne contient que les codes permettant de recouper les données que nous venons de présenter. 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 vise en particuler à se familiariser avec les principaux objets sous R (vecteurs, matrices, tables) et les opérations les plus courantes (calculs mathématiques, assignations, tests conditionnels).

Prérequis

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

Packages nécessaires

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

- *Package* `sqldf` : cette extension (`http://code.google.com/p/sqldf`) permet d'executer des requêtes SQL (« Structured Query Language ») sur les objets *data.frame* de R. SQL est un langage normalisé permettant la gestion de bases de données relationnelles (`http://en.wikipedia.org/wiki/SQL`). Ce *package* est constitué d'une unique fonction éponyme, qui prend pour argument la requête SQL souhaitée, entre guillemets : sqldf(« instruction sql »).
- *Package* `reshape2` : cette extension est développée par Hadley Wickham (`http://had.co.nz/reshape`). Elle permet de passer facilement d'un format de données « long » (une ligne par individu et par variable) à un format « large » (une seule ligne par individu).

Habituellement nous avons les observations en ligne et les variables en colonne. `Reshape` part du principe que ces variables peuvent être divisées en deux groupes : les identifiants et les mesures. A partir de cette définition, ce *package* met à disposition des utilisateurs un certain nombre de commandes pour la transformation des structures de données.

Données nécessaires

Dans ce chapitre trois tableaux de données produits par l'INSEE sont utilisés :

- `data99_07` : données de référence pour 1999 et 2007 avec différentes variables sur la composition socio-professionnelle des communes de Paris et la petite couronne.
- `pop36_08` : données de population sans double compte des recensements de la population de 1936 à 2008 pour Paris et la petite couronne.
- `dMobResid2008` : données de mobilités résidentielles entre communes de la petite couronne en 2008.

Pour une description plus précise du contenu des fichiers, voir le Chapitre 1.

2.1 Description et manipulation des objets proposé par R

R est un langage orienté objet : tout ce qui est créé et manipulé sous R est un objet. R met à notre disposition des objets permettant de stocker/structurer des données.

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. Le vecteur peut être accompagné d'une déclaration de niveaux (`levels=`) et/ou d'étiquettes (`labels=`). Plusieurs fonctions de création de tableaux ou d'importation de données externes 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 une sous-classe 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, texte, etc. Avec RStudio, dans la fenêtre « Espace de travail » on peut cliquer sur les objets de type matrice et les afficher.
- Liste (*list*) : liste ordonnée d'objets. Une liste peut contenir toutes les types d'objets existants dans R (autrement dit les listes permettent de regrouper un ensemble d'entités hétérogène). 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*. L'objet *data.frame* combine des caractéristiques de *list* et de *matrix*. C'est une matrice bi-dimensionnelle dont les champs peuvent contenir différents types de données : valeurs numériques, alphanumériques, booléennes. Avec RStudio, dans la fenêtre « Espace de travail » on peut cliquer sur les objets *data.frame* et les afficher.
- Autre objets : les données spatiales sont stockées dans des objets particuliers qui sont détaillés au Chapitre 9. En dehors de ces objets particuliers, certaines fonctions créent des objets sur mesure : par exemple la fonction `lm()` (*linear model*) créé un objet de type *lm* qui est une liste d'objets (*fitted.values*, *p.value*, etc.).
- Fonction : il faut insister sur deux caractéristiques importantes des fonctions dans R, 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.

Déclarer des objets avec l'opérateur d'assignation

Pour créer et initialiser un objet, on utilise l'opérateur d'assignation `->` ou `<-` . 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`". Le signe `=` fonctionne aussi comme opérateur d'assignation mais nous recommandons d'utiliser `<-` pour éviter les confusions avec les tests d'égalité (`==`).

```
1 + 1  # exécute l'opération et renvoie le résultat dans la console

## [1] 2

MonObjet <- 1 + 1
MonObjet

## [1] 2
```

Dans RStudio, chaque objet créé dans est affiché dans la fenêtre « Espace de travail » 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 une certaine classe d'objet ;
- de façon explicite, c'est-à-dire en déclarant la classe de l'objet au moment de sa création.

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

## [1] "numeric"


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

## [1] "factor"
```

On peut créer les différents types d'objets citées précédemment avec des fonctions éponymes : `vector()`, `matrix()`, `list()`, `data.frame()`, etc. La fonction `c()` combine un ensemble de valeurs dans un vecteur. On l'utilise pour créer un vecteur simple (Cas 1 : Créer un vecteur), pour créer un vecteur et le mettre en forme dans une matrice à 3 lignes et 4 colonnes (Cas 2 : Créer une matrice), pour créer un vecteur de deux noms (ID et CLASSE) qu'on assigne comme noms de colonnes à un *data.frame* (Cas 3 : Créer un *data.frame*) :

Créer un vecteur :

```
vExemple1 <- c(1, 2, 3, 4, 5)
vExemple1

## [1] 1 2 3 4 5

vExemple2 <- c("CP", "CE1", "CE2", "CM1", "CM2")
vExemple2

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

Créer une matrice :

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

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

Créer un *data.frame* :

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, on peut utiliser la fonction `names()` ou `colnames()`.

```r
dExemple <- data.frame(vExemple1, vExemple2, stringsAsFactors = FALSE)

colnames(dExemple)

## [1] "vExemple1" "vExemple2"

colnames(dExemple) <- c("ID", "CLASSE")
dExemple <- data.frame(ID = vExemple1, CLASSE = vExemple2, stringsAsFactors = FALSE)
dExemple

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

Un *data.frame* est une liste de vectors, qui sont de même taille, mais pas forcément de même nature. Comme toutes les listes, les composants de celles-çi peuvent être nommés, c'est ainsi que sont stockés les noms des colonnes de notre *data.frame* (voir la fonction `names()`)

Convertir les objets d'un type à un autre

Il est également possible de transformer un objet d'une certaine classe en une autre classe, avec des fonctions suivant la syntaxe suivante `as.` suivi du type choisi `as.matrix`, `as.numeric`, etc. Ces transformations sont utiles mais posent certains problèmes qui 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.

```r
dExemple2 <- data.frame(col1 = vExemple1, col2 = c(2, 5, 8, 7, 8), stringsAsFactors = FALSE)

mExemple2 <- as.matrix(dExemple2)  # transformation sans perte
mExemple2

##      col1 col2
## [1,]    1    2
## [2,]    2    5
## [3,]    3    8
## [4,]    4    7
## [5,]    5    8

MonVecteur <- as.numeric(MonFacteur)  # transformation avec perte des étiquettes
MonVecteur

## [1] 1 1 2 1 2 2

vExemple2 <- as.vector(mExemple2)  # perte d'une dimension, perte des noms des champs
```

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. En règle générale on désigne la dimension (1 dimension pour un vecteur, 2 pour une matrice, 3 pour un cube, etc.) par son numéro entre crochets ou par son nom précédé d'un dollar :

```
# Sélection
vExemple1[2]  # renvoie la deuxième valeur du vecteur

## [1] 2

mExemple[2, 3]

## [1] 8

# renvoie la valeur située à la deuxième ligne, troisième colonne de la
# matrice
mExemple[, 3]  # renvoie l'ensemble de la troisième colonne de la matrice

## [1] 7 8 9

mExemple[2, ]  # renvoie l'ensemble de la deuxième ligne de la matrice

## [1]  2  5  8 11

dExemple$CLASSE

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

dExemple[, 2]  # idem que précédemment

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

dExemple[3, 2]

## [1] "CE2"

# renvoie la valeur située à la troisième ligne, deuxième colonne du
# data.frame

# Sélection et assignation
vExemple2[1] <- "Cours primaire"
vExemple2[2:3] <- "Cours élémentaire"
vExemple2[4:5] <- "Cours moyen"
vExemple2

##  [1] "Cours primaire"    "Cours élémentaire" "Cours élémentaire"
##  [4] "Cours moyen"       "Cours moyen"       "2"
##  [7] "5"                 "8"                 "7"
## [10] "8"


mExemple[2, 3] <- 99
mExemple

##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5   99   11
## [3,]    3    6    9   12
```

Il est important d'examiner la structuration d'un tableau : il faut prendre garde lors de la manipulation des tableaux avec R du type d'objet concerné. Toutefois, à l'usage, certains utilisateurs trouvent pesant de sans cesse devoir faire référence au *data.frame* accompagné du $ avant de désigner la variable (`dMonTableau$VARIABLE`). Il est possible d'attacher le *data.frame* 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 déconseillé.

Attention dans le cas de vecteur dans un *data.frame*, il n'est pas possible d'utiliser la notation $ pour accéder aux données, seule fonctionne la notation [ ] et [[ ]].

Se renseigner sur les objets et leur contenu

Des commandes permettent de lister et d'effacer les objets déclarés dans R

```
MonObjet <- 1 + 1

# lister les objets
ls()

## [1] "MonObjet"



# effacer un seul objet
rm(MonObjet)

# effacer tous les objets
rm(list = ls())
```

Plusieurs fonctions permettent de se renseigner sur la nature et le contenu des objets, principalement la fonction `str()` qui renvoie des informations sur la structure de l'objet, la fonction `class()` qui renvoie sa classe et les fonctions de type `is.something()` qui renvoient un booléen :

```
data(cars)  # jeu de données servant d'exemple contenu dans la base R
str(cars)

## '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 ...

MonObjet <- 1 + 1
class(MonObjet)

## [1] "numeric"

is.matrix(MonObjet)

## [1] FALSE

is.vector(MonObjet)

## [1] TRUE
```

2.2 Importation et exportation

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

Le répertoire de travail

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

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

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.) ou formats spécifiques de bases de données (à voir au cas par cas).

Codage des caractères et des séparateurs

Il faut bien distinguer le format d'un fichier, indiqué par son extension (.xls, .sav, etc.), du codage (*encoding*) de ses caractères. Il est toujours utile de savoir quel est le codage des documents qu'on utilise, en général UTF-8 ou Latin1. Quand on réutilise ou qu'on importe un fichier avec des champs de caractères en français, et que les accents et les apostrophes sont mal affichés, une corruption des données peut survenir. 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.

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 « Espace de travail » (« *Workspace* ») de la fenêtre placée en haut à droite de l'écran, on utilise le bouton « Importer les données > Fichier texte » (« *Import Dataset* > *From Text file* »). 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=`). On peut aussi préciser le codage du fichier d'origine : si on travaille en UTF-8 et que le fichier importé affiche mal les accents et les apostrophes (noms en français) on utilise l'option de codage (`encoding=`). Par exemple `read.table("chemin/fichier.csv", sep = ";", dec = ",", header = TRUE)`.

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. Les fonctions `read.table()` et `write.table()` peuvent être remplacées par des fonctions qui présentent certains réglages par défaut, en particulier des séparateurs.

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.

Accès à des bases de données externes

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.).

Sélectionner, recoder, classer

Avant de commencer, voici une liste des principaux opérateurs logiques utilisés dans les sélections :

- 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`
- $A \cap B$ (intersection de deux conditions) : `A & B`
- $A \cup B$ (union de deux conditions) : `A | B`

Le réflexe des utilisateurs de logiciels de statistiques classiques est de vouloir faire des sélections et des recodages avec des opérateurs conditionnels (if [...] then). La réalisation conditionnelle d'une instruction possède sous R la syntaxe suivante :

```
if (condition) {instruction si condition vraie}
else {instruction si condition fausse}
```

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 CLASSE = Mineur else CLASSE = Majeur}` .

Avec R cette syntaxe n'est pas très utilisée et nous lui préférons souvent une syntaxe simplifiée et optimisée pour ce type de traitement (sélection puis ré-assignation en bloc). Une formulation pseudo-syntaxique serait alors :

`parcours 1 : tant que (ligne individu != 0 et AGE < 18) {CLASSE = Mineur}`

`parcours 2 : tant que (ligne individu != 0 et AGE >= 18) {CLASSE = Majeur}`

On traite ci-dessous un exemple de ceci sous R :

```r
# Création de la table
dIndividus <- data.frame(c(1, 2, 3, 4, 5), c(12, 17, 24, 45, 8), c("H", "H",
    "F", "F", "F"))
colnames(dIndividus) <- c("ID", "AGE", "SEXE")
colnames(dIndividus)

## [1] "ID"   "AGE"  "SEXE"

# Assignation conditionnelle
dIndividus$CLASSE[dIndividus$AGE < 18] <- "Mineur"
dIndividus$CLASSE[dIndividus$AGE >= 18] <- "Majeur"
```

Ce type de syntaxe est utilisé de la même façon avec les matrices, les vecteurs et les autres types d'objets. A la différence des logiciels classiques, il est possible avec R de faire des sélections et des opérations mettant en jeu plusieurs objets distincts, par exemple :

```r
vAge <- as.vector(dIndividus$AGE)
vSexe <- as.vector(dIndividus$SEXE)
vCombinaison <- vector(mode = "character", length = 5)
vCombinaison[vAge < 18 & vSexe == "H"] <- "Homme mineur"
vCombinaison[vAge < 18 & vSexe == "F"] <- "Femme mineure"
vCombinaison[vAge >= 18 & vSexe == "F"] <- "Femme majeure"
vCombinaison

## [1] "Homme mineur"  "Homme mineur"  "Femme majeure" "Femme majeure"
## [5] "Femme mineure"
```

La fonction `subset()` est très utile pour sélectionner des parties d'une table en fonction de la valeur de tel ou tel paramètre, ou pour retenir certaines colonnes d'une table. À titre d'exemple, il peut parfois être utile de conserver les lignes pour plusieurs valeurs d'une même variable ; pour ce faire on peut utiliser l'instruction "ou" :

```r
subset(dIndividus, dIndividus$AGE == "17" | dIndividus$AGE == "24")

##   ID AGE SEXE CLASSE
## 2  2  17    H Mineur
## 3  3  24    F Majeur
```

Il convient de noter que la fonction `if()` est utilisée dans des boucles mais n'est pas utilisable dans des sélections classiques au sein de tableaux. La fonction `ifelse()` en revanche peut être utilisée dans ce contexte. Elle se compose d'un test (prédicat) qui renvoie un `TRUE`/`FALSE` et assigne une valeur en fonction de ce test. Par exemple, l'instruction ci-dessous permet de calculer autrement la *CLASSE* de l'exemple précédent (Mineur ou Majeur).

```r
dIndividus$MAJMIN <- ifelse(dIndividus$AGE >= 18, "Majeur", "Mineur")
```

Pour trier les valeurs d'un objet dans un certain ordre, croissant, décroissant ou alphabétique, nous utilisons les fonctions `sort()` et `order()`. La fonction `sort()` ne permet de trier les valeurs que d'une seule variable, alors que la fonction `order()` permet de trier un tableau de valeurs en fonction d'une ou plusieurs variables. `order()` sera donc la seule méthode possible pour trier des colonnes dans un *data.frame*, et n'aura que peu d'intérêt lorsqu'il s'agira de trier des données à une seule dimension. Dans ce cas `sort()` sera beaucoup plus simple et appropriée.

Autre différence notable entre les deux fonctions, `order()` renvoie le rang tenu par les valeurs, alors que `sort()` renvoie la liste de valeur ordonnées. Ainsi pour classer la table elle-même il faut utiliser la syntaxe suivante `data[order(data)]` où data représente la structure de données à une dimension à trier. 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 de la table et `data$var` le nom de la variable selon lequel on souhaite trier l'ensemble de la table, on utilise l'instruction `data[order(data$var),]`.

```
# Trier un vecteur
vAgeCroissant <- sort(vAge)
head(vAge)

## [1] 12 17 24 45  8

head(vAgeCroissant)

## [1]  8 12 17 24 45


# La syntaxe équivalente avec order, bien que peu utilisée
vAgeCroissant <- vAge[order(vAge)]

# Trier dans un autre ordre
vAgeDecroissant <- sort(vAge, decreasing = TRUE)
```

Pour trier un tableau, avec `order()`, on utilisera la syntaxe suivante :

```
# Trier un tableau
dIndividusClasses <- dIndividus[order(dIndividus$AGE), ]
dCars <- cars[order(cars$speed, cars$dist), ]
head(cars)

##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10

head(dCars)

##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

Si trier les valeurs d'un vecteur numérique ne pose pas de problème particulier, il est plus délicat de trier un facteur. En effet il faut faire attention au fait que 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.

```
# Trier un facteur
vSexeLevels <- sort(MonFacteur)
vSexeLabels <- sort(as.character(MonFacteur))
head(MonFacteur)

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

```r
head(vSexeLevels)

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

head(vSexeLabels)

## [1] "Femme" "Femme" "Femme" "Homme" "Homme" "Homme"
```

2.2.1 Réaliser des jointures entre tableaux, les fonctions merge et aggregate

Les données utilisées dans cette section concernent les communes de Paris et de la petite couronne. L'importation des données possède la syntaxe suivante.

```r
dData99_07 <- read.csv("data/data99_07.csv", sep = ";")
dPop36_08 <- read.csv("data/pop36_08.csv", sep = ";")
```

La fonction `merge()` permet d'effectuer la jointure de deux tableaux de données en fonction d'une variable commune. Pour donner un exemple, on peut joindre les deux tables *dData99_07* et *dPop36_08* à l'aide de l'identifiant commun des communes, nommé *CODGEO* dans les deux tables.

```r
dAllINSEE <- merge(dData99_07, dPop36_08, by = "CODGEO")
```

Ici, les deux tables ont exactement les mêmes individus statistiques (i.e. les lignes se correspondent deux à deux), et le nom de la colonne commune est le même dans les deux tables. Pour effectuer l'instruction `merge()` dans le cas le plus général, il faudrait écrire :

```r
dAllINSEE <- merge(x = dData99_07, y = dPop36_08, by.x = "CODGEO", by.y = "CODGEO",
    all.x = TRUE, all.y = TRUE)
```

Lorsque deux tableaux de données ne se correspondent que partiellement (individus différents, données manquantes etc.), on utilise la fonction `cbind()`.

```r
# Création des deux tables
d1 <- data.frame(cbind(c("A", "B", "D"), c(4, 3, 2)))
d2 <- data.frame(cbind(c("A", "B", "C"), c(10, 19, 28)))

# Ajout des identifiants des colonnes
names(d1) <- c("id", "var1")
names(d2) <- c("id", "var2")

# Jointure en conservant l'ensemble des lignes
d3 <- merge(d1, d2, by = "id", all = TRUE)
d3

##   id var1 var2
## 1  A    4   10
## 2  B    3   19
## 3  D    2 <NA>
## 4  C <NA>   28


# Jointure en conservant les lignes de la table de gauche
d4 <- merge(d1, d2, by = "id", all.x = TRUE)
d4
```

```
##   id var1 var2
## 1  A    4   10
## 2  B    3   19
## 3  D    2 <NA>
```

```
# Jointure en conservant les lignes de la table de droite
d5 <- merge(d1, d2, by = "id", all.y = TRUE)
d5
```

```
##   id var1 var2
## 1  A    4   10
## 2  B    3   19
## 3  C <NA>   28
```

```
# Jointure en ne conservant que les lignes communes
d6 <- merge(d1, d2, by = "id")
d6
```

```
##   id var1 var2
## 1  A    4   10
## 2  B    3   19
```

La fonction `aggregate()` permet quant à elle de calculer facilement des sommes ou moyennes par modalité de variable, par exemple ici des départements. Les arguments de la fonction sont les suivants : `aggregate(data, by = list(variable), FUN = function)` avec

- `data` = données à traiter,
- `variable` = valeur utilisée pour grouper les lignes,
- `fun` = fonction utilisée pour l'agrégation.

```
# l'argument by prend forcément une liste ou un array en entrée.
dAllINSEE$dep <- substr(dAllINSEE$CODGEO, 1, 2)
dAgg <- aggregate(dAllINSEE[, c("EMPLOI99", "POP1999")], by = list(dAllINSEE$dep),
    FUN = sum, na.rm = T)
```

Ce qui permet après jointure et agrégation de comparer le ratio d'emploi par habitant dans les quatre départements de l'agglomération parisienne :

```
# with(data,expression) = permet d'evaluer une 'expression' sur un tableau
# de données 'data' Nombre d'emplois pour un résident :
dAgg$ratio <- with(dAgg, EMPLOI99/POP1999)
dAgg
```

```
##   Group.1 EMPLOI99 POP1999  ratio
## 1      75  1656036 2125246 0.7792
## 2      92   810226 1428881 0.5670
## 3      93   467702 1382861 0.3382
## 4      94   463520 1227250 0.3777
```

On constate par ce biais l'inégale répartition spatiale des emplois, Paris étant deux fois plus dotée en emplois que la Seine-Saint-Denis et le Val-de-Marne, relativement à sa population.

2.3 Manipulation avancée de données

Il est courant de récupérer des données dans un format différent de celui indiqué pour la réalisation d'un traitement quelconque. La plupart du temps fastidieuses et impossibles à produire manuellement, ces opérations qui consistent à réaliser une sélection complexe des données ou à faire une transformation de la structure de présentation des données peuvent être réalisées beaucoup plus facilement à condition d'utiliser des fonctions spécifiques. Certaines de ces fonctions (`subset()` par exemple) sont disponibles dans `r-base` et d'autres dans des *packages* additionnels (`sqldf`, `reshape2` et `plyr`).

2.3.1 Selection de sous-tables et requêtes *SQL*

La fonction `subset()` est utilisée ici pour effectuer des selections simples dans des tables : choix de lignes selon un critère, choix de colonnes, etc.

Dans l'exemple ci-dessous on choisit les lignes en fonction d'un département (93) et de la taille, mesurée par la population résidante : on choisit les villes de plus de 50 000 habitants.

```
# Sélection des colonnes
dRefColumn <- subset(dPop36_08, select = c("LIBELLE","POP1936","POP2008"))
head(dRefColumn)

##                    LIBELLE POP1936 POP2008
## 1 Paris 1er Arrondissement   37062   17440
## 2  Paris 2e Arrondissement   41445   21793
## 3  Paris 3e Arrondissement   63571   34824
## 4  Paris 4e Arrondissement   62547   27977
## 5  Paris 5e Arrondissement   97396   62143
## 6  Paris 6e Arrondissement   81403   44322


# Sélection des lignes
dRef93 <- subset(dPop36_08, substr(CODGEO,1,2) == '93' & POP2008 > 50000)
head(dRef93)

##    CODGEO          LIBELLE POP1936 POP1954 POP1962 POP1968 POP1975 POP1982
## 57  93001    AUBERVILLIERS   55871   58740   70632   73695   72976   67719
## 58  93005 AULNAY-SOUS-BOIS   31763   38534   47507   61521   78137   75996
## 60  93007  LE BLANC-MESNIL   21660   25363   35708   48487   49107   47037
## 62  93010            BONDY   20539   22411   38039   51652   48333   44301
## 67  93029           DRANCY   42938   50654   65890   68467   64430   60183
## 69  93031 EPINAY-SUR-SEINE   15889   17611   34167   41774   46578   50314
##    POP1990 POP1999 POP2008 SURF
## 57   67557   63136   74528  457
## 58   82314   80021   82188 1576
## 60   46956   46936   50668  756
## 62   46676   46826   53259  551
## 67   60707   62263   66194  752
## 69   48762   46409   52689  423
```

Certains pourront trouver cette formulation de requêtes fastidieuse ou complexe. Le *package* `sqldf` permet d'utiliser le langage normalisé *SQL (Structured Query Language)* pour formuler des requêtes. Cela requiert de se conformer à ce formalisme, mais on gagne en lisibilité et en compréhension sur le long terme. De plus, le langage *SQL* permet une interrogation *via* un langage plus expressif que les simples selections proposées par R.

Avec le *package* `sqldf` il est possible de choisir directement dans la table `data99_07` les communes qui correspondent à la selection qu'on vient d'effectuer dans l'autre table. On indique l'attribut de méthode `method =raw` à `sqldf` pour éviter que `sqldf` ne nous renvoie des colonnes de type *factor* après sélection.

```
# Chargement de la librairie
library("sqldf")

dGrandesCommunes93 <- sqldf("select CODGEO,POUV99,ACTOCC99
                            from dData99_07
                            where CODGEO in (select CODGEO from dRef93)",
                            method ="raw" )
```

2.3.2 Formatage des tableaux

Format des données

Lorsqu'on travaille sur des tables 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 la distinction entre individu et variable n'a pas de sens.

De façon générale, on distingue deux types de format de données : le format *wide*, qui est celui que nous avons décrit jusqu'alors (une ligne représente un individu statistique, et les colonnes les variables), et le format *long*, qui présente un format des données plus éclaté. Si on manipule des données longitudinales par exemple (la même variable observée à intervalle de temps régulier), on aura ainsi le plus souvent deux options de stockage (cf. figure 2.3.2).

- 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, il y a ainsi plus de lignes que dans le format *wide*.

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.

Changer la structure d'un tableau

Il est possible d'utiliser la fonction `reshape()` contenue dans `r-base` pour transformer des données au format *long* vers *wide*, et inversement. Cet exemple illustre le passage d'un format *wide* à un format *long*, mais l'inverse est possible en changeant l'argument `direction` par *wide*

```
# Table de structure de type wide
head(dPop36_08)
```

```
##   CODGEO                 LIBELLE POP1936 POP1954 POP1962 POP1968 POP1975
## 1  75101 Paris 1er Arrondissement   37062   37330   36543   32332   22793
## 2  75102  Paris 2e Arrondissement   41445   41744   40864   35357   26328
## 3  75103  Paris 3e Arrondissement   63571   64030   62680   56252   41706
## 4  75104  Paris 4e Arrondissement   62547   62998   61670   54029   40466
## 5  75105  Paris 5e Arrondissement   97396   98099   96031   83721   67668
## 6  75106  Paris 6e Arrondissement   81403   81991   80262   70891   56331
##   POP1982 POP1990 POP1999 POP2008 SURF
## 1   18509   18360   16888   17440  183
## 2   21203   20738   19585   21793   99
## 3   36094   35102   34248   34824  117
## 4   33990   32226   30675   27977  160
## 5   62173   61222   58849   62143  254
## 6   48905   47891   44919   44322  215
```

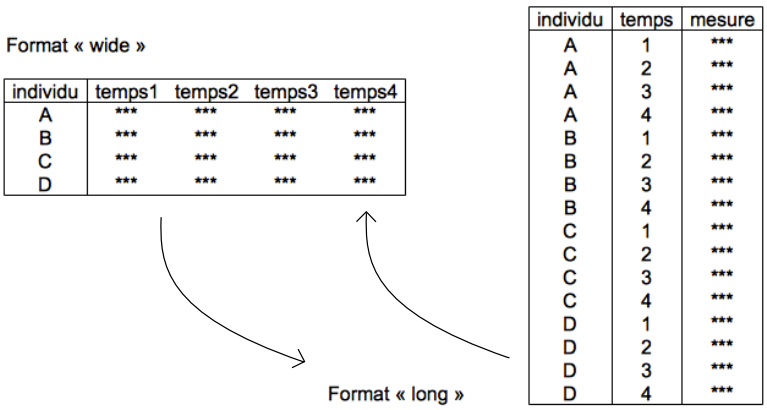


FIGURE 2.1 – Format *long* et format *wide* pour des données longitudinales

```
# Table transformée en structure long
head(reshape(dPop36_08,
        varying = c("POP1936","POP1954","POP1962","POP1968",
                    "POP1975","POP1982","POP1990","POP2008"),
        v.names = "population",
        timevar = "variable",
        direction = "long",
        sep = ""))
```

```
##     CODGEO                  LIBELLE POP1999 SURF variable population id
## 1.1  75101 Paris 1er Arrondissement   16888  183        1      37062  1
## 2.1  75102  Paris 2e Arrondissement   19585   99        1      41445  2
## 3.1  75103  Paris 3e Arrondissement   34248  117        1      63571  3
## 4.1  75104  Paris 4e Arrondissement   30675  160        1      62547  4
## 5.1  75105  Paris 5e Arrondissement   58849  254        1      97396  5
## 6.1  75106  Paris 6e Arrondissement   44919  215        1      81403  6
```

Utilisation de la fonction `melt()`

La commande `reshape()` est toutefois difficile à manipuler. Pour simplifier ces opérations la suite des manipulations sera faite en utilisant le *package* `reshape2` qui non seulement rend plus abordable cette fonction au travers d'un certain nombre de commandes, mais étend aussi son utilisation.

Toutefois, à la différence de la commande `r-base`, `melt()` propose un nouveau format de données s'appuyant sur l'objet R *data.frame*. Cette fonction permet de préparer les données en vue d'une utilisation avec d'autres commandes offertes par le même *package*, comme la fonction `dcast()` que nous verrons par la suite.

```
library(reshape2)
```

```
# On spécifie d'une part les variables à conserver (id) et d'autre part la
# variable pivot (variable.name)
dMelted3608 <- melt(dPop36_08, id = c("CODGEO", "LIBELLE", "SURF"), variable.name = "Year")

# La colonne Year contient les identifiants des années
head(dMelted3608)
```

```
##   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
```

Utilisation de la fonction `cast()`

La fonction `cast()` inclut toute une famille de commandes : `acast()` ou `dcast()` par exemple prennent en entrées des données re-formulées au format *long* par `melt()`, pour ensuite les retransformer au format de données voulu par l'utilisateur. Dans l'exemple qui suit `dcast()` permet de renvoyer des *data.frame*, contrairement à `acast()` utilisée pour renvoyer des objets de type *array* ou *matrix*.

Voici un autre exemple de manipulation de format permettant cette fois ci de revenir au tableau initial *wide*, avant son re-positionnement au format *long* par la fonction `melt()`. La formule utilisé par la fonction `cast()` est du type $column\_variable1 + n \sim row\_variable1 + n$. On remarque aussi la formulation spéciale « ... » qui indique à la fonction de prendre en compte toute les colonnes.

```
dByCodeGeo <- dcast(dMelted3608, ... ~ Year)
head(dByCodeGeo)
```

```
##   CODGEO                 LIBELLE SURF POP1936 POP1954 POP1962 POP1968
## 1  75101 Paris 1er Arrondissement  183   37062   37330   36543   32332
## 2  75102  Paris 2e Arrondissement   99   41445   41744   40864   35357
## 3  75103  Paris 3e Arrondissement  117   63571   64030   62680   56252
## 4  75104  Paris 4e Arrondissement  160   62547   62998   61670   54029
## 5  75105  Paris 5e Arrondissement  254   97396   98099   96031   83721
## 6  75106  Paris 6e Arrondissement  215   81403   81991   80262   70891
##   POP1975 POP1982 POP1990 POP1999 POP2008
## 1   22793   18509   18360   16888   17440
## 2   26328   21203   20738   19585   21793
## 3   41706   36094   35102   34248   34824
## 4   40466   33990   32226   30675   27977
## 5   67668   62173   61222   58849   62143
## 6   56331   48905   47891   44919   44322
```

Pour avoir le code géographique en colonne et le temps en ligne :

```
dByYear <- dcast(dMelted3608, Year ~ LIBELLE + CODGEO)
head(dByYear[, 0:4])
```

```
##      Year ABLON-SUR-SEINE_94001 ALFORTVILLE_94002 ANTONY_92002
## 1 POP1936                 2193             30078        19690
## 2 POP1954                 3220             30195        24512
## 3 POP1962                 5086             32332        46483
## 4 POP1968                 5692             35023        56638
## 5 POP1975                 5531             38057        57540
## 6 POP1982                 5264             36231        54610
```

```
# ou
dByYear <- dcast(dMelted3608, Year ~ CODGEO)
```

Exemple du distancier entre communes de la petite couronne parisienne

Afin d'illustrer la section précédente, la question du stockage de distances entre couples de communes sera abordée. Ce stockage se fait principalement de deux manières :

- sous forme de matrice, les lignes sont les origines, les colonnes sont les destinations et le contenu des cellules représente la distance entre chaque couple origine-destination,
- sous forme de table, avec trois colonnes : l'origine, la destination et la distance.

Le schéma 2.3.2 illustre ces deux formats de données et rappelle les fonctions permettant de passer de l'un à l'autre.

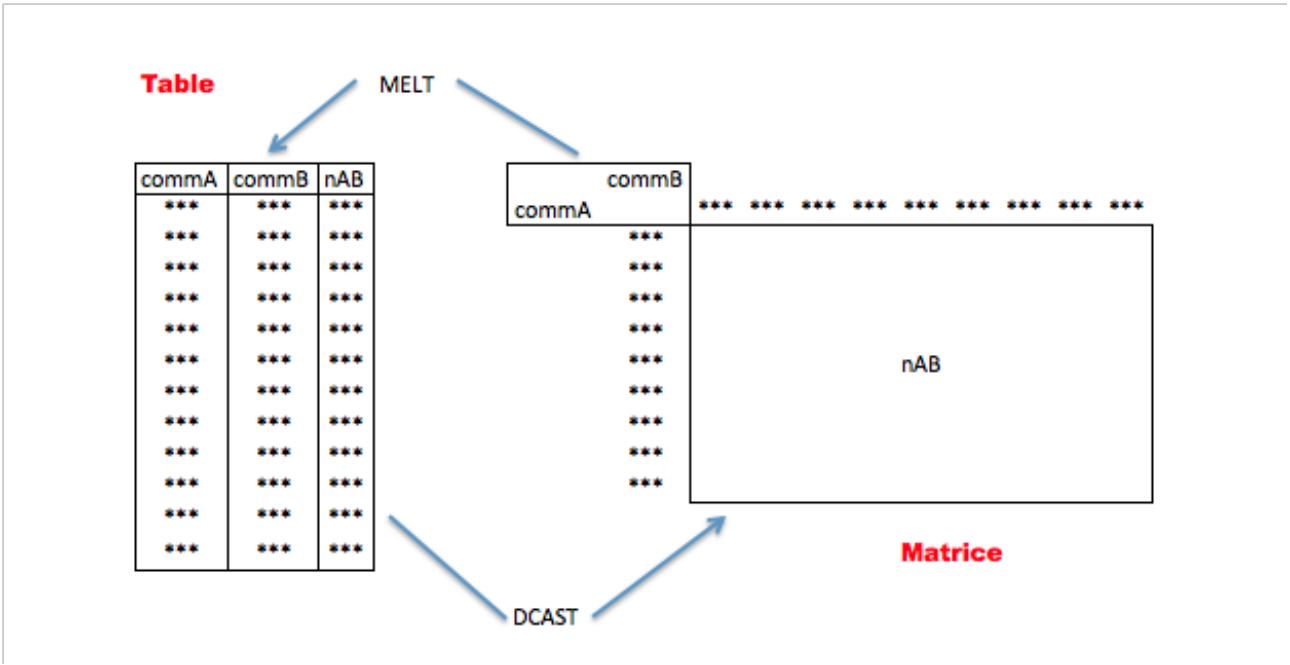


FIGURE 2.2 – Format table et format matriciel pour un distancier

On va ici travailler à partir de la matrice des migrations résidentielles de l'INSEE contenue dans le fichier `dMobResid2008.txt`.

```
dMob <- read.delim("data/dMobResid2008.txt", dec = ",")
head(dMob)
```

```
##   CODGEO                   LIBGEO DCRAN                  L_DCRAN   NBFLUX
## 1  75101 Paris 1er Arrondissement 75101 Paris 1er Arrondissement 11301.64
## 2  75101 Paris 1er Arrondissement 75102  Paris 2e Arrondissement   119.48
## 3  75101 Paris 1er Arrondissement 75103  Paris 3e Arrondissement    99.25
## 4  75101 Paris 1er Arrondissement 75104  Paris 4e Arrondissement    64.59
## 5  75101 Paris 1er Arrondissement 75105  Paris 5e Arrondissement   105.92
## 6  75101 Paris 1er Arrondissement 75106  Paris 6e Arrondissement    81.27
```

On transforme le fichier pour qu'il affiche une matrice des flux à l'aide des fonctions `melt()` et `cast()`. dans l'exemple ci dessous le tableau de donnée est quasiment au bon format, mais pour pouvoir utiliser `cast()`, le passage au format avec `melt()` est nécessaire : la fonction `cast()` a besoin de connaitre les identifiants et les mesures.

```
meltedData <- melt(dMob, id = c("CODGEO", "DCRAN"), measure = "NBFLUX")

flowDataFrame <- dcast(meltedData, CODGEO ~ DCRAN)
```

On utilise le résultat obtenu en sélectionnant les flux dont l'origine est le premier arrondissement de Paris.

```
flowDataFrame <- dcast(meltedData, CODGEO ~ DCRAN, subset = function() {
    (CODGEO == "75101")
})
```

On peut également créer des sous-ensembles de cette table en sélectionnant les cas selon un seuil, par exemple les flux intermunicipaux supérieurs à 100.

```
flowDataFrame <- dcast(meltedData, CODGEO ~ DCRAN, subset = function() {
    (value > 100)
})
```

ou encore afficher uniquement un sous ensemble de données en sélectionannt les cas avec des expressions régulières, par exemple uniquement les arrondissement parisiens.

```
flowDataFrame <- dcast(meltedData, CODGEO ~ DCRAN, subset = function() {
    regexpr("7510*", CODGEO)
})
```

Pour revenir à un tableau au format *wide* de départ, on utilise `melt()` :

```
dMatrixToWide <- melt(flowDataFrame, id = "CODGEO", variable.name = "DCRAN",
    na.rm = TRUE)
dMatrixToWideStep2 <- dcast(dMatrixToWide, CODGEO ~ ...)
head(dMatrixToWideStep2[, 1:6])

##   CODGEO   75101   75102   75103   75104   75105
## 1  75101 11301.6   119.5      NA      NA   105.9
## 2  75102   184.8 13514.5   144.1   121.3      NA
## 3  75103   118.2   174.7 22893.2   308.9   133.4
## 4  75104      NA      NA   155.4 18608.5   105.6
## 5  75105      NA      NA   205.3   214.2 40276.4
## 6  75106      NA      NA      NA      NA   317.1
```

2.3.3 Etude des structures itératives en R : les boucles

La boucle `while` et le test conditionnel `if` 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 de réaliser des instructions en fonction de l'état des variables à un moment donné.

Dans R, il est possible d'itérer dans les tables de données en les utilisant comme des matrices, et de changer les valeurs de lignes, colonnes ou cellules particulières.

Boucles `for` et `while`

La boucle `for` (équivalent de « pour chacun de ») permet de répéter une instruction sur un nombre d'itérations déterminé à l'avance. La boucle `while` (équivalent de « 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.

Boucle for

```
for (i in 1:n)
{
 instruction répétée n fois
 les valeurs de l'entier i sont comprises entre 1 et n
}
```

Boucle while

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

Exemple d'application : emploi d'une double boucle `for` pour calculer la distance entre communes

En utilisant la table `dData99_07` on calcule un distancier entre communes. Pour cela on parcourt deux fois la table dans deux boucles imbriquées, afin de traiter tous les couples possibles de communes. Dans cette boucle on utilise la notation [ligne,colonne] pour accéder aux données de la table, sachant qu'il est également possible de la filtrer par ligne et par colonne.

```
(dData99_07[dData99_07$CODGEO == 75101, ]$X)
# est équivalent de :
dData99_07[1, 3]
```

Pour faire référence à la valeur de la Lième ligne de la Cième colonne de la table `dData99_07` on utilise la formulation `dData99_07[L,C]`.

Dans cet algorithme, on utilise les colonnes suivantes :

- la première colonne contient la liste des identifiants des communes
- la troisième colonne contient les coordonnées X des communes
- la quatrième colonne contient les coordonnées Y des communes

Il existe plusieurs méthodes pour construire le distancier entre communes :

Méthode 1 : construction d'une table

Pour stocker cette table on créé d'abord trois vecteurs vides :

```
lCommA <- vector()
lCommB <- vector()
lDistAB <- vector()
```

On remplit ces vecteurs progressivement, à l'aide d'une double boucle `for`. Nous calculons ici à chaque pas la distance euclidienne entre commune, mais nous aurions pu faire un autre choix. Les chapitres 7 et 10 donnent à voir un panel plus important de distances possibles.

La méthodologie retenue ici consiste à utiliser un compteur numérique, k, qui est mis à jour à chaque passage dans la boucle la plus petite.

```
k <- 1
n <- nrow(dData99_07)
for (i in 1:(n - 1)) {
    for (j in (i + 1):n) {
        lCommA[k] <- dData99_07[i, 1]
```

```
        lCommB[k] <- dData99_07[j, 1]
        lDistAB[k] <- sqrt((dData99_07[i, 3] - dData99_07[j, 3])^2 + (dData99_07[i,
            4] - dData99_07[j, 4])^2) * 0.1  # Calcul de la distance (en km).
        k <- k + 1
    }
}
```

Puis on réunit ces trois vecteurs en un *data.frame*.

```
distancierParisPC = data.frame(cbind(lCommA, lCommB, lDistAB))
names(distancierParisPC) = c("commA", "commB", "distAB")
head(round(distancierParisPC, 2))

##   commA commB distAB
## 1 75101 75102   0.58
## 2 75101 75103   0.80
## 3 75101 75104   1.35
## 4 75101 75105   1.90
## 5 75101 75106   1.78
## 6 75101 75107   1.96
```

Méthode 2 : construction d'une matrice

On crée dans ce second exemple une matrice complète avec la fonction `melt()` pour obtenir à nouveau une table.

```
n <- nrow(dData99_07)
# Initialisation de la matrice à 0
mDistance <- matrix(0, nrow = n, ncol = n, byrow = FALSE, dimnames = list(dData99_07$CODGEO,
    dData99_07$CODGEO))
for (i in 1:(n - 1)) {
    for (j in (i + 1):n) {
        temp <- sqrt((dData99_07[i, 3] - dData99_07[j, 3])^2 + (dData99_07[i,
            4] - dData99_07[j, 4])^2) * 0.1
        mDistance[i, j] <- temp
        mDistance[j, i] <- temp  # Création d'une matrice symétrique
    }
}
head(round(mDistance[, 1:6], 2))

##       75101 75102 75103 75104 75105 75106
## 75101  0.00  0.58  0.80  1.35  1.90  1.78
## 75102  0.58  0.00  1.21  1.91  2.43  2.06
## 75103  0.80  1.21  0.00  0.92  2.02  2.36
## 75104  1.35  1.91  0.92  0.00  1.35  2.16
## 75105  1.90  2.43  2.02  1.35  0.00  1.30
## 75106  1.78  2.06  2.36  2.16  1.30  0.00

library(reshape2)
tmDistance <- melt(mDistance, id = "row.names")
head(round(tmDistance, 2))

##    Var1  Var2 value
## 1 75101 75101  0.00
## 2 75102 75101  0.58
```

```
## 3 75103 75101  0.80
## 4 75104 75101  1.35
## 5 75105 75101  1.90
## 6 75106 75101  1.78
```

2.4 Code

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

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

# création d'objets
vExemple1 <- c(1, 2, 3, 4, 5)
vExemple2 <- c("CP", "CE1", "CE2", "CM1", "CM2")

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

dExemple <- data.frame(vExemple1, vExemple2, stringsAsFactors=FALSE)

colnames(dExemple)
colnames(dExemple) <- c("ID", "CLASSE")
dExemple <- data.frame("ID" = vExemple1, "CLASSE" = vExemple2, stringsAsFactors=FALSE)

dExemple2 <- data.frame("col1" = vExemple1, "col2" = c(2,5,8,7,8), stringsAsFactors=FALSE)

# transformation d'objets
mExemple2 <- as.matrix(dExemple2) # transformation sans perte
MonVecteur <- as.numeric(MonFacteur) # transformation avec perte des étiquettes
vExemple2 <- as.vector(mExemple2) # perte d'une dimension, perte des noms des champs

# Sélection
vExemple1[2]
mExemple[2, 3]
mExemple[ , 3]
mExemple[2, ]
dExemple$CLASSE
dExemple[ , 2]
dExemple[3, 2]

# Sélection et assignation
vExemple2[1] <- "Cours primaire"
vExemple2[2:3] <- "Cours élémentaire"
vExemple2[4:5] <- "Cours moyen"

mExemple[2, 3] <- 99
mExemple

# lister les objets
ls()

# effacer un seul objet
```

```r
rm(MonObjet)

# effacer tous les objets
rm(list = ls())

data(cars) # jeu de données servant d'exemple contenu dans la base R
str(cars) # renseigne sur la classe et le contenu de l'objet "cars"

# Création d'un data.frame
dIndividus <- data.frame(c(1, 2, 3, 4, 5), c(12, 17, 24, 45, 8), c("H", "H", "F", "F", "F"))
colnames(dIndividus) <- c("ID", "AGE", "SEXE")
colnames(dIndividus)

# Sélection et assignation conditionnelle
dIndividus$CLASSE[dIndividus$AGE < 18] <- "Mineur"
dIndividus$CLASSE[dIndividus$AGE >= 18] <- "Majeur"
vAge <- as.vector(dIndividus$AGE)
vSexe <- as.vector(dIndividus$SEXE)
vCombinaison <- vector(mode="character", length=5)
vCombinaison[vAge < 18 & vSexe == "H"] <- "Homme mineur"
vCombinaison[vAge < 18 & vSexe == "F"] <- "Femme mineure"
vCombinaison[vAge >= 18 & vSexe == "F"] <- "Femme majeure"
subset(dIndividus, dIndividus$AGE == "17" | dIndividus$AGE == "24")
dIndividus$MAJMIN <- ifelse(dIndividus$AGE >= 18, "Majeur", "Mineur")

# Trier un vecteur
vAgeCroissant <- sort(vAge)
head(vAge)
head(vAgeCroissant)

# La syntaxe équivalente avec order, bien que peu utilisée
vAgeCroissant <- vAge[order(vAge)]

# Trier dans un autre ordre
vAgeDecroissant <- sort(vAge, decreasing = TRUE)

# Trier un tableau
dIndividusClasses <- dIndividus[order(dIndividus$AGE), ]
dCars <- cars[order(cars$speed, cars$dist), ]

# Trier un facteur
vSexeLevels <- sort(MonFacteur)
vSexeLabels <- sort(as.character(MonFacteur))

# Manipulation avancée
dData99_07 <- read.csv("data/data99_07.csv", sep=";")
dPop36_08 <- read.csv("data/pop36_08.csv", sep=";")
dAllINSEE = merge(dData99_07,dPop36_08, by = "CODGEO")
dAllINSEE = merge(x = dData99_07,
                  y = dPop36_08,
                  by.x = "CODGEO",
                  by.y = "CODGEO",
                  all.x = TRUE,
                  all.y = TRUE)

# Création des deux tables
d1 <- data.frame(cbind(c("A","B","D"),c(4,3,2)))
d2 <- data.frame(cbind(c("A","B","C"),c(10,19,28)))
```

```r
# Ajout des identifiants des colonnes
names(d1) <- c("id","var1")
names(d2) <- c("id","var2")

# Jointure en conservant l'ensemble des lignes
d3 <- merge(d1, d2, by = "id", all=TRUE)

# Jointure en conservant les lignes de la table de gauche
d4 <- merge(d1, d2, by = "id", all.x=TRUE)

# Jointure en conservant les lignes de la table de droite
d5 <- merge(d1, d2, by = "id", all.y=TRUE)

# Jointure en ne conservant que les lignes communes
d6 <- merge(d1, d2, by = "id")

# Agrégation
dAllINSEE$dep <- substr(dAllINSEE$CODGEO,1,2)
dAgg <- aggregate(dAllINSEE[,c("EMPLOI99","POP1999")],
                  by = list(dAllINSEE$dep),
                  FUN = sum,
                  na.rm=T)
dAgg$ratio <- with(dAgg,EMPLOI99/POP1999)

# Sélection des colonnes
dRefColumn <- subset(dPop36_08, select = c("LIBELLE","POP1936","POP2008"))

# Sélection des lignes
dRef93 <- subset(dPop36_08, substr(CODGEO,1,2) == '93' & POP2008 > 50000)

# Requêtes SQL
library("sqldf")

dGrandesCommunes93 <- sqldf("select CODGEO,POUV99,ACTOCC99
                            from dData99_07
                            where CODGEO in (select CODGEO from dRef93)",
                            method ="raw")

# Transformation des formats de tableaux (wide / long)

# Table transformée de wide à long
head(reshape(dPop36_08,
             varying = c("POP1936","POP1954","POP1962","POP1968",
                         "POP1975","POP1982","POP1990","POP2008"),
             v.names = "population",
             timevar = "variable",
             direction = "long",
             sep = ""))

library(reshape2)

# Préciser les variables à conserver (id) et la variable pivot (variable.name)
dMelted3608 <- melt(dPop36_08,
                    id = c("CODGEO","LIBELLE","SURF"),
                    variable.name = "Year")

dByCodeGeo <- dcast(dMelted3608, ... ~ Year)
dByYear <- dcast(dMelted3608, Year ~ LIBELLE + CODGEO)
```

```
dByYear <- dcast(dMelted3608, Year ~ CODGEO)

dMob <- read.delim("data/dMobResid2008.txt", dec=",")
meltedData <- melt(dMob, id = c("CODGEO","DCRAN"), measure = "NBFLUX")
flowDataFrame <- dcast(meltedData, CODGEO ~ DCRAN)
flowDataFrame <- dcast(meltedData, CODGEO ~ DCRAN, subset = function () {(CODGEO == "75101")})
flowDataFrame <- dcast(meltedData, CODGEO ~ DCRAN, subset = function () {(value > 100)})
flowDataFrame <- dcast(meltedData, CODGEO ~ DCRAN, subset = function () {regexpr("7510*", CODGEO)})
dMatrixToWide <- melt(flowDataFrame, id = "CODGEO", variable.name = "DCRAN", na.rm = TRUE)
dMatrixToWideStep2 <- dcast(dMatrixToWide, CODGEO ~ ...)

# Boucles

lCommA <- vector()
lCommB <- vector()
lDistAB <- vector()

k <- 1
n = nrow(dData99_07)
for(i in 1:(n-1)) {
  for(j in (i+1):n) {
    lCommA[k] <- dData99_07[i,1]
    lCommB[k] <- dData99_07[j,1]
    lDistAB[k] <- sqrt((dData99_07[i,3] - dData99_07[j,3])^2 +
                         (dData99_07[i,4] - dData99_07[j,4])^2) * 0.1
    k <- k + 1
  }
}

distancierParisPC <- data.frame(cbind(lCommA,lCommB,lDistAB))
names(distancierParisPC) <- c("commA", "commB", "distAB")
head(round(distancierParisPC,2))

n  <-  nrow(dData99_07)

# Initialisation de la matrice à 0
mDistance <- matrix(0, nrow = n, ncol= n, byrow=FALSE,
                    dimnames = list(dData99_07$CODGEO,dData99_07$CODGEO))
for(i in 1:(n-1)) {
  for(j in (i+1):n) {
    temp <- sqrt((dData99_07[i,3] - dData99_07[j,3])^2 +
                   (dData99_07[i,4] - dData99_07[j,4])^2) * 0.1
    mDistance[i,j] <- temp
    mDistance[j,i] <- temp # Création d'une matrice symétrique
  }
}

tmDistance <- melt(mDistance, id = "row.names")
```

Chapitre 3

Programmation

Objectifs

Ce chapitre traite des techniques de programmation de niveau débutant ou intermédiaire, permettant une manipulation experte de tables de données (manipulations de matrices, traitement au niveau des cellules individuelles), ainsi que l'automatisation de procédures (fonctions et algorithmique).

Prérequis

Ce chapitre s'appuie directement sur le Chapitre 2 pour ce qui concerne la manipulation des données. Il ne requiert pas de prérequis mathématiques particuliers, puisqu'il introduit toutes les notions utiles (fonctions élémentaires, algorithmes). Ce chapitre est toutefois destiné à un public particulier, souhaitant maîtriser des fonctionnalités avancées de traitement des données sous R pour des développements spécifiques.

Packages nécessaires

Dans ce chapitre on utilise à la fois des fonctions existant de façon native dans R, mais également deux *packages* (`reshape2` et `plyr`) développés par Hadley Wickham qui, utilisés conjointement, permettront d'aller plus loin dans la manipulation des données.

- *Package* `reshape2`.

  Cette extension permet de passer facilement d'un format de données « long » à un format « large » (voir section suivante). Habituellement les observations sont en ligne et les variables en colonnes. Reshape part du principe que ces variables peuvent être divisées en deux groupes : les identifiants, et les mesures. A partir de ce constat, ce *package* met à disposition des utilisateurs un certain nombre de commandes pour la transformation des structures de données.

- *Package* `plyr`

  Ce *package* tente de répondre à une stratégie relativement commune en analyse de données : découper (*split*) les données selon les modalités d'une variable, appliquer une fonction sur chacun des sous-ensembles et recombiner les résultats ainsi obtenus. Cette extension reprend les fonctions existantes de R (la famille `*apply`). Elle propose à l'utilisateur une interface de manipulation des entrées/sorties unifiées, avec une syntaxe d'utilisation grandement simplifiée, et des performances plus élevées sur les larges tableaux de données (parallélisation des traitements).

Données nécessaires

Dans ce chapitre les deux tableaux de données produits par l'INSEE sont utilisés :

- Feuille `data99_07` : données de référence pour 1999 et 2007 avec différentes variables sur la composition socio-professionnelle des communes de Paris et la petite couronne.

- Feuille `pop36_08` : données de population sans double compte des recensements de la population de 1936 à 2008 pour Paris et la petite couronne.

Pour une description plus précise du contenu des fichiers, voir le Chapitre 1.

Préparation des données

```
dData99_07 <- read.csv("data/data99_07.csv", sep = ";")
dPop36_08 <- read.csv("data/pop36_08.csv", sep = ";")
```

3.1 Manipulation des fonctions sous R

3.1.1 Les fonctions sous R

Dans les cas où l'on répète plusieurs fois la même opération, il peut s'avérer utile de créer des fonctions. Les fonctions sont des « boîtes » auxquelles on donne des valeurs d'entrées (arguments de la fonction) et desquelles on retire des valeurs de sortie (résultats de la fonction). Il n'y a pas de contraintes fortes sur les types d'objets R qui peuvent être en argument ou en sortie d'une fonction.

Définir une fonction

Un cas simple est celui d'une fonction d'un nombre réel, qui renvoie un nombre réel. L'instruction s'écrit de façon générale :

```
Definition
function (arguments) {
... traitements ...
return resultat
}
```

A titre d'exemple, l'instruction ci-dessous définit la fonction carré :

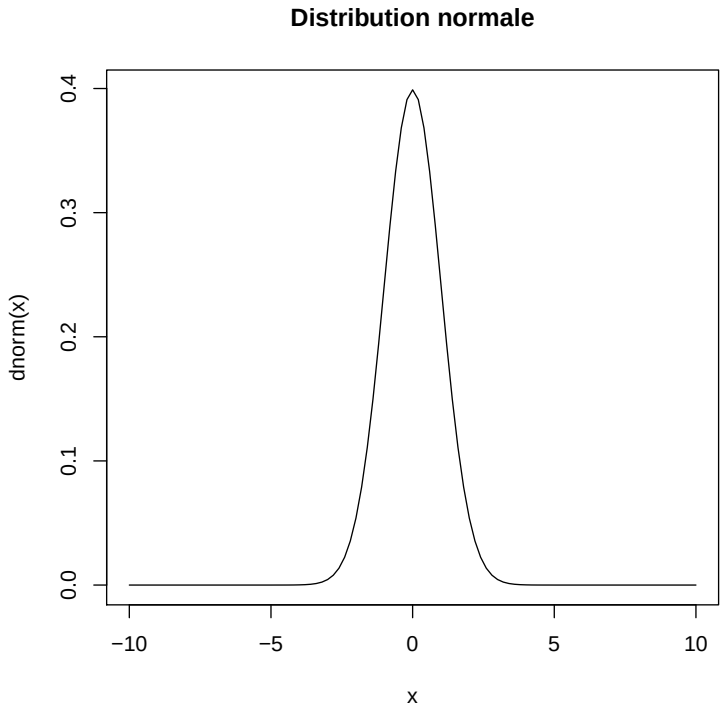
```
Carre <- function(x) {
    return(x^2)
}

Carre(8)

## [1] 64


# qui peut s'écrire également sous cette forme :

Carre <- function(x) (x^2)

Carre(8)

## [1] 64
```

Certaines fonctions sont déjà incluses dans R, ou dans des *packages* R. Souvent rangées par famille de fonctions, on citera par exemple la famille de fonctions capables de générer des distributions statistiques comme la loi normale (`dnorm()` pour la densité de probabilité et `pnorm()` pour la fonction de répartition).

Représenter une fonction

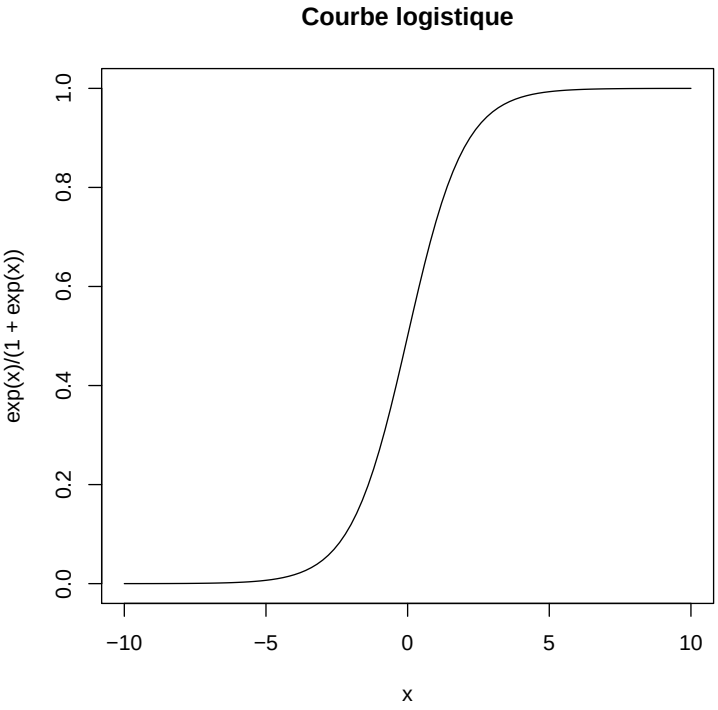
De même que les tables de données peuvent être représentées au moyen d'un `plot()`, les fonctions possèdent la plupart du temps des fonctionnalités de représentation graphique. La fonction `curve()` prend ainsi en argument une fonction et des bornes et affiche le graphe de cette fonction dans l'intervalle choisi. A titre d'exemple on peut représenter la densité de probabilité de la loi normale déjà rencontrée :

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



Dériver une fonction

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 $x \to ax^b$ est $x \to abx^{b-1}$ ce que retrouve bien R. La possibilité de dériver des fonctions est rendue possible par la fonction `D()` existant nativement dans R. L'exemple ci-dessous illustre cette possibilité.

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

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

3.1.2 Application 1 : 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 que la route 1 est plus courte (t1* = 10 minutes si le trafic est nul au lieu de t2* = 30 minutes), mais qu'il ne s'agit que d'une route de faible capacité (capacité c1 = 1000 véhicules par heure), alors que la route 2 est une route plus importante (capacité c2 = 3000 véhicules par heure). Donc si l'ensemble des 4000 véhicules dans cet exercice cherche à prendre la route 1, celle-ci sera bouchée 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. A l'aide de fonctions sous R, on peut chercher 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.

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 $q1 + q2$ fait le total de 4000 véhicules (il n'y a donc en réalité qu'une seule variable dans ce problème). Les temps de trajets en prenant en compte la congestion sont, d'après les équations classiques du *Bureau of Public Roads* :

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

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

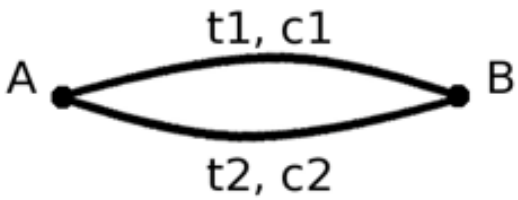


FIGURE 3.1 – Attributs des deux routes permettant d'aller du point A au point B : t1 = 10 min, t2 = 30 min, c1 = 1000 veh./h, c2 = 3000 veh./h.

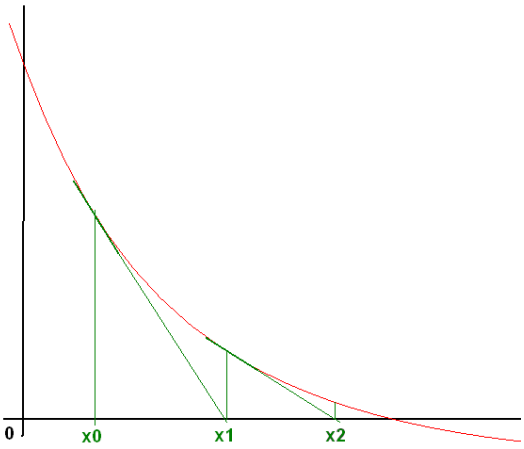


FIGURE 3.2 – Convergence de la méthode de Newton

En application de la théorie économique dite de l'équilibre de Wardrop, dans la configuration qui minimise le temps de trajet des individus, on doit avoir $t1 = t2$. On va 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 :

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

On cherche donc x tel que $f(x) = 0$ pour une certaine fonction f. On décide ainsi d'une première estimation x_0 et on va ensuite chercher successivement des estimations plus précises d'un x tel que $f(x) = 0$ par application de la méthode de Newton.

Si x_0 est la première estimation du nombre de voitures sur la voie 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 : $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

On aborde ici un exemple numérique où cette convergence est effective, mais on ne cherche pas à 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 [Bonnel 2002].

On propose une résolution numérique de ce problème : on définit d'abord les valeurs des constantes et les fonctions.

Définition des constantes :

```
alpha <- 0.15
beta <- 4
t1 <- 10
t2 <- 30
c1 <- 1000
```

```
c2 <- 3000
Q <- 4000
```

Définition des 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.

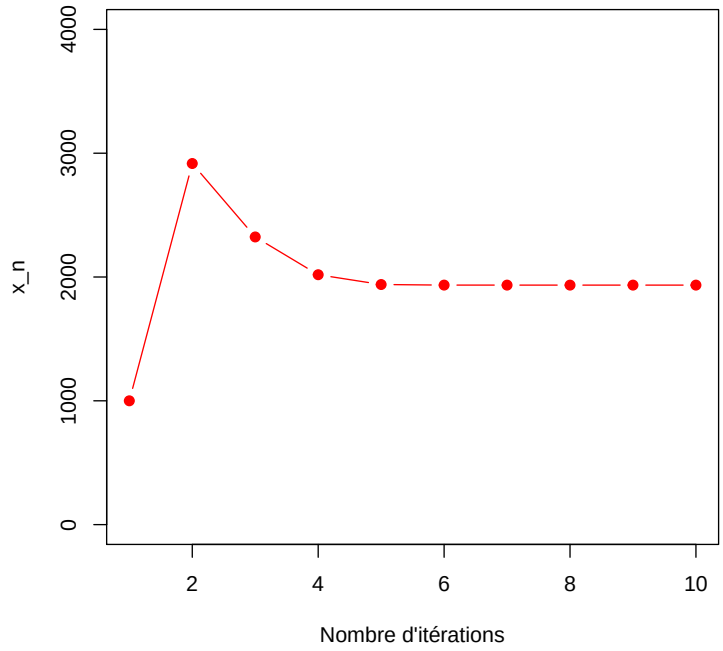
```
f1 <- function(x) (t1 * (1 + alpha * (x/c1)^beta) - t2 * (1 + alpha * ((Q -
    x)/c2)^beta))
f2 <- function(xx) eval({
    x <- xx
    (D(expression(t1 * (1 + alpha * (x/c1)^beta) - t2 * (1 + alpha * ((Q - x)/c2)^beta)),
        "x"))
})
f3 <- function(x) x - f1(x)/f2(x)
```

Pour commencer la résolution numérique, on décide d'une première estimation « à vue de nez » 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.

```
L <- NULL
L[1] <- 1000
for (i in 2:10) {
    L[i] <- f3(L[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(L, type = "b", pch = 19, col = "red", ylim = c(0, 4000), xlab = "Nombre d'itérations",
    ylab = "x_n")
```




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

## [1] 31.01
```

Fonctions de plusieurs variables

Une fonction peut prendre en entrée plusieurs arguments, comme dans l'exemple ci-dessous.

```
s <- function(x, y) {
    x + y
}
s(Carre(3), Carre(4))

## [1] 25
```

De même une fonction peut renvoyer plusieurs variables :

```
trigo <- function(x) {
    c(cos(x), cos(x + pi/2), cos(x + pi), cos(x + 3 * pi/4))
}
trigo(pi/6)

## [1]  0.8660 -0.5000 -0.8660 -0.9659
```

Une foncion peut également prendre une liste en argument et renvoyer une liste ; on voit sur l'exemple ci-dessous que R est souple avec les formats d'objets utilisés puisqu'il convertit automatiquement un nombre en liste si c'est approprié.

```
transposer <- function(L) {
    L + length(L)
}
transposer(1:10)

##  [1] 11 12 13 14 15 16 17 18 19 20
```

3.1.3 Application 2 : la suite de Syracuse

Pour apprendre la manipulation des listes par des fonctions et des tests conditionnels, on va s'éloigner un peu du monde des villes et de la géographie. On va créer une liste correspondant à une suite mathématique : la suite de Syracuse. On part d'un nombre quelconque, puis 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 $(u_n)_n$ par :

$$\left\{ \begin{array}{rcl} u_{n+1} & = & \frac{u_n}{2} \quad si \quad u_n \quad pair \\ u_{n+1} & = & 3u_n + 1 \quad si \quad u_n \quad impair \end{array} \right. \tag{3.1}$$

Il semble (bien que personne ne l'ai démontré, ce qui est amusant vu que cela fait des siècles qu'on essaye) que quelle que soit la valeur de l'entier de départ, au bout d'un moment, on finit toujours par tomber sur le chiffre 1. On va ici tenter de le vérifier empiriquement sur un certain nombre de cas.

Pour ce faire, on crée une liste vide, à remplir (objet `NULL`). On initialisez ensuite la liste avec une valeur quelconque (rappel : L[i] est le ième terme de la liste L).

A l'aide d'une boucle `while`, on complète les valeurs de la suite : il faut tester si le nombre courant est pair ou impair, et selon le résultat de ce test calculer la valeur suivante. On affiche alors les termes successifs de la suite, en mettant l'axe des ordonnées sous forme logarithmique.

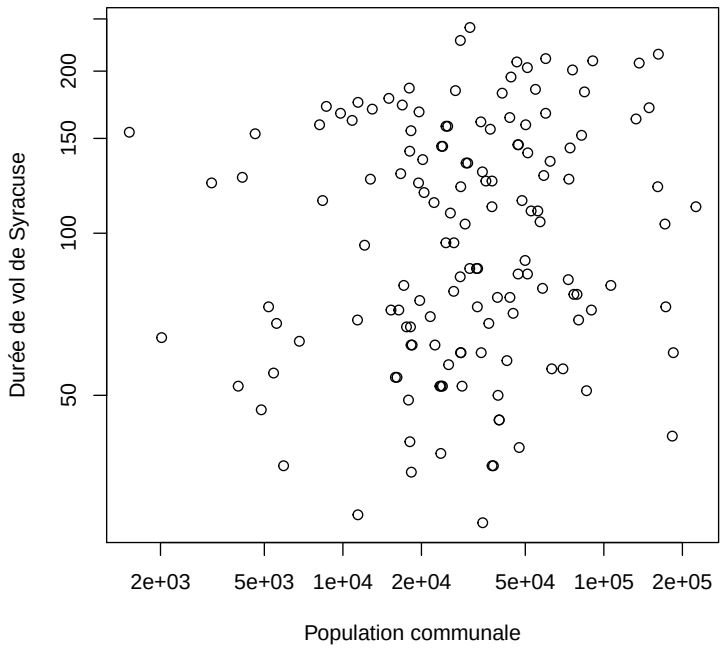
```
syracuse <- function(nombre) {
    L <- NULL
    L[1] <- nombre
    i <- 1
    # Test de parité
    while (L[i] != 1) {
        if (round(L[i]/2) == L[i]/2) {
            L[i + 1] <- L[i]/2
        } else {
            L[i + 1] <- 3 * L[i] + 1
        }
        i <- i + 1
    }
    L
}

dureeVolSyracuse <- function(x) {
    length(syracuse(x))  # La taille de la liste correspond à la durée du vol
}
```

On va maintenant calculer la durée des vols de Syracuse de façon automatique, pour une liste de valeurs de départ prises dans les tables de données de l'agglomération parisienne. On va déterminer la commune parisienne qui correspond, en partant de sa population en 1999, à la durée de vol la plus haute. Pour ce faire, on crée une fonction calculant les durées de vol pour un ensemble de points de départ.

```
dureeVolSyracuseListe <- function(L) {
    M <- NULL
    j <- 1
    while (j <= length(L)) {
        M[j] <- dureeVolSyracuse(L[j])
        j <- j + 1
    }
    M
}
```

On pourrait se dire que plus on part d'une population importante, plus le vol de Syracuse dure longtemps, mais le problème est plus complexe :

```
lVolsSyracuseParis <- dureeVolSyracuseListe(dPop36_08$POP1999)
plot(dPop36_08$POP1999, lVolsSyracuseParis, log = "xy", xlab = "Population communale",
    ylab = "Durée de vol de Syracuse")
```



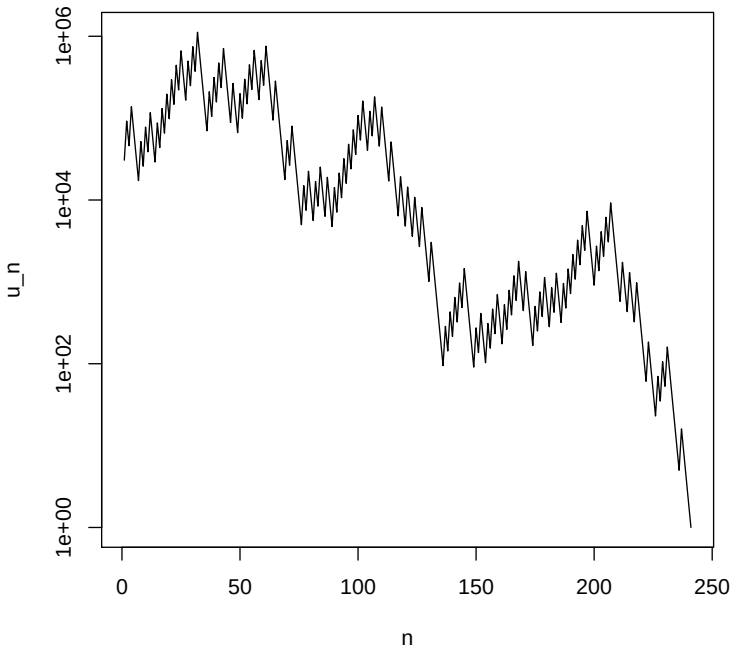

```
max(lVolsSyracuseParis)

## [1] 241
```

La plus haute durée de vol (241 iterations) est obtenue pour une commune assez peu peuplée : on va trouver laquelle par l'instruction suivante (remarquez l'utilisation de la fonction *order* qui donne le rang des termes d'une liste donnée).

```
dPop36_08[order(lVolsSyracuseParis, decreasing = TRUE)[1], ]

##   CODGEO                LIBELLE POP1936 POP1954 POP1962 POP1968 POP1975
## 4  75104 Paris 4e Arrondissement   62547   62998   61670   54029   40466
##   POP1982 POP1990 POP1999 POP2008 SURF
## 4   33990   32226   30675   27977  160
```

Il s'agit donc du quatrième arrondissement de Paris dont on peut représenter graphiquement le vol.

```
plot(syracuse(dPop36_08[order(lVolsSyracuseParis, decreasing = TRUE)[1], 10]),
    type = "l", log = "y", xlab = "n", ylab = "u_n")
```



3.1.4 Fonctions permettant des traitements en série

Un avantage de la souplesse des fonctions sous R est la possibilité de faire des traitements automatisés pour un très grand nombre de tables en entrée : on définit une fois pour toute une série d'instructions à faire avec une série de tables (préalablement formatées) et on peut ainsi calculer à la chaîne des valeurs de sorties sur toutes les tables. Nous allons illustrer cette fonctionnalité par le calcul d'un indice de Gini illustrant la répartition de la population entre communes au niveau départemental.

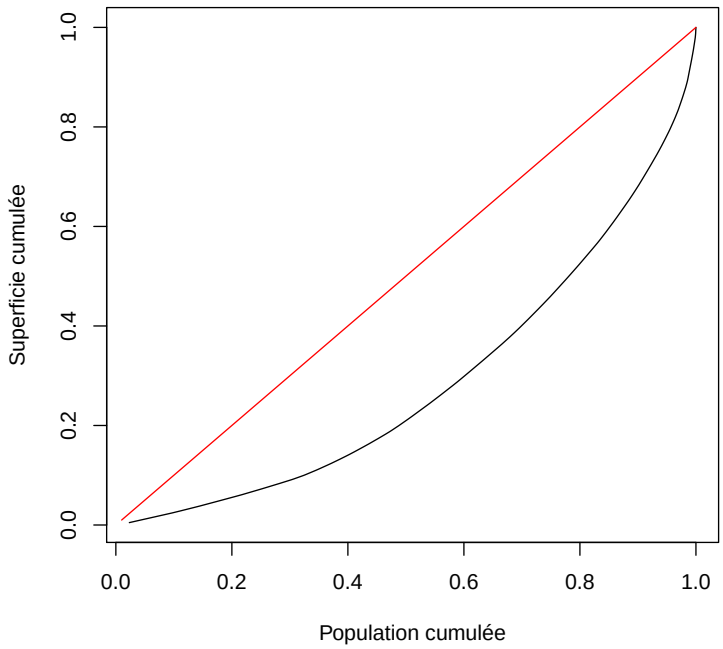
Tout d'abord, si on fait les traitements sur tout le territoire considéré :

Application 3 : courbe de Lorenz et indice de Gini

L'explication de la courbe de Lorenz et de l'indice de Gini n'étant pas au cœur de cette application, on renvoie aux manuels de géographie statistique ou à la page wikipédia pour plus de détails.

On ordonne les communes par populations croissantes :

```
dGini <- subset(dPop36_08, select = c("CODGEO", "LIBELLE", "POP2008", "SURF"))

dGini <- dGini[order(dGini$POP2008/dGini$SURF, decreasing = TRUE), ]
dGini$POP2008cum <- cumsum(dGini$POP2008)/sum(dGini$POP2008)
dGini$SURFcum <- cumsum(dGini$SURF)/sum(dGini$SURF)
plot(dGini$POP2008cum, dGini$SURFcum, type = "l", xlab = "Population cumulée",
    ylab = "Superficie cumulée")
# Affichage de la courbe correspondant à une égalité parfaite
lines(1:100/100, 1:100/100, col = "red")
```



On calcule l'intégrale sous la courbe à l'aide de la fonction suivante (méthode des rectangles) :

```
calculIntegraleApprochee01 <- function(vecteurX, vecteurY) {
    total <- vecteurY[1]/2 * vecteurX[1]
    i <- 2
    while (i <= length(vecteurX)) {
        total <- total + (vecteurX[i] - vecteurX[i - 1]) * (vecteurY[i] + vecteurY[i -
            1])/2
        i <- i + 1
    }
    total
}
# L'indice de Gini G vaut 2 fois l'aire A comprise entre la courbe et la
# première bissectrice. Pour calculer A, on calcule l'aire sous la courbe
# de Lorenz I. A + I vaut 1/2 car il s'agit au total de l'aire sous la
# première bissectrice.  Donc G = 2A = 2*(1/2 - I)
Gini <- 2 * (0.5 - calculIntegraleApprochee01(dGini$POP2008cum, dGini$SURFcum))
Gini

## [1] 0.4262
```

Traitements automatiques

On crée ensuite une fonction qui automatise la démarche vue précédemment (hormis la production du graphique, inutile pour le calcul) :

```
calculIndiceGini <- function(table) {
    dGini <- subset(table, select = c("CODGEO", "LIBELLE", "POP2008", "SURF"))
    dGini <- dGini[order(dGini$POP2008/dGini$SURF, decreasing = TRUE), ]
    dGini$POP2008cum <- cumsum(dGini$POP2008)/sum(dGini$POP2008)
    dGini$SURFcum <- cumsum(dGini$SURF)/sum(dGini$SURF)
    2 * (0.5 - calculIntegraleApprochee01(dGini$POP2008cum, dGini$SURFcum))
}
```

On peut ainsi retrouver la valeur précédente et faire le calcul pour chaque département :

```
calculIndiceGini(dPop36_08)

## [1] 0.4262

calculIndiceGini(dPop36_08[substr(dPop36_08$CODGEO, 1, 2) == "75", ])

## [1] 0.2621

calculIndiceGini(dPop36_08[substr(dPop36_08$CODGEO, 1, 2) == "92", ])

## [1] 0.3315

calculIndiceGini(dPop36_08[substr(dPop36_08$CODGEO, 1, 2) == "93", ])

## [1] 0.3063

calculIndiceGini(dPop36_08[substr(dPop36_08$CODGEO, 1, 2) == "94", ])

## [1] 0.3829
```

On en conclut que le Val-de-Marne possède une répartition particulièrement hétérogène de la densité par rapport aux autres départements centraux.

3.2 La famille des fonctions `*apply()`

Nous allons voir dans cette section comment appliquer des fonctions sur des sous groupes de données, et remplacer/limiter ainsi les boucles `loop` et `while` qui sont généralement plus lentes et gourmandes en ressources.

- `apply` - Appliquer une fonction aux lignes, ou aux colonnes d'un tableau de 1 a n dimensions

  `apply (data, dimensions d'application, nomdelafonction)`

```
# Matrice à deux dimensions
M <- matrix(runif(5), 4, 4)

# on applique la fonction min aux lignes
apply(M, 1, min)

## [1] 0.04304 0.04304 0.04304 0.04304


# on applique la fonction min aux colonnes
apply(M, 2, min)

## [1] 0.04304 0.04304 0.04304 0.04304


# Tableau à trois dimensions
M <- array(runif(5), dim = c(4, 4, 2))

apply(M, 1, sum)  #Le résultat est à une dimension
```

```
## [1] 2.959 3.631 4.387 4.413
```

```
apply(M, c(1, 2), sum)  #Le résultat est à deux dimensions (sum des lignes)
```

```
##        [,1]   [,2]   [,3]   [,4]
## [1,] 1.2114 0.3132 0.3397 1.0949
## [2,] 1.7672 1.2114 0.3132 0.3397
## [3,] 1.0949 1.7672 1.2114 0.3132
## [4,] 0.3397 1.0949 1.7672 1.2114
```

- `lapply()` - Appliquer une fonction sur chaque élément d'une liste, et retourne une liste en retour : `lapply(x, nomdelafonction)`
- `sapply` - Appliquer une fonction sur chaque élément d'une liste, et renvoie un vecteur
  `sapply(x, nomdelafonction)`
- `tapply` - Appliquer une fonction à des sous groupes de vector subdivisés à partir d'un vecteur contenant la liste des variables de découpe, en général un facteur.
  `tapply(Summary Variable, Group Variable, nomdelafonction)`

Pour qu'une utilisation de `tapply()` soit possible il faut être dans la situation suivante :

- Un *data.frame* à découper.
- On veut le découper en groupes à partir de variables
- Pour chaque groupe obtenu, on veut appliquer une fonction sur nos données

```
dData99_07$dep <- substr(dData99_07$CODGEO, 1, 2)

tapply(dData99_07$TXCHOM99, list(dData99_07$dep, 10 * round(dData99_07$POUV07/10)),
    mean)
```

```
##        0     10    20    30 40
## 75 9.125 12.917    NA    NA NA
## 92 7.000  8.667 14.80 20.00 NA
## 93    NA  9.500 14.75 19.87 23
## 94 8.000  8.545 12.43 16.67 NA
```

```
# Une liste avec nom :
a <- list(John = 3, Serge = 2)

# Une liste avec nom contenant des vecteurs :
a <- list(John = 3, Serge = c(2, 3))

# Les données sont accessible avec [], [[]], ou $nom
a[2]
```

```
## $Serge
## [1] 2 3
```

```
a[[2]]
```

```
## [1] 2 3
```

```
a["Serge"]

## $Serge
## [1] 2 3

a[["Serge"]]

## [1] 2 3

a$Serge

## [1] 2 3
```

Pour appliquer les fonctions `apply()` sur des lignes, il est impossible d'accéder avec la notation dollar $ aux données, seule la notation [ ] et [[ ]] fonctionne. Dans le cas de la fonction `lapply()`, il est absurde d'essayer d'accéder à d'autres colonnes du *data.frame*, sachant que `lapply()` renvoie seulement le contenu de la colonne en cours)

```
dat <- read.table(textConnection("X1 Y1 X2 Y2\n1 3.5 2.1 4.1 2.9\n2 3.1 1.2 0.8 4.3\n"))

# Pour changer le nom d'une colonne de data.frame, on accède à sa liste de
# noms, comme pour une liste normale.
b <- list(a = 2, b = 3)
names(b)[1] <- "x"
b

## $x
## [1] 2
##
## $b
## [1] 3



# Idem avec le data.frame
names(dat)[2] <- "Z"
dat

##    X1   Z  X2  Y2
## 1 3.5 2.1 4.1 2.9
## 2 3.1 1.2 0.8 4.3
```

- Il parait intéressant d'utiliser `apply()` avec un *data.frame*, car on peut le manipuler par ligne ou par colonne, mais attention à la nature des traitements à réaliser car en réalité, `apply()` transforme le *data.frame* en matrice pour procéder aux calculs.
- Préférer `lapply()` a `sapply()`, car `sapply()` procède à une simplification des données en sorties, pas toujours bienvenue.
- `lapply()` et `sapply()` appliquent une fonction sur chaque colonne, pas sur les lignes, donc inutile de chercher à accéder à des données contenues dans les autres colonnes si vous décidez de créer votre propre fonction. Par conséquent, il est aussi inutile d'utiliser le $ pour accéder à votre autre colonne, puisque `lapply()` et `sapply()` ne travaillent que sur une colonne à la fois.
- Si vous utilisez `apply()` sur un *data.frame*, pensez que la donnée qui vous est renvoyée est sous forme de vecteur. Il est impossible d'y accéder par la notation $ (qui ne marche que pour les listes), par contre [« nomdelacolonne »] et [« numerodelacolonne »] fonctionnent.

Utilisation du *package* `plyr` et `reshape2`

```
library(reshape2)
library(plyr)
```

`melt()` est une commande disponible dans le *package* `reshape2`, qui permet d'une part de faire la même chose que la commande reshape dans R base, et de passer d'un format « wide » à un format « long ». Toutefois, à la différence de la commande de base, `melt()` propose un nouveau format de données s'appuyant sur l'objet R *data.frame*. Cette commande permet ici de préparer les données en vue d'une utilisation avec d'autres commandes du *package* `plyr`.

```
# On spécifie d'une part les variables à conserver (id) et d'autre part la
# variable pivot (variable.name)
dMelted3608 <- melt(dPop36_08, id = c("CODGEO", "LIBELLE", "SURF"), variable.name = "Year")

# La colonne Year contient les identifiants des années
head(dMelted3608)
```

```
##   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
```

Syntaxe des fonctions `plyr`

La syntaxe des fonctions offertes pour le découpage et la manipulation de bloc de données par `plyr` est relativement simple :

Objet possible en entrée des fonctions :

- a = *array*
- d = *data.frame*
- l = *list*

Objet possible en sortie des fonctions :

- a = *array*
- d = *data.frame*
- l = *list*
- _ = vide

Donc :

- `ddply()` = *data.frame* en *input*, *data.frame* en *output*
- `ldply()` = *list* en *input*, *data.frame* en *output*

Ce qui fait au moins trois type de fonction, avec chacune une syntaxe légèrement différentes du fait de la nature des données :

```
a*ply(.data,.margins,.fun,...,.progress = "none")

d*ply(.data,.variables,.fun,...,.progress = "none")

l*ply(.data,.fun,...,.progress = "none")
```

La description des arguments est la suivantes :

- `.data` = données en entrées
- `.margins` or `.variables` = comment découper les données
- `.fun` = la fonction à appliquer sur chacun des blocs découpés

Toujours à partir du même exemple de sortie de simulation, on applique ici la fonction `ddply()` qui prend en entrée et en sortie un *data.frame* :

Remarques importantes :

1. `ddply()` peut fonctionner en ligne ou en colonne, car il renvoie des morceaux de *data.frame* issu du découpage !
2. `ddply()` possède plusieurs modes de fonctionnements : on en verra au moins trois :
   - le mode `transform` qui permet d'ajouter des colonnes supplémentaires en se basant sur une formule ou une fonction specifique (nomdelanouvellecolonne = formule et/ou fonction)
   - le mode `summarise` qui permet de travailler sur des agrégats basé sur les données renvoyés par le découpage du *data.frame*, marche aussi avec l'appel de fonction
   - le mode `subset` qui permet de selectionner une ligne en fonction d'une condition
   - le mode `fonction` qui permet d'appliquer n'importe quel fonction
3. `ddply()` + `transform` fonctionne sur des lignes, mais vous pouvez forcer son fonctionnement sur des colonnes en utilisant des fonctions de type sum, mean,etc qui prendront en paramètre le nom de la colonne
4. `ddply()` + `summarise` fonctionne sur l'agrégation des colonnes, et il en résulte un nouveau *data.frame*
5. `ddply() + fonction` fonctionne sur ce que vous voulez, ligne ou colonne, tout dépend si vous appelez dans votre fonction des fonctions qui utilisent des vecteurs colonne (ex `colwise()`, `sum()`, `max()`, `mean()`, etc ...) ou pas.
6. Il est possible de filtrer le *data.frame* avant application de la fonction, mais par contre, dans le cas où la variable de découpe est exclue par votre filtre, vous devez l'indiquer avec nomdudataframe$variable àdécouper

Utilisation conjointe avec la fonction `subset()`

```
# minimum of population
dComputeMin <- ddply(dMelted3608, .(Year), subset, value == min(value))
```

Utilisation conjointe avec la fonction `transform()`

Ajoute une/des colonnes supplémentaires aux données originales, après application d'une ou de plusieurs fonctions basées sur un parcours de la fonction en ligne (ce qui n'empeche pas d'appeler des fonctions sur les colonnes, voir ci-dessous).

```
# Calcul de l'écart de population à la population minimum par ville sur
# l'ensemble des temps pour une zone géographique
dComputeEcartPop <- ddply(dMelted3608, .(CODGEO), transform, ecart_pop_min = value -
    min(value))
```

Ou bien calculer des rangs pour la colonne population :

```
dRankByTime <- ddply(dMelted3608, .(CODGEO), transform, rank = rank(value))
head(dRankByTime[order(dRankByTime$CODGEO, dRankByTime$rank), ])

##   CODGEO                  LIBELLE SURF    Year value rank
## 8  75101 Paris 1er Arrondissement  183 POP1999 16888    1
## 9  75101 Paris 1er Arrondissement  183 POP2008 17440    2
## 7  75101 Paris 1er Arrondissement  183 POP1990 18360    3
## 6  75101 Paris 1er Arrondissement  183 POP1982 18509    4
## 5  75101 Paris 1er Arrondissement  183 POP1975 22793    5
## 4  75101 Paris 1er Arrondissement  183 POP1968 32332    6
```

Ou bien appliquer une fonction spécifiquement sur des colonnes, récopié sur toutes les lignes de chaque *data.frame* résultat de la découpe en fonction de `CODGEO` :

```
head(ddply(dMelted3608, .(CODGEO), transform, sommeRef = mean(value)))

##   CODGEO                  LIBELLE SURF    Year value sommeRef
## 1  75101 Paris 1er Arrondissement  183 POP1936 37062    26362
## 2  75101 Paris 1er Arrondissement  183 POP1954 37330    26362
## 3  75101 Paris 1er Arrondissement  183 POP1962 36543    26362
## 4  75101 Paris 1er Arrondissement  183 POP1968 32332    26362
## 5  75101 Paris 1er Arrondissement  183 POP1975 22793    26362
## 6  75101 Paris 1er Arrondissement  183 POP1982 18509    26362
```

Utilisation conjointe avec la fonction `summarise()`

```
head(ddply(dMelted3608, .(CODGEO), summarise, mean = mean(value), sd = sd(value)))

##   CODGEO  mean    sd
## 1  75101 26362  9232
## 2  75102 29895  9798
## 3  75103 47612 13657
## 4  75104 45175 14962
## 5  75105 76367 17215
## 6  75106 61879 16566
```

Utilisation conjointe avec une de vos fonctions personnelles

Celle ci peut utiliser des opérations en ligne ou en colonnes, à votre convenance :

```
mafonction <- function(data,arguments) {...}
ddply(df, .(Ville), function(data) {mafonction(data,arguments)})
```

Exemple de cette fonction un peu complexe qui ajoute la population sommée comme population de référence pour chaque ligne de notre tableau

```
head(ddply(dMelted3608, .(CODGEO), function(x) {
    t <- colwise(sum)(x[c("value")])
    x$sommeRef <- t$value
    return(x)
}))

##   CODGEO                  LIBELLE SURF    Year value sommeRef
## 1  75101 Paris 1er Arrondissement  183 POP1936 37062   237257
## 2  75101 Paris 1er Arrondissement  183 POP1954 37330   237257
## 3  75101 Paris 1er Arrondissement  183 POP1962 36543   237257
## 4  75101 Paris 1er Arrondissement  183 POP1968 32332   237257
## 5  75101 Paris 1er Arrondissement  183 POP1975 22793   237257
## 6  75101 Paris 1er Arrondissement  183 POP1982 18509   237257
```

Que l'on aurait pu évidemment écrire sous cette forme là également, beaucoup plus simple (voir `transform()`) :

```
head(ddply(dMelted3608, .(CODGEO), transform, sommeRef = sum(value)))

##   CODGEO                  LIBELLE SURF    Year value sommeRef
## 1  75101 Paris 1er Arrondissement  183 POP1936 37062   237257
## 2  75101 Paris 1er Arrondissement  183 POP1954 37330   237257
## 3  75101 Paris 1er Arrondissement  183 POP1962 36543   237257
## 4  75101 Paris 1er Arrondissement  183 POP1968 32332   237257
## 5  75101 Paris 1er Arrondissement  183 POP1975 22793   237257
## 6  75101 Paris 1er Arrondissement  183 POP1982 18509   237257
```

Utilisation conjointe avec « colwise »

Un raccourci existe également pour appliquer une fonction sur chacune des colonnes (sauf certaines, par exemple `Temps` ...)

```
head(ddply(dData99_07, .(dep), colwise(sum, .(EMPLOI99))))

##   dep EMPLOI99
## 1  75  1656036
## 2  92   810226
## 3  93   467702
## 4  94   463520
```

Ou encore sur plusieurs colonnes, cette fois-ci filtrées avant application de la somme :

```
head(ddply(dData99_07[, 6:7], .(dData99_07$dep), colwise(sum)))

##   dData99_07$dep EMPLOI99 ACTOCC99
## 1             75  1656036   991003
## 2             92   810226   651004
## 3             93   467702   556628
## 4             94   463520   542775
```

3.3 Code

```r
# Importation des données
dData99_07 <- read.csv("data/data99_07.csv", sep=";")
dPop36_08 <- read.csv("data/pop36_08.csv", sep=";")

### GENERALITES SUR LES FONCTIONS

# Fonction carré
Carre <- function (x) {
  return(x^2)
}

Carre(8)

# qui peut s'écrire également sous cette forme
Carre <- function (x) (x^2)
Carre(8)

# Autres fonctions
curve(dnorm, -10,10, main = "Distribution normale")
curve(exp(x)/ (1 + exp(x)), -10, 10, main = "Courbe logistique")
D(expression(a * x^b),"x")


### EQUILIBRE DE WARDROP

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

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)

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

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

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

```r
### EXEMPLES DE FONCTIONS SIMPLES

s <- function (x,y) { x + y }
s(Carre(3),Carre(4))

trigo <- function (x) {c(cos(x), cos(x + pi / 2), cos(x + pi), cos(x + 3 * pi / 4))}
trigo(pi / 6)

transposer <- function (L) {L + length(L)}
transposer(1:10)


### DEUX EXEMPLES DE FONCTIONS AVANCEES

# Suite de Syracuse

syracuse <- function(nombre)  {
  L <- NULL
  L[1] <- nombre
  i <- 1
  # Test de parité
  while (L[i] != 1) {
    if (round(L[i] / 2) == L[i] / 2) {
      L[i + 1] <- L[i] / 2
      }
    else {
      L[i + 1] <- 3 * L[i] + 1
    }
    i <- i + 1
  }
  L
}

dureeVolSyracuse <- function(x) {
length(syracuse(x)) # La taille de la liste correspond à la durée du vol
}


dureeVolSyracuseListe <- function(L) {
 M <- NULL
 j <- 1
 while (j<=length(L)) {
   M[j] <- dureeVolSyracuse(L[j])
   j <- j + 1
 }
 M
}

lVolsSyracuseParis <- dureeVolSyracuseListe(dPop36_08$POP1999)
plot( dPop36_08$POP1999, lVolsSyracuseParis,
      log = "xy",
      xlab = "Population communale",
      ylab = "Durée de vol de Syracuse")
max(lVolsSyracuseParis)

dPop36_08[order(lVolsSyracuseParis, decreasing=TRUE)[1],]

plot(syracuse(dPop36_08[order(lVolsSyracuseParis, decreasing=TRUE)[1],10]),
```

```
     type = "l",
     log = "y",
     xlab = "n",
     ylab = "u_n")


# Courbe de Lorenz et indice de Gini

dGini <- subset(dPop36_08, select = c("CODGEO","LIBELLE","POP2008","SURF"))
dGini <- dGini[order(dGini$POP2008 / dGini$SURF, decreasing = TRUE),]
dGini$POP2008cum <- cumsum(dGini$POP2008) / sum(dGini$POP2008)
dGini$SURFcum <- cumsum(dGini$SURF) / sum(dGini$SURF)
plot(dGini$POP2008cum,
     dGini$SURFcum,
     type = "l",
     xlab = "Population cumulée",
     ylab = "Superficie cumulée")

# Affichage de la courbe correspondant à une égalité parfaite

lines(1:100 /100, 1:100 / 100, col = "red")

calculIntegraleApprochee01 <- function (vecteurX, vecteurY) {
total <- vecteurY[1] / 2 * vecteurX[1]
i <- 2
  while (i <= length(vecteurX)) {
    total <- total + (vecteurX[i] - vecteurX[i - 1]) * (vecteurY[i] + vecteurY[i - 1]) / 2
    i <- i + 1
  }
  total
}

Gini <- 2 * (0.5 - calculIntegraleApprochee01(dGini$POP2008cum, dGini$SURFcum))
Gini

calculIndiceGini <- function(table) {
dGini <- subset(table, select = c("CODGEO","LIBELLE","POP2008","SURF"))
dGini <- dGini[order(dGini$POP2008 / dGini$SURF, decreasing = TRUE),]
dGini$POP2008cum <- cumsum(dGini$POP2008) / sum(dGini$POP2008)
dGini$SURFcum <- cumsum(dGini$SURF) / sum(dGini$SURF)
2*(0.5 - calculIntegraleApprochee01(dGini$POP2008cum,dGini$SURFcum))
}

calculIndiceGini(dPop36_08)
calculIndiceGini(dPop36_08[substr(dPop36_08$CODGEO,1,2) == '75',])
calculIndiceGini(dPop36_08[substr(dPop36_08$CODGEO,1,2) == '92',])
calculIndiceGini(dPop36_08[substr(dPop36_08$CODGEO,1,2) == '93',])
calculIndiceGini(dPop36_08[substr(dPop36_08$CODGEO,1,2) == '94',])

# fonction apply()

M <- matrix(runif(5),4,4)
apply(M, 1, min)
apply(M, 2, min)
M <- array(runif(5), dim = c(4, 4, 2))
apply(M, 1, sum)
apply(M, c(1, 2), sum)
```

```r
# fonction tapply()

dData99_07$dep <- substr(dData99_07$CODGEO,1,2)
tapply(dData99_07$TXCHOM99, list(dData99_07$dep, 10*round(dData99_07$POUV07/10)), mean)

# Accéder aux objets et à leurs attributs

a <- list (John = 3, Serge = 2)

# Une liste avec nom contenant des vecteurs
a <- list (John = 3, Serge  = c(2,3))

# Les données sont accessible avec [], [[]], ou $nom
a[2]
a[[2]]
a['Serge']
a[['Serge']]
a$Serge

dat <- read.table(textConnection("X1 Y1 X2 Y2
1 3.5 2.1 4.1 2.9
2 3.1 1.2 0.8 4.3
"))

# Pour changer le nom d'une colonne de data.frame
b <- list (a = 2, b = 3)
names(b)[1] <- "x"
b

# Idem avec le data.frame
names(dat)[2] <- "Z"
dat

### PACKAGES RESHAPE2 et PLYR

# Chargement des packages

library(reshape2)
library(plyr)

dMelted3608 <- melt(dPop36_08,
                    id=c("CODGEO", "LIBELLE", "SURF"),
                    variable.name="Year")

head(dMelted3608)

# Utilisation conjointe avec la fonction subset()

dComputeMin <- ddply(dMelted3608, .(Year), subset, value == min(value))

# Utilisation conjointe avec la fonction transform()

dComputeEcartPop <- ddply(dMelted3608, .(CODGEO), transform, ecart_pop_min = value -
min(value))
dRankByTime <- ddply(dMelted3608, .(CODGEO), transform, rank = rank(value))
head(dRankByTime[order(dRankByTime$CODGEO, dRankByTime$rank), ])
head(ddply(dMelted3608, .(CODGEO), transform, sommeRef = mean(value)))
```

```r
# Utilisation avec summarize

head(ddply(dMelted3608, .(CODGEO), summarise, mean = mean(value), sd = sd(value)))

# Application par lignes, option 1

head(ddply(dMelted3608, .(CODGEO), function(x) {
t <- colwise(sum)(x[c("value")])
x$sommeRef <- t$value
return(x)
}))

# option 2
head(ddply(dMelted3608, .(CODGEO), transform, sommeRef = sum(value)))

# Application par colonnes

head(ddply(dData99_07, .(dep), colwise(sum, .(EMPLOI99))))
head(ddply(dData99_07[, 6:7], .(dData99_07$dep), colwise(sum)))
```

Chapitre 4

Analyse univariée

Objectifs

Ce chapitre vise à présenter des manipulations simples nécessaires pour calculer des taux et réaliser des recodages, des analyses univariées et les représentations graphiques associées.

Prérequis

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

Packages nécessaires

La grande majorité des fonctions utilisées font partie des *packages* de base installés et chargés par défaut : le *package* `base` et le *package* `stats`. Il y a donc peu d'installation supplémentaire à réaliser avant de commencer. Deux *packages* supplémentaires sont nécessaires, le premier est `Hmisc` (**H**arrell **Misc**ellaneous) développé par Frank E. Harrell permettant de calculer des mesures pondérées. Le second est `classInt` développé par Roger Bivand, qui contient un ensemble de fonctions utiles pour la discrétisation de variables continues.

Données nécessaires

Deux tableaux de données produites par l'INSEE :

- Feuille `data99_07` : données de référence pour 1999 et 2007 avec différentes variables sur la composition socio-professionnelle des communes de Paris et la petite couronne.
- Feuille `pop36_08` : données de population sans double compte des recensements de la population de 1936 à 2008 pour Paris et la petite couronne.

Pour une description plus précise du contenu des fichiers, voir la Section 1.6.

Préparation des données

L'importation des données est détaillée dans le Chapitre 2. Une fois les données importées, on contrôle leur contenu soit en cliquant sur le tableau qui apparait dans la fenêtre « Espace de travail », soit avec les fonctions présentées dans le Chapitre 2. Si on travaille avec un encodage UTF8, on remarquera que le champ `LIGBEO` du tableau `dPop3608` pose problème : les noms de communes avec des accents sont mal importés. En effet, l'encodage du fichier d'origine est « latin1 » (ISO 8859-1), il faut alors préciser l'encodage dans les options d'importation :

```
dRef9907 <- read.table("data/data99_07.csv",
                       sep=";",
                       dec=",",
                       quote = "\"",
                       header = TRUE,
                       encoding = "latin1"
                       )

dPop3608 <- read.table("data/pop36_08.csv",
                       sep=";",
                       dec=",",
                       header = TRUE,
                       quote = "\"",
                       encoding = "latin1"
                       )
```

Comme préalable aux manipulations présentées ce chapitre, plusieurs nouveaux champs sont calculés. Ces calculs simples et les recodages associés sont regroupés dans la section suivante.

4.1 Calculs simples et recodages

Les variables suivantes sont créées : le taux de variation total et le taux de variation moyen entre 1936 et 2008, la densité de population en 2008 ainsi que le taux d'emplois résidents en 2006 (nombre d'emplois localisés dans une commune divisé par le nombre d'actifs résidant dans cette commune) au niveau des communes et des arrondissements parisiens. Trois nouvelles variables sont créées dans le tableau `dPop3608` et une nouvelle variable dans le tableau `dRef9907` :

```
dPop3608$TXCVARTOT <- ((dPop3608$POP2008 / dPop3608$POP1936) - 1) * 100
dPop3608$TXCVARMOY <- (((dPop3608$POP2008 / dPop3608$POP1936) ^
                              (1 / (2008 - 1936))) - 1) * 100
dPop3608$DENSITE <- dPop3608$POP2008 / dPop3608$SURF
dRef9907$TXEMPLOI <- dRef9907$EMPLOI06 / dRef9907$ACTOCC06
```

Pour ce genre d'instructions, la fonction `with()` (présentée au Chapitre 2) est utile pour éviter la répétition du *data.frame* de référence. Par exemple, pour le calcul du taux d'emplois résidents :

```
dRef9907$TXEMPLOI <- with(dRef9907, 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 : une variable « Département » (`CODDEP`), et une variable « Distance à Paris » (`DISTCONTINUE`). La variable « Département » est une extraction des deux premiers chiffres du code communal (`CODGEO`) :

```
dRef9907$CODDEP <- substr(dRef9907$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 1^{r} 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. Pour référencer les coordonnées géographiques du 1^{r} arrondissement soit on stocke ces coordonnées dans un vecteur à part (Option 1), soit on fait une sélection à l'intérieur de la formule de calcul de la distance, ce qui peut être réalisé de deux façons (Options 2 et 2bis) :

```
# Option 1
vCoordPremierArr <- as.numeric(dRef9907[1, 3:4])
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - vCoordPremierArr[1])^2 + (dRef9907$Y -
```

```
    vCoordPremierArr[2])^2)/10

# Option 2
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - dRef9907$X[dRef9907$CODGEO == 75101])^2 +
    (dRef9907$Y - dRef9907$Y[dRef9907$CODGEO == 75101])^2)/10

# Option 2 bis
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - dRef9907[1, 3])^2 + (dRef9907$Y -
    dRef9907[1, 4])^2)/10
```

Là encore la fonction `with()` est utile pour éviter les répétitions du *data.frame* de référence :

```
dRef9907$DISTCONTINUE <- with(dRef9907, sqrt((X - X[CODGEO == 75101])^2 + (Y -
    Y[CODGEO == 75101])^2)/10)
```

Dans la seconde option, l'expression `dRef9907$X[dRef9907$CODGEO == 75101]` désigne la valeur de la variable `X` du tableau `dRef9907` quand la valeur de la variable `CODGEO` est égale à 75101, c'est-à-dire plus simplement la coordonnée `X` du premier arrondissement de Paris. Maintenant il ne reste plus qu'à discrétiser la variable `DISTCONTINUE` dans une nouvelle variable `DISTCLASS`. Deux options sont proposées, dans le premier cas, on crée des classes au cas par cas, puis on construit un facteur en étiquetant les valeurs. Dans le second cas, le facteur est créé en une seule et même manipulation avec la fonction `cut()` qui découpe une variable continue selon des seuils (`breaks =`) :

```
# Option 1
dRef9907$DISTCLASS[dRef9907$DISTCONTINUE < 5] <- 1
dRef9907$DISTCLASS[dRef9907$DISTCONTINUE >= 5] <- 2
dRef9907$DISTCLASS[dRef9907$DISTCONTINUE >= 10] <- 3
dRef9907$DISTCLASS[dRef9907$DISTCONTINUE >= 15] <- 4

dRef9907$DISTCLASS <- factor(dRef9907$DISTCLASS,
                             levels = c(1, 2, 3, 4),
                             labels = c("[0,5[", "[5,10[", "[10,15[", "[15,24]"))

# Option 2
vSeuilsDist <- c(0,5,10,15, max(dRef9907$DISTCONTINUE))

dRef9907$DISTCLASS <- cut(dRef9907$DISTCONTINUE,
                          breaks = vSeuilsDist,
                          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.

```
min(dPop3608$TXCVARTOT)

## [1] -55.27

max(dPop3608$TXCVARTOT)

## [1] 2278

mean(dPop3608$TXCVARTOT)
```

```
## [1] 185.1
```

```
median(dPop3608$TXCVARTOT)
```

```
## [1] 68.18
```

```
sd(dPop3608$TXCVARTOT)
```

```
## [1] 315
```

```
summary(dPop3608$TXCVARTOT)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   -55.3    19.3    68.2   185.0   219.0  2280.0
```

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

Ces fonctions sont utiles, par exemple, pour calculer la proportion moyenne de cadres sur l'espace d'étude en 1999. Les données (`dRef9907`) comportent une variable d'effectif de la population active occupée (`ACTOCC99`) et une variable qui est la proportion des cadres par commune (`PCAD99`). Une première solution serait de calculer les effectifs des cadres en multipliant les deux variables puis de calculer le rapport entre la somme des cadres et la somme des actifs occupés. 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 :

```
# Option 1
dRef9907$NCAD99 <- 0.01 * dRef9907$PCAD99 * dRef9907$ACTOCC99
100 * sum(dRef9907$NCAD99)/sum(dRef9907$ACTOCC99)
```

```
## [1] 26.45
```

```
# Option 2
wtd.mean(dRef9907$PCAD99, weights = dRef9907$ACTOCC99)
```

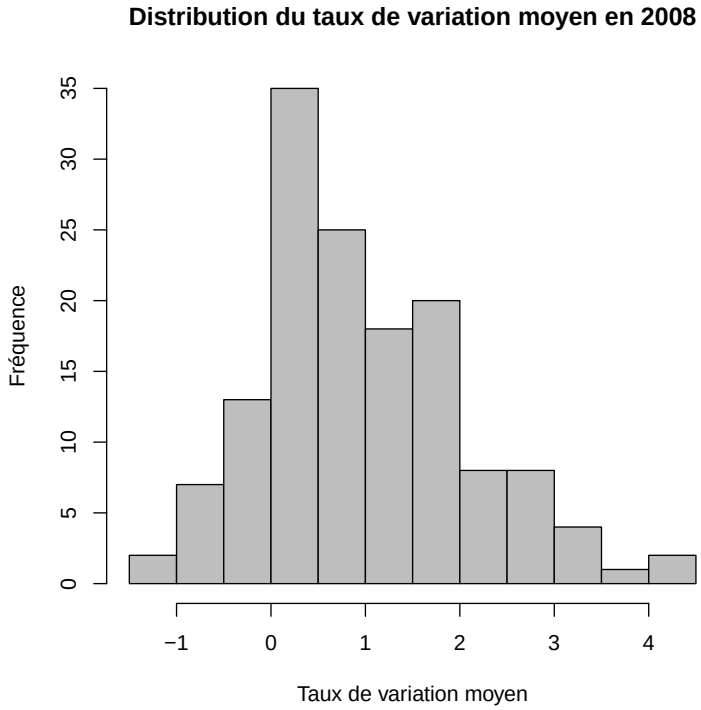
```
## [1] 26.45
```

La fonction `wtd.table()` du *package* `Hmisc` permet également de réaliser des tableaux de contingence pondérés.

4.3 Représentations graphiques des distributions statistiques

On peut ensuite utiliser les fonctions graphiques pour représenter la distribution de la variable. Il existe plusieurs *packages* spécialisés dans les représentations graphiques, mais ici sont utilisées uniquement les fonctions du *package* `graphics`, installé et chargé par défaut. Leur nom est en général explicite quant au type de graphique créé : `plot()`, `hist()`, `barplot()`, `boxplot()`, etc.

```
hist(dPop3608$TXCVARMOY,
     main = "Distribution du taux de variation moyen en 2008",
     xlab = "Taux de variation moyen",
     ylab = "Fréquence",
     col = "grey")
```



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. Chapitre 2). 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()`).

```
vSomPop3608 <- apply(dPop3608[, 3:11], 2, sum)
vSomPop3608

## POP1936 POP1954 POP1962 POP1968 POP1975 POP1982 POP1990 POP1999 POP2008
## 5310964 5580874 6230582 6423315 6276600 6081238 6140816 6164238 6578258

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

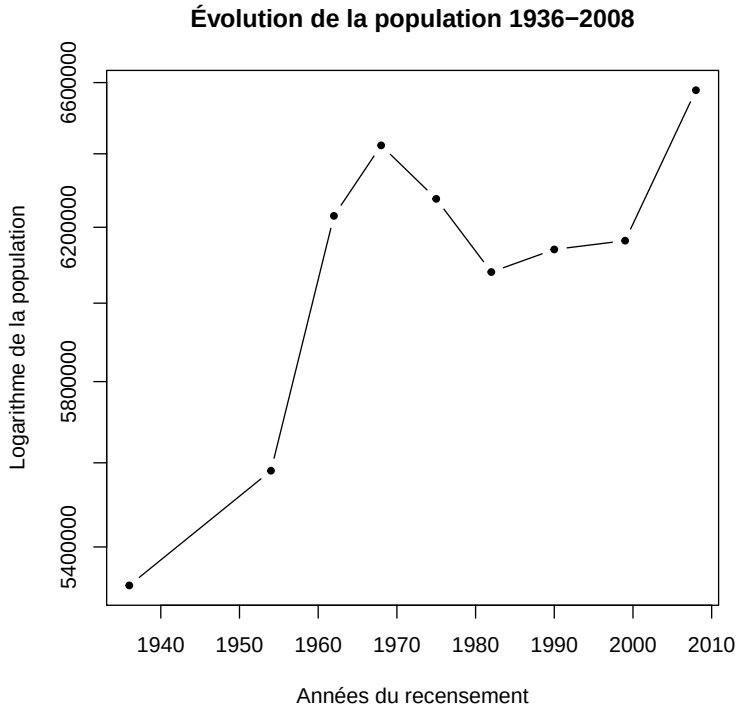
Concernant les dates des recensements, on peut soit créer un vecteur en spécifiant chaque valeur « à la main », soit extraire les dates des noms de colonnes de l'objet créé précédemment (`vSomPop3608`). Noter ici l'emboîtement de trois fonctions, `names()`, `substr()` et `as.integer()` : prendre les noms dans l'objet `vSomPop3608`, en extraire les caractères 4 à 7 et les transformer en entiers :

```
vDates <- as.integer(substr(names(vSomPop3608), 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 =`, *point character*). On peut obtenir la liste des symboles ponctuels dans l'aide (`?pch`) :

```
plot(vSomPop3608 ~ vDates,
     type = "b",
     log = "y",
     pch = 20,
```

```
     main = "Évolution de la population 1936-2008",
     xlab = "Années du recensement",
     ylab = "Logarithme de la population")
```

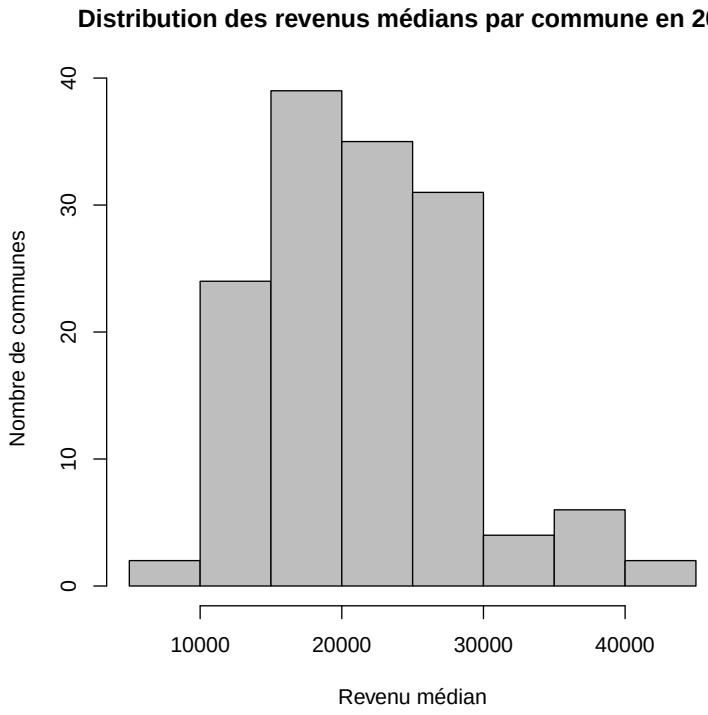


On cherche maintenant à avoir des résumés de l'ensemble des variables renseignant sur la structure socio-économique des communes en 1999 et en 2007 (de la variables `P20ANS` à la variable `RFUCQ2`). Cette requête est immédiate grâce à la fonction `summary()` appliquée à un ensemble de colonnes par la fonction `apply()`. Le résultat est un *data.frame* avec les différents résumés statistiques en ligne et les variables en colonnes. On transpose ce tableau pour obtenir un résultat plus lisible avec la fonction `t()` :

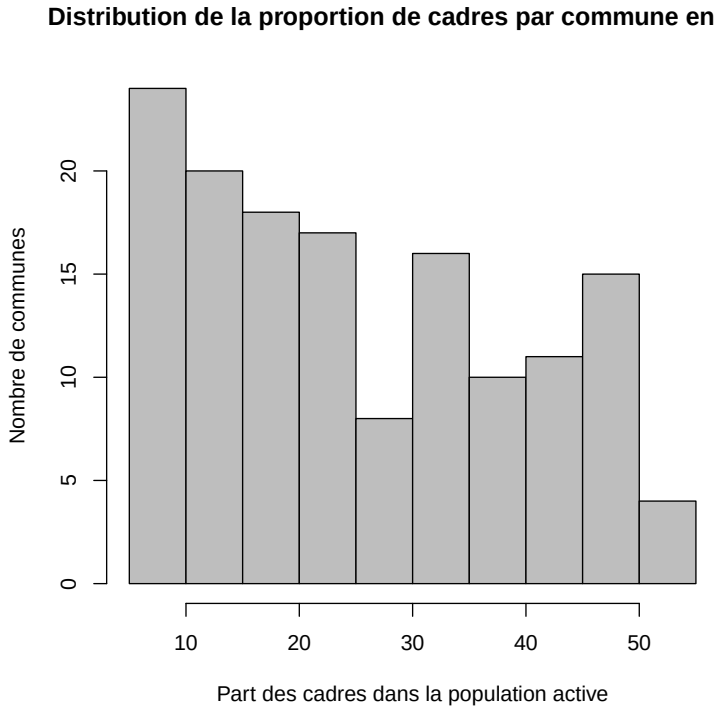
```
dResumes99 <- apply(dRef9907[, 8:20], 2, summary)
dResumes07 <- apply(dRef9907[, 23:35], 2, summary)

t(dResumes07)

##            Min. 1st Qu. Median     Mean 3rd Qu.  Max.
## P20ANS07     16    23.5     26    25.60    28.0    37
## PNDIP07       6    11.0     16    18.80    25.5    46
## TXCHOMA07     6     8.0     10    11.50    14.0    23
## INTA007       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
## PMON007       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 des variables de revenus médians en 2007 et de la part de cadres en 2007 :

```
hist(dRef9907$RFUCQ207,
     main = "Distribution des revenus médians par commune en 2007",
     xlab = "Revenu médian",
     ylab = "Nombre de communes",
     col = "grey")
```

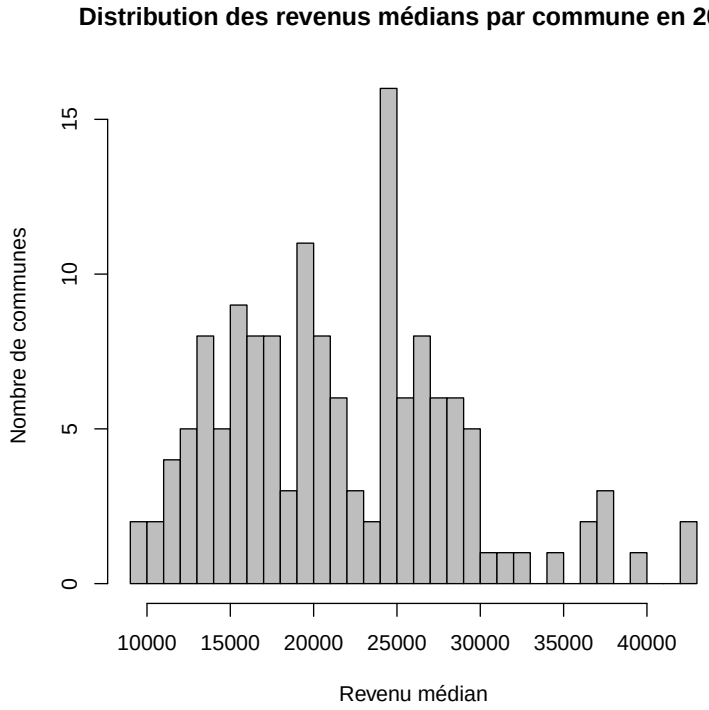



```
hist(dRef9907$PCAD07,
     main = "Distribution de la proportion de cadres par commune en 2007",
     xlab = "Part des cadres dans la population active",
     ylab = "Nombre de communes",
     col = "grey")
```



A partir de l'histogramme, on recode la variable de revenus médians en 2007 en 3 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 21 000 € et un autre vers 31 000 €.

```
hist(dRef9907$RFUCQ207,
     main = "Distribution des revenus médians par commune en 2007",
     breaks = 35,
     xlab = "Revenu médian",
     ylab = "Nombre de communes",
     col = "grey")
```



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

```
vSeuilsRevMedian <- c(0,21000,31000, max(dRef9907$RFUCQ207))

dRef9907$RFUCQ207CLASS <- cut(dRef9907$RFUCQ207,
                              breaks = vSeuilsRevMedian,
                              include.lowest = TRUE,
                              labels = c("[9000, 20000[",
                                         "[20000, 30000[",
                                         "[30000, 40000["))
```

A noter qu'il existe un *package* spécialisé dans la discrétisation nommé `classInt`. Il comprend des fonctions très utiles pour faire des cartes choroplèthes et ainsi retrouver les modes de discrétisation des logiciels tels que ArcGis ou MapInfo. 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 contenues dans un attributs `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.

```
lJenks <- classIntervals(var=dRef9907$RFUCQ207,
                         n=3,
                         style="jenks")

vSeuilsJenks <- lJenks$brks

dRef9907$RFUCQ207CLASSJENKS <- cut(dRef9907$RFUCQ207,
                                   breaks = vSeuilsJenks,
                                   include.lowest = TRUE)
```

Ces résumés statistiques, ces représentations graphiques et ces nouvelles variables calculées ou recodées permettent maintenant d'aborder les analyses bivariées.

4.4 Code

```
# Importation des données
dRef9907 <- read.table("data/data99_07.csv",
                       sep=";",
                       dec=",",
                       quote = "\"",
                       header = TRUE,
                       encoding = "latin1"
                       )

dPop3608 <- read.table("data/pop36_08.csv",
                       sep=";",
                       dec=",",
                       header = TRUE,
                       quote = "\"",
                       encoding = "latin1"
                       )



# Calcul des taux de variation, du taux d'emploi et de la densité
dPop3608$TXCVARTOT <- ((dPop3608$POP2008 / dPop3608$POP1936) - 1) * 100
dPop3608$TXCVARMOY <- (((dPop3608$POP2008 / dPop3608$POP1936) ^
                              (1 / (2008 - 1936))) - 1) * 100
dPop3608$DENSITE <- dPop3608$POP2008 / dPop3608$SURF
dRef9907$TXEMPLOI <- dRef9907$EMPLOI06 / dRef9907$ACTOCC06

# Création d'une variable département
dRef9907$CODDEP <- substr(dRef9907$CODGEO, 1, 2)

# Création d'une variable distance à Paris

# Option 1
vCoordPremierArr <- as.numeric(dRef9907[1, 3:4])
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - vCoordPremierArr[1]) ** 2 +
                                (dRef9907$Y - vCoordPremierArr[2]) ** 2) / 10

# Option 2
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - dRef9907$X[dRef9907$CODGEO == 75101]) ** 2 +
                                (dRef9907$Y - dRef9907$Y[dRef9907$CODGEO == 75101]) ** 2)
                         / 10

# Option 2 bis
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - dRef9907[1, 3]) ** 2 +
                                (dRef9907$Y - dRef9907[1, 4]) ** 2) / 10

# Discrétisation de la variable distance

# Option 1
dRef9907$DISTCLASS[dRef9907$DISTCONTINUE < 5] <- 1
dRef9907$DISTCLASS[dRef9907$DISTCONTINUE >= 5] <- 2
dRef9907$DISTCLASS[dRef9907$DISTCONTINUE >= 10] <- 3
dRef9907$DISTCLASS[dRef9907$DISTCONTINUE >= 15] <- 4

dRef9907$DISTCLASS <- factor(dRef9907$DISTCLASS,
                             levels = c(1, 2, 3, 4),
                             labels = c("[0,5[", "[5,10[",
                                        "[10,15[", "[15,24]"))
```

```
# Option 2
vSeuilsDist <- c(0,5,10,15, max(dRef9907$DISTCONTINUE))

dRef9907$DISTCLASS <- cut(dRef9907$DISTCONTINUE,
                          breaks = vSeuilsDist,
                          include.lowest = TRUE)

# Résumés statistiques
min(dPop3608$TXCVARMOY)
max(dPop3608$TXCVARMOY)
mean(dPop3608$TXCVARMOY)
median(dPop3608$TXCVARMOY)
sd(dPop3608$TXCVARMOY)
summary(dPop3608$TXCVARMOY)

# Calcul d'une moyenne pondérée

# Option 1
dRef9907$NCAD99 <- 0.01 * dRef9907$PCAD99 * dRef9907$ACTOCC99
100 * sum(dRef9907$NCAD99) / sum(dRef9907$ACTOCC99)

# Option 2
wtd.mean(dRef9907$PCAD99, weights = dRef9907$ACTOCC99)


# Représentations graphiques : histogramme
hist(dPop3608$TXCVARMOY,
     main = "Distribution du taux de variation moyen en 2008",
     xlab = "Taux de variation moyen",
     ylab = "Fréquence",
     col = "grey")

# Calcul de l'évolution de la population

vSomPop3608 <- apply(dPop3608[,3:11], 2, sum)
vSomPop3608
vDates <- c(1936, 1954, 1962, 1968, 1975, 1982, 1990, 1999, 2008)

# Représentation graphique de l'évolution de la population

plot(vSomPop3608 ~ vDates,
     type = "b",
     log = "y",
     main = "Évolution de la population 1936-2008",
     xlab = "Années du recensement",
     ylab = "Logarithme de la population")


# Résumés statistiques d'un ensemble de variables

dResumes99 <- apply(dRef9907[ ,8:20], 2, summary)
dResumes07 <- apply(dRef9907[ ,23:35], 2, summary)
t(dResumes99)
t(dResumes07)

# Représentations graphiques : histogrammes
hist(dRef9907$RFUCQ207,
     main = "Distribution des revenus médians par commune en 2007",
```

```
     xlab = "Revenu médian",
     ylab = "Nombre de communes",
     breaks = 35,
     col = "grey")

hist(dRef9907$PCAD07,
     main = "Distribution de la proportion de cadres par commune en 2007",
     xlab = "Part des cadres dans la population active",
     ylab = "Nombre de communes",
     col = "grey")

# Discrétisation manuelle de la variable de revenu
vSeuilsRevMedian <- c(0,21000,31000, max(dRef9907$RFUCQ207))
dRef9907$RFUCQ207CLASS <- cut(dRef9907$RFUCQ207,
                              breaks = vSeuilsRevMedian,
                              include.lowest = TRUE,
                              labels = c("[9000, 20000[",
                                         "[20000, 30000[",
                                         "[30000, 40000["))

# Discrétisation de la variable de revenu avec l'aglorithme de Jenks
lJenks <- classIntervals(var=dRef9907$RFUCQ207,
                         n=3,
                         style="jenks")

vSeuilsJenks <- lJenks$brks

dRef9907$RFUCQ207CLASSJENKS <- cut(dRef9907$RFUCQ207,
                                   breaks = vSeuilsJenks,
                                   include.lowest = TRUE)
```

Chapitre 5

Analyses bivariées

Objectif

L'objectif de ce chapitre est de proposer un ensemble de méthodes issues de la statistique bivariée appliquées à l'étude de problèmes géographiques dans R. Les types de représentation et les méthodes d'analyse des relations entre deux variables diffèrent selon le type des variables. Nous en donnons ici trois exemples : (1) une étude de la relation entre deux variables quantitatives (densité d'habitants au kilomètre carré en 2008 et distance euclidienne au centre de Paris), (2) une étude de la relation entre une variable quantitative et une variable qualitative (revenu par habitant en 2007 et classes de distance à Paris), et (3) une étude de la relation entre deux variables qualitatives (classes de revenu par habitant en 2007 et appartenance départementale).

Prérequis

Corrélation et régression linéaire, analyse de la variance, test du χ^2.

Packages R nécessaires

Les *packages* de base installés et chargés par défaut suffisent pour les analyses proposées ici. Le *package* `lattice` développé par Deepayan Sarkar, est cependant très utile pour représenter des relations bi- ou multivariées comme la répartition d'une distribution au sein des différentes modalités d'un facteur.

Données nécessaires

Dans ce chapitre nous utilisons les deux tableaux suivants :

- `data99_07.csv` : données de référence pour 1999 et 2007 avec différentes variables sur la composition socio-professionnelle des communes de Paris et la petite couronne.
- `pop36_08.csv` : données de population sans double compte des recensements de la population de 1936 à 2008 pour Paris et la petite couronne.

Pour une description plus précise du contenu des fichiers, voir la Section 1.6.

Préparation des données :

Les deux tableaux de données sont en format texte (.csv), le séparateur de champs `(sep=)` est le point-virgule, le séparateur décimal `(dec=)` est la virgule, les champs alphanumériques `(quote=)` sont entre guillemets et ils contiennent des intitulés de colonne `(header=)`. Pour l'importation il faut donc sélectionner ces options dans les menus . Si on utilise l'interface graphique (onglet « Importation » de la fenêtre en haut à droite de RStudio) ou bien les préciser si on utilise la fonction `read.table()`.

```
dRef9907 <- read.table("data/data99_07.csv",
                       sep=";",
                       dec=",",
                       quote = "\"",
                       header = TRUE,
                       encoding = "latin1"
                       )

dPop3608 <- read.table("data/pop36_08.csv",
                       sep=";",
                       dec=",",
                       header = TRUE,
                       quote = "\"",
                       encoding = "latin1"
                       )
```

Maintenant que les données sont importées, nous pouvons contrôler leur contenu soit en cliquant sur le tableau qui apparaît dans la fenêtre « Espace de travail », soit avec les fonctions `str()`, `head()`, `dim()` et/ou `class()`.

Pour le besoin des analyses menées dans ce chapitre, il est nécessaire de créer une variable `CODDEP`, renseignant sur le département d'appartenance des communes.

Une variable `DISTCONTINUE` renseignant sur la distance euclidienne entre le centroïde des communes et celui du premier arrondissement parisien est également nécessaire (cf. Chapitre chap :uni). L'utilisation de la fonction puissance dans la formule se fait, dans R, par l'introduction des signes `**` ou `^`.

Les densités d'habitants au kilomètre carré par commune (`DENS08`) seront également utilisées. Leur calcul nécessite une jointure. Celle-ci permet de récupérer la variable de population du tableau `dPop3608` et de l'insérer dans le tableau `dRef9907` où se trouvent les variables avec lesquelles les analyses seront réalisées *a posteriori*. Attention, les données utilisées requièrent une correction de la surface enregistrée pour le 12^{e} et le 16^{e} arrondissements, afin de soustraire la surface des bois de Vincennes (995 ha) et de Boulogne (846 ha).

Enfin, une variable `RFUCQ207CLASS` distinguant les revenus inférieurs et les revenus supérieurs au revenu médian de l'espace d'étude sera utilisée. La discrétisation se fait à l'aide de la fonction `quantile()` qui découpe une série continue en groupe d'effectifs égaux. L'option `(probs=)` permet de fixer le nombre de ces groupes *via* les bornes de fréquence et l'option `(include.lowest=)` permet d'ouvrir ou de fermer les intervalles à gauche.

```
# Définition des codes départementaux
dRef9907$CODDEP <- substr(dRef9907$CODGEO, 1, 2)

# Calcul des distances euclidiennes au centre de Paris
vCoordPremierArr <- as.numeric(dRef9907[1, 3:4])
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - vCoordPremierArr[1]) ** 2 +
                              (dRef9907$Y - vCoordPremierArr[2]) ** 2) / 10

# Calcul des densités d'habitants par km2
vSurfBoisVincennes <- 995
vSurfBoisBoulogne <- 846
dRef9907 <- merge(dRef9907, dPop3608[ , c(1, 11)], by = "CODGEO")
dRef9907[12, 5] <- dRef9907[12, 5] - vSurfBoisVincennes
dRef9907[16, 5] <- dRef9907[16, 5] - vSurfBoisBoulogne
dRef9907$DENS08 <- dRef9907$POP2008 / (dRef9907$SURF * 0.01)

# Discrétisation autour du revenu médian
vBornes <- quantile(dRef9907$RFUCQ207, probs = seq(0, 1, length=3))
dRef9907$RFUCQ207CLASS <- cut(dRef9907$RFUCQ207,
                              breaks = vBornes,
                              labels = c("Inférieur", "Supérieur"),
                              include.lowest = TRUE)
```

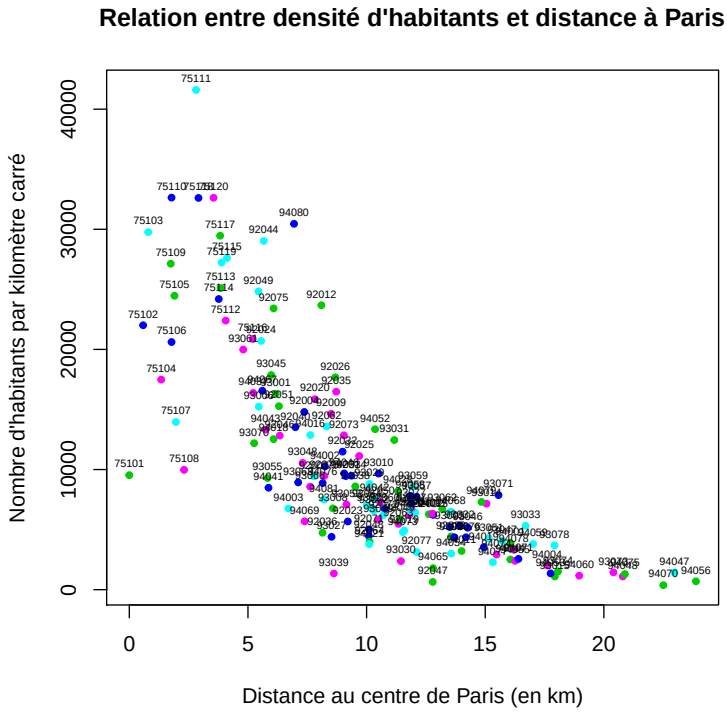
5.1 Relation entre deux variables quantitatives

Dans cet exemple, on s'intéresse à la relation entre la densité de population et la distance au centre de l'agglomération.

Représentation graphique de la relation

La fonction `plot()` est une fonction graphique générique (*i.e.* dont le type de sortie dépend du type d'objet qu'elle prend en entrée). Appliquée à deux variables quantitatives (vecteurs numériques), on l'utilise pour réaliser un nuage de points (*scatterplot*). Pour faciliter l'interprétation du graphique, les communes sont différenciées selon leur département d'origine à l'aide de la variable visuelle couleur. Enfin, la fonction `text()` permet d'ajouter une étiquette identifiant chaque point afin de rendre plus aisé, dans le cadre d'une démarche exploratoire, le repérage des points qui s'écartent le plus de la tendance.

```
plot(dRef9907$DISTCONTINUE, dRef9907$DENS08,
     type = "p",
     pch = 20,
     col=levels(factor(dRef9907$CODDEP)),
     xlab = "Distance au centre de Paris (en km)",
     ylab = "Nombre d'habitants par kilomètre carré",
     main="Relation entre densité d'habitants et distance à Paris")

text(x = dRef9907$DISTCONTINUE,
     y = dRef9907$DENS08,
     labels = dRef9907$CODGEO,
     adj = c(0.5, -1), # spécifie l'ajustement des labels
     cex = 0.5) # spécifie la taille des labels
```



A 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 ($densit = aexp^{-bx}$ où a est la densité au centre et b le gradient de densité).

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 les variables 23 à 35 du tableau de données, c'est à dire l'ensemble des variables quantitatives allant de `P20ANS07` à `DISTCLASS` décrivant les communes de la petite couronne et les arrondissements parisiens pour l'année 2007 que nous intégrons dans un nouveau *data.frame*. La méthode utilisé ici est celle de Pearson, mais l'argument `method=` permet d'en utiliser d'autres.

```
# Taux de chômage, les 20 ans, les non diplômés, les cadres et les
# ouvriers.
dVarCorr <- dRef9907[, c((23:35), 37)]
# Et montrer la matrice.
cor(dVarCorr, method = "pearson")
```

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 d'un cœfficient de corrélation est bornée dans un intervalle compris entre -1 et $+1$. Un cœfficient négatif proche de -1 signifie une forte corrélation négative entre les variables : quand l'une augmente, l'autre a tendance à décroître. A 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 en même temps. Enfin, un cœfficient de corrélation proche de zéro décrit une absence de relation linéaire entre les deux variables.

La figure précédente montre que la relation entre les deux variables n'est pas linéaire. Soit on utilise une méthode qui ne requiert pas la linéarité de la relation (Spearman), soit on travaille sur une variable transformée, ici le logarithme de la variable de densité. La fonction `cor.test()` fournit les résultats de tests statistiques de corrélation paramétrique (r de Pearson) et non paramétrique (r de Kendall ou de Spearman).

```
cor.test(dRef9907$DISTCONTINUE, dRef9907$DENS08, method = "spearman")
```

```
##
##  Spearman's rank correlation rho
##
## data:  dRef9907$DISTCONTINUE and dRef9907$DENS08
## S = 907838, p-value < 2.2e-16
## alternative hypothesis: true rho is not equal to 0
## sample estimates:
##     rho
## -0.8628
```

```
cor.test(dRef9907$DISTCONTINUE, log(dRef9907$DENS08), method = "pearson")
```

```
##
##  Pearson's product-moment correlation
##
## data:  dRef9907$DISTCONTINUE and log(dRef9907$DENS08)
## t = -19.41, df = 141, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.8923 -0.8011
## sample estimates:
##     cor
## -0.8531
```

La fonction `cor.test()` donne les éléments suivants en sortie : la statistique de test, le degré de liberté, la valeur de la probabilité critique (valeur de p ou *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 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, et inversement.

Régression linéaire simple

La relation linéaire entre les deux variables (quantitatives) est modélisée par le biais de la fonction `lm()`, laquelle permet de réaliser 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).

On cherche à estimer la valeur des paramètres de l'équation $y = \alpha * x + \beta$ où y est le logarithme de la densité et x la distance au centre, ici le centroïde du premier arrondissement parisien.

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 : description des résidus, des cœfficients (paramètres estimés) et de la qualité de l'ajustement linéaire.

Plus particulièrement, la matrice des cœfficients renseigne, pour chacun des deux paramètres du modèle que sont la constante β (intercept) et la pente α associée à la variable de distance, sur :

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

La signification des tests diffère entre l'intercept et la variable explicative. Dans le premier cas, une probabilité critique inférieure au seuil choisi pour le risque de première espèce (ex : une probabilité critique inférieure à 5%) signifie que la constante doit être prise en compte dans le modèle. Une même valeur de probabilité critique pour la variable explicative (la pente du modèle) révèle une relation linéaire statistiquement significative entre la variable dépendante et la variable indépendante. Dans une régression linéaire simple, la validité du cœfficient et la validité globale du modèle (obtenue par la mise en place d'un test de Fisher et donnée par la valeur de p lui étant associée) sont identiques.

```
model1 <- lm(log(dRef9907$DENS08) ~ dRef9907$DISTCONTINUE)
names(model1)

##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"

summary(model1)

##
## Call:
## lm(formula = log(dRef9907$DENS08) ~ dRef9907$DISTCONTINUE)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.0196 -0.2191  0.0617  0.2887  0.9287
##
## Coefficients:
##                         Estimate Std. Error t value Pr(>|t|)
## (Intercept)             10.47422    0.09006   116.3   <2e-16 ***
## dRef9907$DISTCONTINUE -0.15541    0.00801   -19.4   <2e-16 ***
```

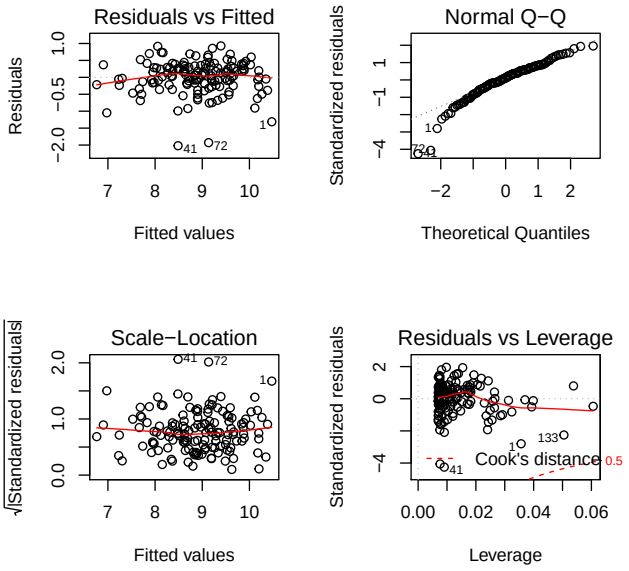
```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.477 on 141 degrees of freedom
## Multiple R-squared: 0.728,Adjusted R-squared: 0.726
## F-statistic:  377 on 1 and 141 DF,  p-value: <2e-16
```

Le cœfficient de détermination est égal à $0,72$, ce qui signifie que, en 2008, 72% de la variation du logarithme de la densité de population des communes étudiées s'expliquait par la variation de leur distance au centre de Paris. Le signe du paramètre α confirme l'hypothèse sur le sens de la relation entre les variables élaborée à la suite des analyses de corrélation : 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 β, nommé *Intercept*, informe quant à lui sur la valeur théorique de la densité de population lorsque la distance au centre de Paris est égale à 0. Dans cet exemple, la densité de population estimée pour le premier arrondissement de Paris est de $exp(10.468386) = 35185$ habitants au kilomètre carré.

La fonction `plot()`, lorsqu'elle est 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)

```
par(mfrow = c(2,2))
plot(model1)
```

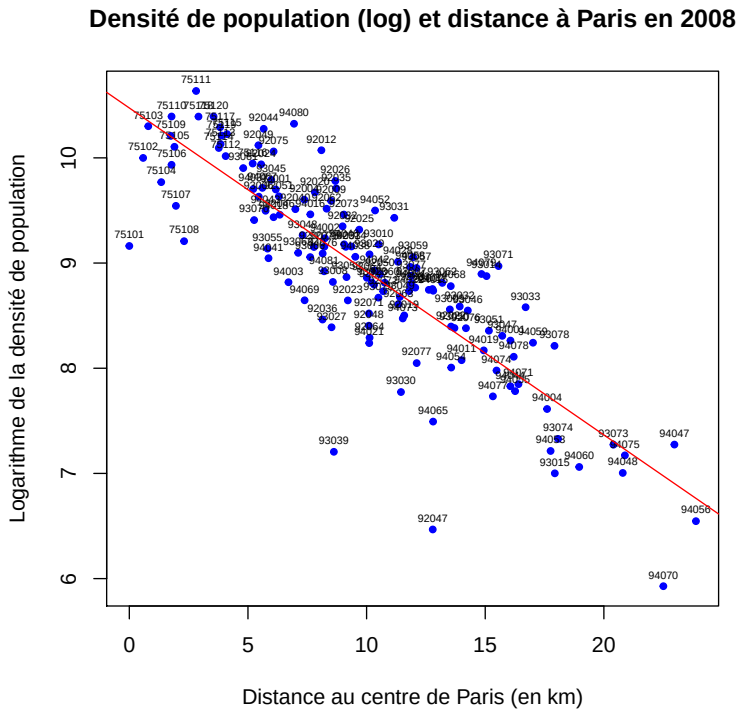


L'étude des graphiques montre qu'il n'y a ni homoscédasticité des résidus, ni auto-corrélation de ces derniers. En effet, les deux figures renseignant sur la normalité des résidus (figures du haut) montrent que la variance des résidus a tendance à être constante quelque soit la valeur de la densité. 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 densité. Les densités de population ne sont pas plus

sur- ou sous-estimées par la distance au centre selon qu'elles sont faibles ou élevées. Enfin, il serait possible de recommencer l'analyse de régression sans intégrer les cas extrêmes dont figurent les labels dans la figure 4 (en bas à droite), afin d'observer un meilleur ajustement de la relation.

Il est également intéressant de distinguer les individus statistiques en fonction de leur écart au modèle en surimposant la droite de régression linéaire au nuage de points formé par le logarithme de la densité de population des communes en 2008 (variable dépendante du modèle) et la distance au centre de Paris (variable indépendante du modèle).

```
plot(dRef9907$DISTCONTINUE,
     log(dRef9907$DENS08),
     pch=20,
     col="blue",
     xlab="Distance au centre de Paris (en km)",
     ylab="Logarithme de la densité de population",
     main="Densité de population (log) et distance à Paris en 2008")

text(x=dRef9907$DISTCONTINUE,
     y=log(dRef9907$DENS08),
     labels = dRef9907$CODGEO,
     adj = c(0.5, -1),
     cex = 0.5)

abline(model1, col='red')
```



Cette analyse visuelle montre, par exemple, que le 11ème arrondissement de Paris a une densité de population encore plus importante que ce que laisserait supposer sa distance au centre de Paris, alors qu'une commune comme l'Ile-Saint-Denis a une densité de population plus faible que ce que l'on serait en mesure d'attendre relativement à sa distance au centre de Paris. Dans l'hypothèse où ces écarts seraient liés à un processus spatial plutôt qu'à une composante aléatoire ou à une sous-détermination du modèle, il est utile de cartographier les résidus. Les outils de cartographie sous R sont présentés dans le Chapitre 9.

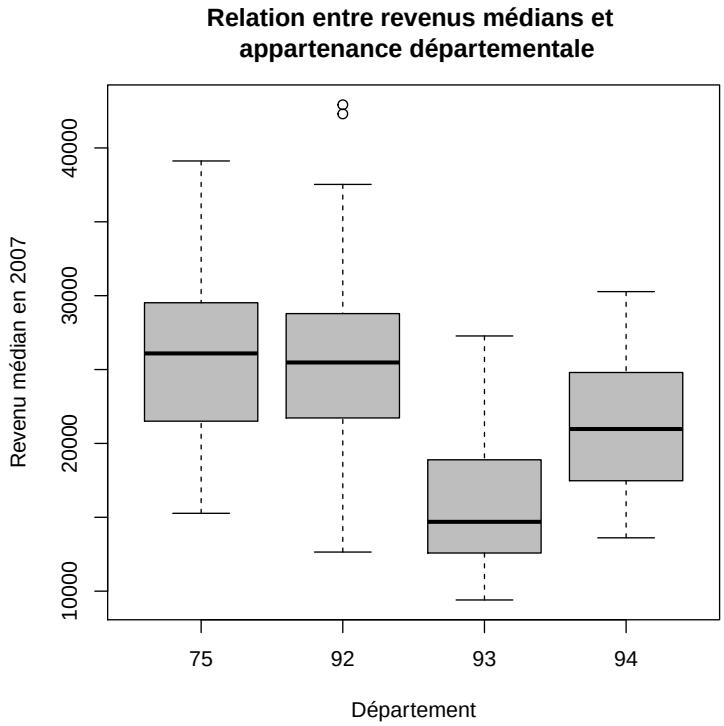
5.2 Relation entre une variable quantitative et une variable qualitative

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

Représentation graphique de la relation

La fonction `boxplot()` permet de réaliser des graphiques appelés diagrammes en boîtes ou boîtes à moustaches. Ces graphiques résument visuellement la distribution statistique d'une variable quantitative. Si cela est spécifié, la fonction `boxplot()` permet de représenter la distribution en fonction de l'appartenance aux modalités de la variable qualitative.

```
boxplot(dRef9907$RFUCQ207 ~ dRef9907$CODDEP,
        col = "grey",
        names = c("75","92","93","94"),
        ylab="Revenu médian en 2007",
        xlab="Département",
        main="Relation entre revenus médians et \n appartenance départementale")
```



Analyse de la variance à un facteur

Une analyse de la variance à un facteur (ANOVA) est mise en place afin de déterminer si l'appartenance départementale (variable catégorielle) permet d'expliquer les différences de revenus médians (variable quantitative) entre communes.

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. Dans R, il est recommandé d'estimer les paramètres du modèle avec la fonction `lm()` au préalable de l'analyse du tableau de la variance donné par la fonction `anova()`. On peut également regarder les résultats du test en affichant le `summary()` de la fonction `aov()` intégrant la formule d'analyse de variance. Celle-ci représente la décomposition

de la variance totale en la variance due au facteur (variance intergroupe) plus 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()`.

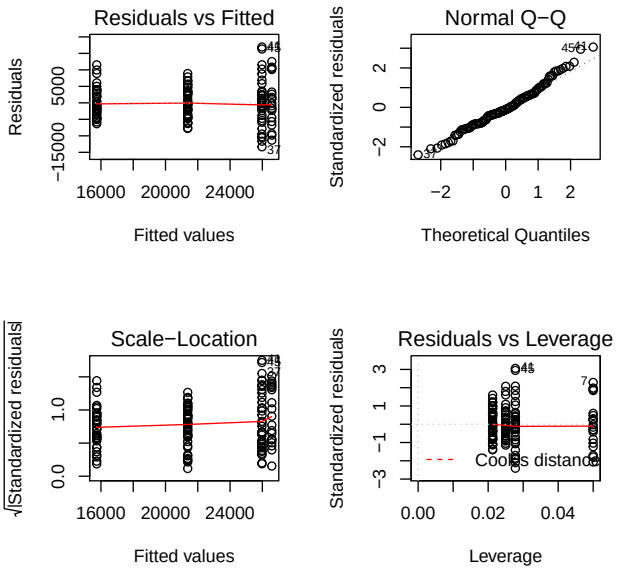
```
# Transformation de la variable qualitative
is.factor(dRef9907$CODDEP)

## [1] FALSE

dRef9907$CODDEP <- as.factor(dRef9907$CODDEP)

# Anova - Option 1
model2 <- lm(dRef9907$RFUCQ207~dRef9907$CODDEP)
anova(model2)

## Analysis of Variance Table
##
## Response: dRef9907$RFUCQ207
##                   Df   Sum Sq  Mean Sq F value  Pr(>F)
## dRef9907$CODDEP    3 2.56e+09 8.54e+08      27 7.8e-14 ***
## Residuals        139 4.39e+09 3.16e+07
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1



# Anova - Option 2
model2bis <- aov(dRef9907$RFUCQ207~dRef9907$CODDEP)
summary(model2bis)

##                   Df   Sum Sq  Mean Sq F value  Pr(>F)
## dRef9907$CODDEP    3 2.56e+09 8.54e+08      27 7.8e-14 ***
## Residuals        139 4.39e+09 3.16e+07
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

La table `df` donne les éléments de calcul du test global du modèle, à savoir le Fisher, les degrés de liberté associés aux variances et *Sum Sq* la somme des carrés des écarts. L'élément important est la significativité du test de Fisher qui est donnée sous l'étiquette $Pr(> F)$, c'est à dire le risque d'erreur en utilisant les départements pour expliquer les variations de revenus. Ici ce risque est très inférieur à 0.001.

Tout comme avec le modèle 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. On présente ici les quatre principales :

- 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)

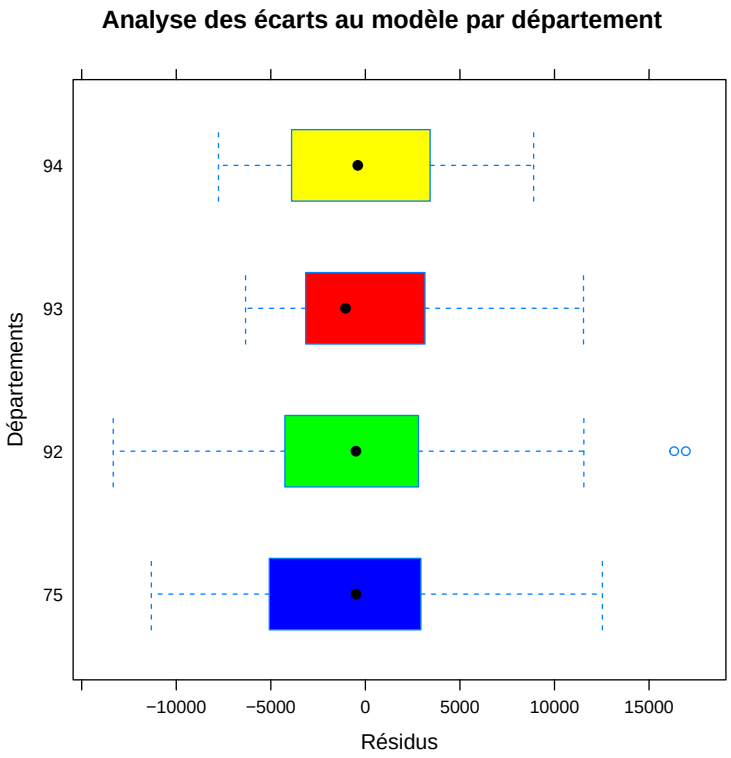
```
par(mfrow = c(2,2))
plot(model2bis)
```



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 une moins bonne adaptation 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 unités 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. Ici, nous utilisons les graphiques du *package* `lattice` afin d'obtenir une mise en page différente de celle obtenue avec la fonction `boxplot()`, qui est la fonction par défaut du *package* `graphics`.

```
library(lattice)

bwplot(CODDEP ~ residuals(model2),
       data = dRef9907,
       fill = c("blue", "green", "red", "yellow"),
       ylab = "Départements",
       xlab = "Résidus",
       main = "Analyse des écarts au modèle par département")
```



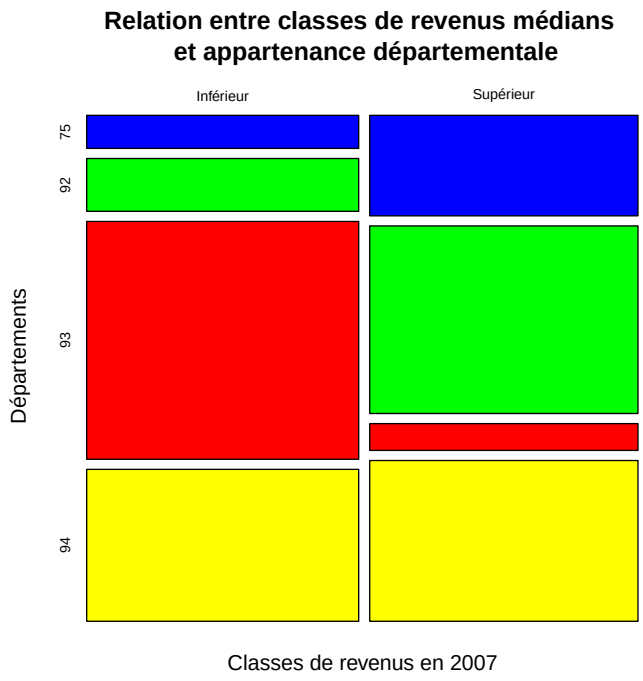
5.3 Relation entre deux variables qualitatives

Cet exercice traite de la même question que précédemment en proposant une résolution un peu différente : le revenu est ici discrétisé en une variable qualitative à deux modalités (selon s'il est inférieur ou supérieur à la médiane des revenus médians observés en 2007) et leur appartenance territoriale (mesurée à l'échelle départementale).

Représentation graphique de la relation

La fonction `mosaicplot()` permet de représenter la relation entre deux variables qualitatives en rendant proportionnelle la répartition conjointe des sous-populations des deux variables.

```
mosaicplot(~ dRef9907$RFUCQ207CLASS + dRef9907$CODDEP,
           color = c("blue", "green", "red", "yellow"),
           ylab="Départements",
           xlab="Classes de revenus en 2007",
           main="Relation entre classes de revenus médians \n et appartenance départementale")
```



Test d'indépendance du χ^2 : relation entre classes de revenus et appartenance départementale

Le test du χ^2 permet de mesurer l'écart entre deux distributions statistiques. Ici, il s'agit de la distribution observée et de la distribution théorique obtenue sous l'hypothèse d'indépendance entre les deux variables : $n^*_{i,j} = \frac{n_i * n_j}{n}$ avec n la valeur de l'effectif, i le numéro de ligne et j le numéro de colonne. Nous rappelons que la dépendance statistique n'implique pas la causalité, et que l'existence d'une relation de causalité ne pourra pas être établie à partir de l'analyse du tableau de contigence. L'analyse ci-dessous permet toutefois de savoir si la connaissance du département d'appartenance d'une commune peut informer, ou non, sur son appartenance à une catégorie de revenus particulière.

```
# Création de la table de contingence des effectifs observés
mTabContingence <- table(dRef9907$CODDEP, dRef9907$RFUCQ207CLASS)
colnames(mTabContingence) <- c("Revenus inférieurs","Revenus supérieurs")
mTabContingence
```

```
##
##      Revenus inférieurs Revenus supérieurs
##   75                  5                 15
##   92                  8                 28
##   93                 36                  4
##   94                 23                 24
```

L'observation du tableau de contingence laisse penser que la répartition des communes par classes de revenus entre départements n'est pas homogène. Le test du χ^2 réalisé avec la fonction `chisq.test()` va permettre de mesurer la validité statistique d'une telle hypothèse. Au préalable, il est nécessaire de s'assurer que les effectifs théoriques (sous l'hypothèse d'une équirépartition entre les modalités des deux variables) sont suffisamment grands (supérieurs ou égaux à cinq).

Les effectifs théoriques sont calculés par la fonction `chisq.test()` :

```
# Calcul des effectifs théoriques
model3 <- chisq.test(mTabContingence)
model3$expected
```

```
##
##      Revenus inférieurs Revenus supérieurs
##   75              10.07               9.93
##   92              18.13              17.87
##   93              20.14              19.86
##   94              23.66              23.34
```

```
# Test du $\chi^2$
model3 <- chisq.test(mTabContingence)
model3
```

```
##
##  Pearson's Chi-squared test
##
## data:  mTabContingence
## X-squared = 41.73, df = 3, p-value = 4.584e-09
```

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%. Nous pouvons rejeter l'hypothèse d'indépendance pour un risque de première espèce (rejeter l'hypothèse nulle alors que celle-ci est vraie) de 0.001%.

On peut maintenant chercher à savoir quelles sont les combinaisons de modalités qui ont le plus contribué à la statistique du χ^2 observée, autrement dit à la non-indépendance des deux facteurs. La fonction `chisq.test()` crée un objet de classe *htest* qui contient, entre autres choses, les résidus qui représentent les racines carrés de ces contributions et à partir desquels ont été calculées les contributions sous la forme de pourcentages (arrondis au centième). Enfin, l'analyse des résidus et de leurs signes permet de connaître les associations de modalités qui contribuent le plus à l'écart de répartition par rapport au profil d'équirépartition. Par exemple, dans la table des résidus absolus on peut lire que le département des Hauts-de-Seine (92) a 10 communes en moins dans la classe des revenus inférieurs qu'il n'aurait eu si la répartition était la même que sur l'ensemble de la région.

```
# Calcul des contributions
round(100 * model3$residuals^2 / model3$stat, 2)
```

```
##
##      Revenus inférieurs Revenus supérieurs
##   75               6.12               6.20
##   92              13.56              13.75
##   93              29.93              30.35
##   94               0.04               0.05
```

```
# Calcul des résidus Khi-2
round(model3$residuals, 2)
```

```
##
##      Revenus inférieurs Revenus supérieurs
##   75              -1.60               1.61
##   92              -2.38               2.40
##   93               3.53              -3.56
##   94              -0.14               0.14
```

```
# Calcul des résidus absolus
resabs <- mTabContingence - model3$expected
round(resabs, 2)
```

```
##
##      Revenus inférieurs Revenus supérieurs
##   75              -5.07               5.07
##   92             -10.13              10.13
##   93              15.86             -15.86
##   94              -0.66               0.66
```

```
# Calcul des résidus relatifs
resrel <- mTabContingence / model3$expected
round(resrel, 2)
```

```
##
##      Revenus inférieurs Revenus supérieurs
##   75               0.50               1.51
##   92               0.44               1.57
##   93               1.79               0.20
##   94               0.97               1.03
```

Le nombre de communes ayant un revenu supérieur à la médiane des revenus des communes de la petite couronne est donc moins important dans le département de la Seine-Saint-Denis (93) que ce que l'on pourrait attendre dans le cas d'une indépendance statistique entre les deux variables. A l'inverse, les communes des Haut-de-Seine (92) et les arrondissements parisiens (75) sont plus nombreux à avoir un revenu médian supérieur au revenu médian de la petite couronne qu'attendu. Le département du Val-de-Marne (94) observe, quant à lui, une relative bonne équirépartition de ses communes entre les deux classes de revenu.

5.4 Code

```
# Importation des données
dRef9907 <- read.table("data99_07.csv",
                       sep=";",
                       dec=",",
                       quote = "\"",
                       header = TRUE,
                       encoding = "latin1"
                       )

dPop3608 <- read.table("pop36_08.csv",
                       sep=";",
                       dec=",",
                       header = TRUE,
                       quote = "\"",
                       encoding = "latin1"
                       )

# Définition des codes départementaux
dRef9907$CODDEP <- substr(dRef9907$CODGEO, 1, 2)

# Calcul des distances euclidiennes au centre de Paris
vCoordPremierArr <- as.numeric(dRef9907[1, 3:4])
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - vCoordPremierArr[1]) ** 2 +
                              (dRef9907$Y - vCoordPremierArr[2]) ** 2) / 10

# Calcul des densités d'habitants par km2
vSurfBoisVincennes <- 995
vSurfBoisBoulogne <- 846
dRef9907 <- merge(dRef9907, dPop3608[ , c(1, 11)], by = "CODGEO")
```

```r
dRef9907[12, 5] <- dRef9907[12, 5] - vSurfBoisVincennes
dRef9907[16, 5] <- dRef9907[16, 5] - vSurfBoisBoulogne
dRef9907$DENS08 <- dRef9907$POP2008 / (dRef9907$SURF * 0.01)

# Discrétisation autour du revenu médian
vBornes <- quantile(dRef9907$RFUCQ207, probs = seq(0, 1, length=3))
dRef9907$RFUCQ207CLASS <- cut(dRef9907$RFUCQ207,
                              breaks = vBornes,
                              labels = c("Inférieur", "Supérieur"),
                              include.lowest = TRUE)

# Analyse des corrélations
dVarCorr <- dRef9907[, c((23:35),37)]
cor(dVarCorr, method = "pearson")
cor.test(dRef9907$DISTCONTINUE, log(dRef9907$DENS08), method="pearson")

# Régression linéaire
model1 <- lm(log(dRef9907$DENS08)~dRef9907$DISTCONTINUE)
names(model1)
summary(model1)

# Transformation de la variable département en facteur (si besoin)
is.factor(dRef9907$CODDEP)
dRef9907$CODDEP <- as.factor(dRef9907$CODDEP)

# Réalisation de l'Anova: méthode 1
model2 <- lm(dRef9907$RFUCQ207 ~ dRef9907$CODDEP)
anova(model2)

# Réalisation de l'Anova: méthode 2
model2bis <- aov(dRef9907$RFUCQ207 ~ dRef9907$CODDEP)
summary(model2bis)

# Création de la table de contingence
mTabContingence <- table(dRef9907$CODDEP, dRef9907$RFUCQ207CLASS)
colnames(mTabContingence) <- c("Revenu inférieur", "Revenu supérieur")
mTabContingence

# Calcul des effectifs théoriques
model3 <- chisq.test(mTabContingence)
model3$expected

# Test du $\chi^2$
model3 <- chisq.test(mTabContingence)
model3

# Calcul des contributions
round(100 * model3$residuals^2 / model3$stat, 2)

# Calcul des résidus Khi-2
round(model3$residuals, 2)

# Calcul des résidus absolus
resabs <- mTabContingence-expected
round(resabs, 2)

# Calcul des résidus relatifs
resrel <- mTabContingence/expected
```

Chapitre 6

Analyses factorielles

Objectifs

Ce chapitre présente deux exemples d'analyses multivariées sous R : une analyse en composantes principales (ou ACP) et une analyse factorielle des correspondances (ou AFC). 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. Les analyses réalisées dans ce chapitre s'appuient sur [Sanders 1989] et sur [Husson at al. 2009].

Prérequis

Analyse en composantes principales et analyse factorielle des correspondances.

Packages nécessaires

Pour réaliser l'ACP et l'AFC, nous avons sélectionné des fonctions provenant du *package* `FactoMineR`. Ce *package*, dédié à l'analyse de données multivariées, a été développé par F. Husson, J. Josse, S. Lê, et J. Mazet et intègre un ensemble de méthodes dites « à la française ».

Ce *package* nécessite le chargement de quatre autres *packages* : `ellipse` qui permettra de dessiner des ellipses de confiance dans certaines sorties graphiques des analyses, `lattice` qui propose des options graphiques supplémentaires à celles offertes dans les *packages* de base, `cluster` puisque le *package* `FactoMiner` propose également des méthodes de classification et `scatterplot3d` lequel, comme son nom l'indique, permet de dessiner des nuages de points en 3 dimensions. Le téléchargement de ces dépendances peut se faire automatiquement lors de l'installation du *package* par le biais de la fonction `install.package()` en ajoutant l'option `(dependencies=)`.

```
library(ellipse)
library(lattice)
library(cluster)
library(scatterplot3d)
library(FactoMineR)
```

Données nécessaires

Dans ce chapitre nous utilisons deux tableaux de données produites par l'INSEE :

- `data99_07.csv` : données de référence pour 1999 et 2007 avec différentes variables sur la composition socio-professionnelle des communes de Paris et la petite couronne.
- `pop36_08.csv` : données de population sans double compte des recensements de la population de 1936 à 2008 pour Paris et la petite couronne.

Pour une description plus précise du contenu des fichiers, voir Section 1.6.

```
dRef9907 <- read.table("data/data99_07.csv",
                       sep = ";",
                       dec = ",",
                       quote = "\"",
                       header = TRUE
                       )


dPop3608 <- read.table("data/pop36_08.csv",
                       sep = ";",
                       dec = ",",
                       quote = "\"",
                       header = TRUE
                       )
```

Maintenant que les données sont importées, nous pouvons contrôler leur contenu soit en cliquant sur le tableau qui apparaît dans la fenêtre « Espace de travail », soit avec les fonctions `str()`, `head()`, `dim()` et/ou `class()`.

Préparation des données

Pour le besoin des analyses menées dans la première partie ce chapitre, la création d'un tableau de données contenant les variables suivantes est nécessaire :

1. une variable renseignant le département sur le tableau d'origine,
2. une variable calculant pour chaque commune sa distance à Paris (voir le Chapitre 4),
3. les identifiants des communes (codes INSEE),
4. et l'ensemble des variables entre la part des moins de 20 ans jusqu'aux deux variables qui viennent d'être créées.

Le tableau est nommé `SOC07`.

Pour faciliter les manipulations suivantes, les numéros de colonne correspondant aux identifiants sont également indiqués.

```
dRef9907$CODDEP <- substr(dRef9907$CODGEO, 1, 2)

vCoordPremierArr <- as.numeric(dRef9907[1, 3:4])
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - vCoordPremierArr[1])^2 + (dRef9907$X -
    vCoordPremierArr[1])^2)/10

SOC07 <- data.frame(dRef9907$CODGEO, dRef9907[, 23:37], row.names = 1)
```

Pour la deuxième partie du chapitre, il est nécessaire de construire un tableau à partir du tableau de données `dPop3608` ne gardant que les variables de population et désignant les codes postaux des communes comme les identifiants des individus statistiques.

```
dPopEvol <- data.frame(dPop3608[, -c(2, 12)], row.names = 1)
```

Ce tableau est nommé `dPopEvol`.

6.1 Réaliser une analyse en composantes principales

Différenciations socio-démographiques entre communes

L'objectif de cet exercice est de déterminer les combinaisons de variables socio-démographiques qui introduisent le plus de différenciations entre les communes de la petite couronne parisienne.

Présentation de la méthode

L'analyse en composantes principales que nous allons réaliser porte sur les caractéristiques socio-démographiques des communes de la petite couronne et des arrondissements parisiens pour l'année 2007. Les données utilisées sont issues du tableau `SOC07` dans lequel les individus spatiaux (les communes) sont donnés en lignes et les variables décrivant les répartitions géographiques sont décrites en colonnes. Il est important de rappeler ici que l'ACP est une technique qui ne s'applique qu'aux variables quantitatives.

L'ACP va permettre de connaître la structure de différenciation entre communes de la petite couronne et arrondissements au regard des combinaisons de variables socio-démographiques contenues dans le tableau. La fonction `PCA()` (Principal Component Analysis) est utilisée à cette fin. Il s'agit d'une procédure qui produit un *object*, que l'on nommera ici `resACPSOC07` afin de pouvoir y faire appel par la suite .

Les données à analyser étant très hétérogènes, il faut centrer-réduire les variables en amont de l'ACP. Cette opération permet de faire ressortir la structure des données en neutralisant les ordres de grandeur des mesures de centralité et de dispersion. Attention, dans le *package* `FactoMineR`, la fonction `PCA()` standardise les variables par défaut. Il est possible de ne pas centrer-réduire les variables en utilisant l'argument `FALSE` dans l'option `(scale.unit=)`. Pour rappel, la réalisation d'une ACP non normée se fait à partir d'une matrice de covariances, et non plus de corrélations.

Nous allons ajouter deux variables « supplémentaires », qui n'entreront pas en compte dans la création des composantes de l'ACP : la variable quantitative rendant compte de la distance à Paris et la variable qualitative informant sur le département d'appartenance. Si les variables supplémentaires ne structurent pas directement l'information donnée par l'ACP, ces variables pourront être projetées par la suite sur les facteurs afin de nous permettre d'émettre des hypothèses quant à leur capacité à décrire les différenciations socio-démographiques des unités géographiques. A noter que la fonction `PCA()` ne permet pas d'appeler les variables par leurs noms mais requiert les indices de colonnes.

Enfin, une astuce technique : l'option `(graph=)` permet d'afficher ou non les graphiques associés à l'ACP (projections des variables actives et des individus sur le premier plan factoriel). Comme nous souhaitons d'abord étudier la qualité de la réduction de l'information, les liaisons linéaires entre variables et les similarités entre individus, nous avons choisi de ne pas afficher ces graphiques.

```
SOC07$CODDEP <- as.factor(SOC07$CODDEP)
resACPSOC07 <- PCA(SOC07, quanti.sup = 15, quali.sup = 14, graph = FALSE)
```

La hiérarchie des composantes principales

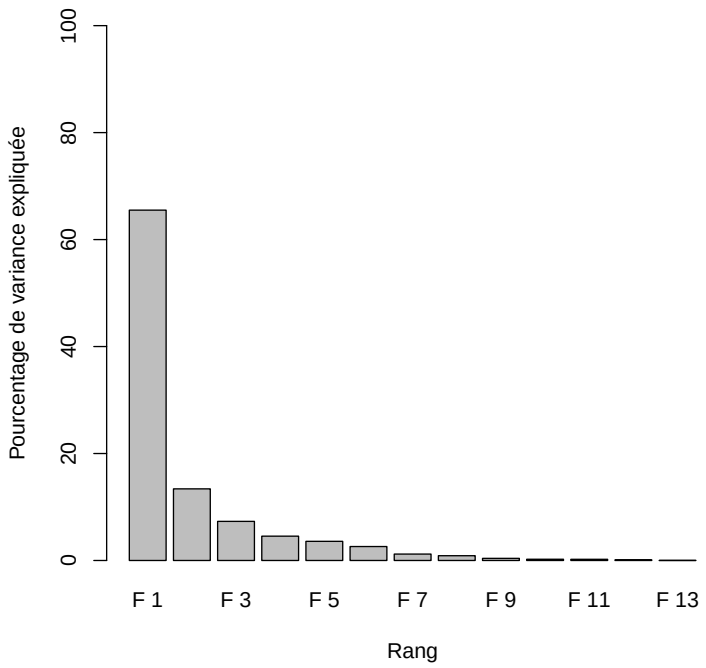
L'ACP consiste au passage d'un système d'observations dans un espace à p variables à un espace à p composantes principales. Les composantes correspondent à des combinaisons linéaires de l'ensemble des indicateurs analysés. Elles se hiérarchisent en fonction de la variance des coordonnées des individus 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 spatiale initiale.

Une telle analyse produit un certain nombre d'éléments pour caractériser ces composantes, à commencer par les valeurs propres (*eigenvalues* en anglais, d'où leur nomination sous le terme de « eig » dans la fonction `PCA()`). 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. Il est d'usage de regarder les parts relatives de chaque composante (*percentage of variance*), ainsi que leurs parts cumulées (*cumulative percentage of variance*).

```
head(resACPSOC07$eig)

##        eigenvalue percentage of variance cumulative percentage of variance
## comp 1     8.5158                 65.506                             65.51
## comp 2     1.7410                 13.392                             78.90
## comp 3     0.9505                  7.311                             86.21
## comp 4     0.5911                  4.547                             90.76
## comp 5     0.4653                  3.579                             94.34
## comp 6     0.3388                  2.606                             96.94
```

Une représentation du pourcentage de la valeur expliquée par chaque composante sous la forme d'un histogramme avec la fonction `barplot()` apporte une aide visuelle à la détermination des facteurs les plus utiles à l'analyse.

```
barplot(resACPSOC07$eig[, 2], names = paste("F", 1:nrow(resACPSOC07$eig)), xlab = "Rang",
    ylab = "Pourcentage de variance expliquée", ylim = c(0, 100))
```



Dans cet exemple, l'histogramme est très concentré, ce qui signifie que les valeurs propres des axes sont très différenciées et que la structure de la différenciation de l'espace résultante est forte. Il apparaît, en effet, 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. Les deux premiers axes factoriels, lesquels rendent compte de 78,9% de l'information, seront néanmoins analysés.

Interprétation des composantes principales

L'interprétation du rôle des variables (et des variables supplémentaires) dans la structuration de l'information apportée par l'ACP est permise l'observation de leurs coordonnées sur les deux premiers facteurs ; les coordonnées représentant le cœfficient de corrélation entre une variable et un axe (*i.e.* la proximité géométrique entre les vecteurs caractérisant les variables et l'axe).

```
resACPSOC07$var$coord[, 1:2]
```

```
##              Dim.1    Dim.2
## P20ANS07   0.7202 -0.16624
## PNDIP07    0.9700  0.12861
## TXCHOMA07  0.8997  0.31449
## INTA007    0.8309  0.08548
## PART07    -0.4007  0.49948
## PCAD07    -0.8949  0.34971
## PINT07    -0.3216 -0.85711
## PEMP07     0.9230 -0.25193
## POUV07     0.9683 -0.05472
## PRET07    -0.5978 -0.35649
## PMON007    0.8802 -0.25382
```

```
## PREFETR07  0.8520  0.42805
## RFUCQ207  -0.9185  0.20727
```

Pour rappel, les coordonnées varient entre -1 et $+1$. Les valeurs proche de 0 décrivent une faiblesse de la relation linéaire entre la variable et l'axe. A l'inverse, des valeurs absolues élevées (proches de -1 ou $+1$) indiquent une forte corrélation entre la variable et l'axe.

Dans cet exemple, l'axe 1 se caractérise par une opposition entre les communes bénéficiant principalement de valeurs élevées pour les variables suivantes :

- part des non diplômés dans la population de plus de 15 ans hors étude (`PNDIP07`) ;
- part des chômeurs dans la population active totale (`TXCHOMA07`) ;
- part des intérimaires dans les actifs occupés (`INTAO07`) ;
- part des employés dans les actifs occupés (`PEMP07`) ;
- part des ouvriers dans les actifs occupés (`POUV07`).

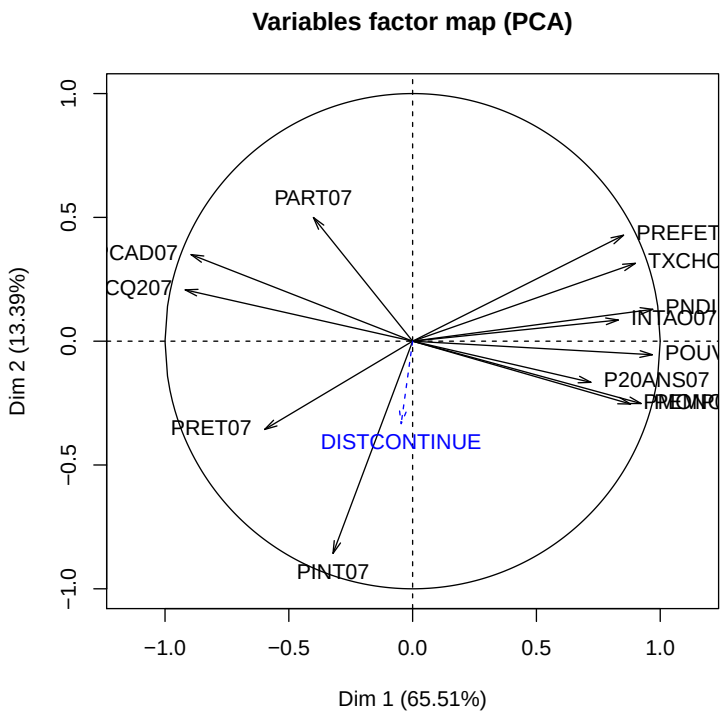
et les communes observant de fortes valeurs pour les variables :

- part des cadres dans actifs occupés (`PCAD07`) ;
- revenu médian en euros courants en 2007 (`RFUCQ207`).

Le deuxième axe 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 permet de mieux comprendre comment les deux premiers facteurs structurent l'espace des variables. On utilise pour cela la fonction `plot.PCA()` (ou simplement la fonction `plot()` une fois le *package* `FactoMineR` chargé). Il permet de visualiser le positionnement conjoint des variables `(choix=)` sur les deux premiers facteurs `(axes=)`. Par défaut, la fonction affiche également le centre de gravité des variables quantitatives supplémentaires ; ici la variable « Distance à Paris ».

```
plot(resACPSOC07, choix = "var", axes = 1:2, new.plot = F)
```

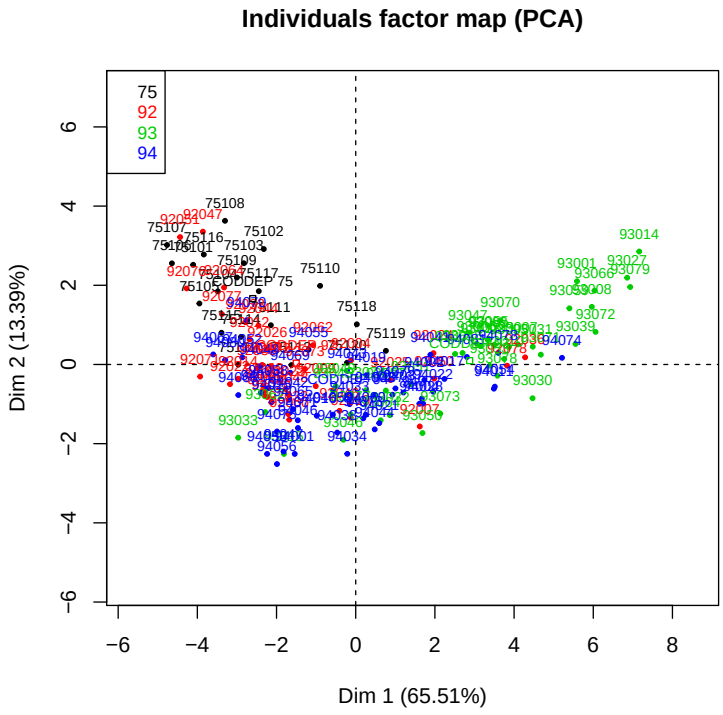


On remarque notamment que les communes ayant une part élevée de professions intermédiaires au sein des individus actifs sont plutôt corrélées avec une distance à Paris importante.

Grâce à la fonction `dimdesc()`, il est également possible de connaître la significativité des valeurs des cœfficients de corrélation entre une variable et une dimension.

Afin de comprendre comment les individus s'associent ou s'opposent entre eux, nous allons représenter leurs positions (`choix="ind"`) sur le premier plan factoriel (`axes=1:2`). L'option `(habillage=indice de la colonne)` permet de choisir la variable par laquelle les individus seront différenciés visuellement (ici le département). La différenciation par départements (`habillage=14`) offre ainsi une première idée du rôle d'une telle appartenance dans les différenciations et ressemblances entre individus statistiques.

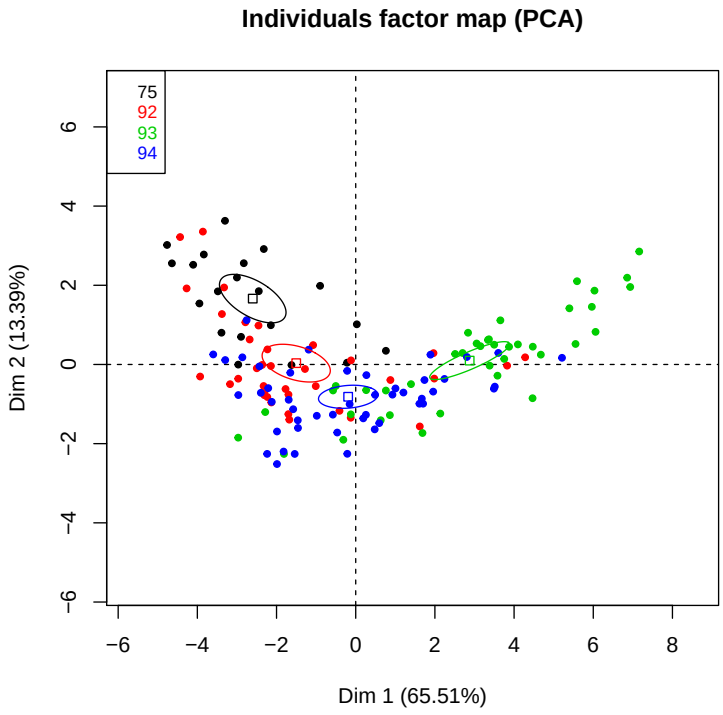
```
plot(resACPSOC07, choix = "ind", habillage = 14, axes = 1:2, new.plot = F, cex = 0.7)
```



Afin d'illustrer la capacité des départements à discriminer les individus, il est également possible de construire des ellipses de confiance autour des barycentres des départements. Il est au préalable nécessaire d'agréger les coordonnées des individus sur les facteurs par département par le biais de la fonction `cbind.data.frame()` pour créer des ellipses de confiance autour du barycentre avec la fonction `coord.ellipse()`. On ajoute ensuite l'option `(ellipse=)` dans la fonction `plot()` utilisée pour afficher les résultats de l'ACP.

```
concat <- cbind.data.frame(SOC07[, 14], resACPSOC07$ind$coord)
ellipse.coord <- coord.ellipse(concat, bary = T)
```

```
plot.PCA(resACPSOC07, habillage = 14, ellipse = ellipse.coord, label = FALSE)
```



Analyse des contributions

La contribution d'une variable à la formation d'un axe 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). Les contributions sont renseignées dans l'objet `contr` que nous allons exprimer en pour mille.

```
vContrVarF1 <- resACPSOC07$var$contr[, 1]/sum(resACPSOC07$var$contr[, 1]) *
    1000
summary(vContrVarF1)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    12.1    60.9    91.0    76.9    99.1   110.0

vContrVarF1

##  P20ANS07   PNDIP07 TXCHOMA07   INTA007    PART07    PCAD07    PINT07
##     60.91    110.50     95.06     81.08     18.85     94.05     12.14
##    PEMP07    POUV07    PRET07   PMON007 PREFETR07  RFUCQ207
##    100.05    110.11     41.96     90.98     85.23     99.08
```

Les hauteurs des contributions des différentes variables 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 est également donnée dans l'objet `contr`. Elle 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.

```
vContrIndF1 <- resACPSOC07$ind$contr[, 1]/sum(resACPSOC07$ind$contr[, 1]) *
    1000
summary(vContrIndF1)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    1.67    4.29    6.99    9.41   42.10

which(vContrIndF1 > 9.41)

## 75101 75104 75105 75106 75107 75115 75116 92036 92047 92051 92071 92076
##     1     4     5     6     7    15    16    37    41    45    50    54
## 92078 93001 93005 93007 93008 93010 93014 93027 93029 93030 93031 93039
##    56    57    58    60    61    62    64    66    67    68    69    72
## 93059 93066 93070 93071 93072 93078 93079 94011 94054 94067 94074 94078
##    83    88    89    90    91    95    96   101   123   130   136   140
```

On note que les communes ayant le plus contribué à déterminer la direction de l'axe d'allongement principal du nuage de points ont des départements d'appartenance plus variés que les communes qui sont le mieux représentées par cet axe. Ceci illustre parfaitement l'effet des différences de poids entre unités géographiques.

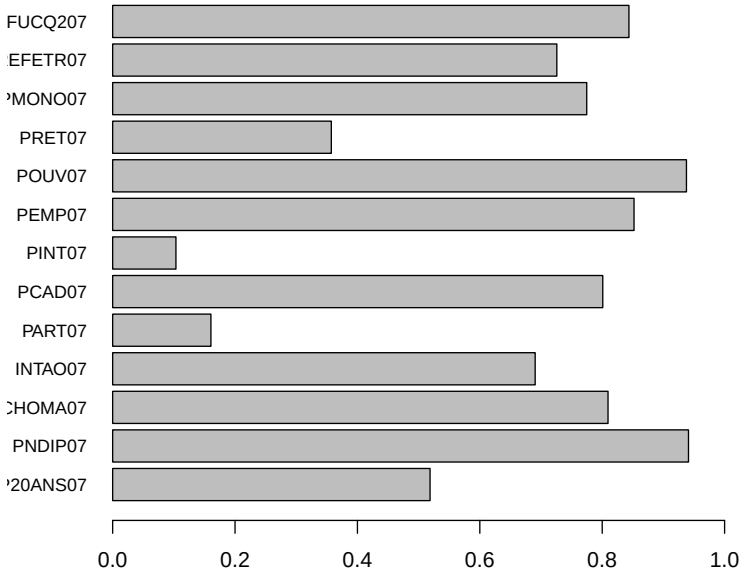
Analyse des qualités de représentations

La qualité de la représentation d'une variable par un axe est renseignée dans l'objet `cos2`. 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 cœfficient de corrélation. La somme des carrés des coordonnées d'une variable sur l'ensemble des axes est égale à 1.

```
resACPSOC07$var$cos2[, 1:2]

##             Dim.1    Dim.2
## P20ANS07  0.5187 0.027635
## PNDIP07   0.9410 0.016541
## TXCHOMA07 0.8095 0.098902
## INTA007   0.6904 0.007306
## PART07    0.1606 0.249483
## PCAD07    0.8009 0.122295
## PINT07    0.1034 0.734642
## PEMP07    0.8520 0.063469
## POUV07    0.9377 0.002994
## PRET07    0.3573 0.127086
## PMON007   0.7747 0.064425
## PREFETR07 0.7258 0.183230
## RFUCQ207  0.8437 0.042959
```

Nous pouvons faciliter l'analyse des qualités de représentation des variables par leur représentation graphique. Afin d'afficher le nom des variables, nous créons un histogramme horizontal. La taille des caractères est spécifiée par le biais de l'option `(cex.names=)` et les noms des variables sont écrits horizontalement grâce à l'option `(las=)`.

```
barplot(c(resACPSOC07$var$cos2[, 1]), horiz = T, xlim = c(0, 1), las = 1, cex.names = 0.8)
```



On remarque que les variables `RFUCQ207`, `POUV07`, `PEMP07`, `INTAO07`, `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 renseignée dans l'objet `cos2`. Elle mesure la proximité d'un individu à un axe : plus le cosinus est grand, plus petit sera l'angle séparant l'individu de l'axe par rapport au centre de gravité, et inversement. Un cosinus de 1 signifie que l'individu est confondu avec l'axe et que ce dernier explique 100% de la position particulière de cet individu par rapport à la moyenne. Il est possible de connaître les communes pour lesquelles les qualités de représentation sur le premier axe sont les plus fortes à l'aide de la fonction `which()`.

```
summary(resACPSOC07$ind$cos2[, 1])

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0001  0.3100  0.6100  0.5360  0.8060  0.9790

which(resACPSOC07$ind$cos2[, 1] > 0.82)

## 92012 92022 92035 92036 92078 93001 93005 93006 93007 93008 93010 93027
##    25    29    36    37    56    57    58    59    60    61    62    66
## 93029 93031 93039 93048 93053 93055 93059 93063 93066 93070 93071 93072
##    67    69    72    76    80    81    83    86    88    89    90    91
## 93079 94011 94017 94018 94052 94067 94074 94078 94080 94081
##    96   101   104   105   121   130   136   140   142   143
```

On remarque ainsi que 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) ; la majorité des communes bénéficiant des qualités de représentation sur le premier axe les plus élevées étant issues de l'un ou l'autre des deux départements à l'exception de quelques communes des Hauts-de-Seine (92).

6.2 Réaliser une analyse factorielle des correspondances

On souhaite à présent connaître 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.

Présentation de la méthode

Les analyses réalisées dans ce chapitre sont faites sur les données contenues dans le tableau `dPopEvol` créé en début de chapitre. Celui-ci dénombre les individus (ici la population résidante) 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). Les caractéristiques principales des 9 modalités sont résumées par la fonction `summary()`.

```
summary(dPopEvol)
```

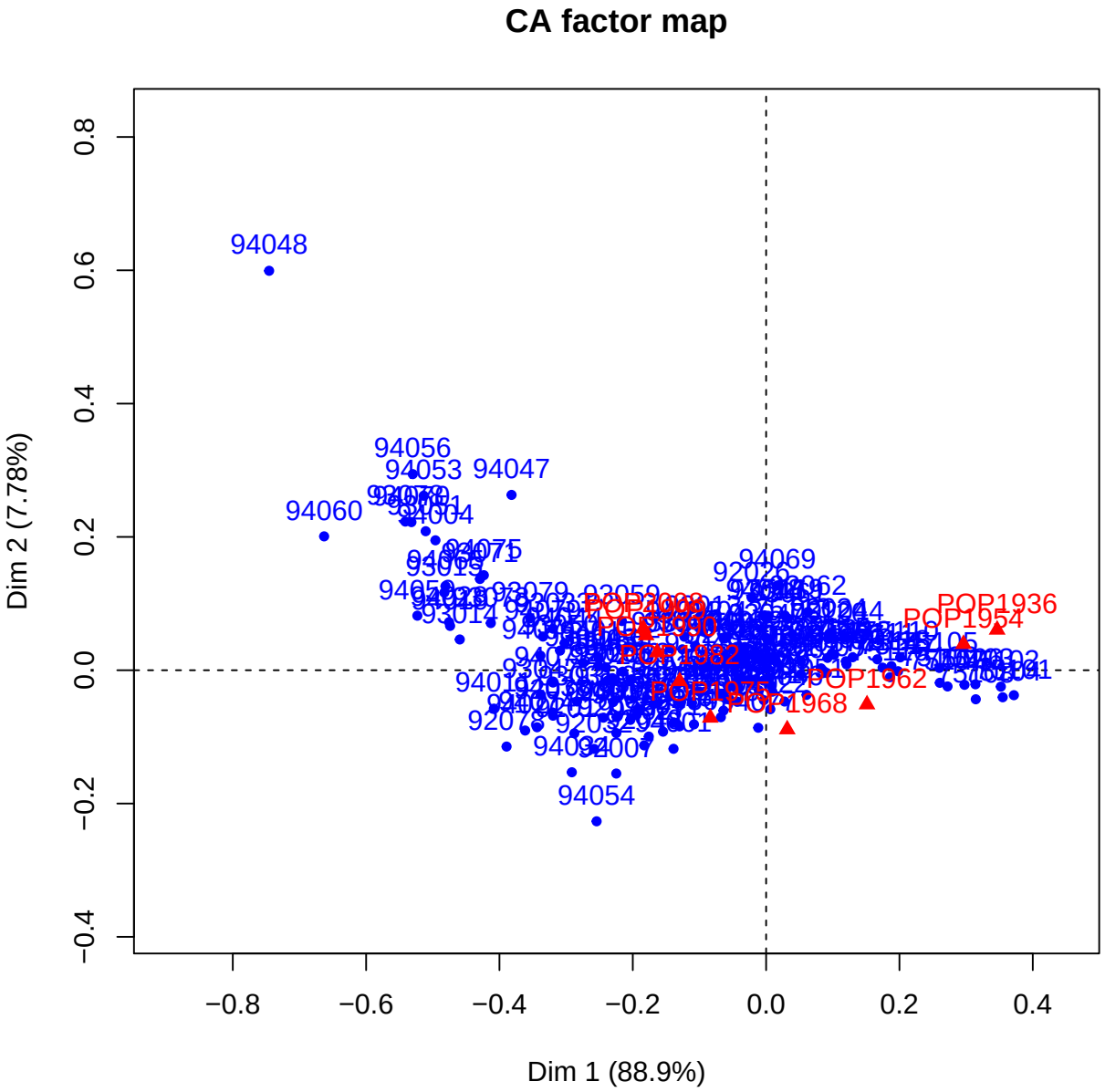
```
##     POP1936          POP1954          POP1962          POP1968
##  Min.   :   276   Min.   :   311   Min.   :   309   Min.   :   437
##  1st Qu.:  6952   1st Qu.:  8916   1st Qu.: 13784   1st Qu.: 16782
##  Median : 18172   Median : 19028   Median : 25792   Median : 28934
##  Mean   : 37140   Mean   : 39027   Mean   : 43571   Mean   : 44918
##  3rd Qu.: 43384   3rd Qu.: 44556   3rd Qu.: 51902   3rd Qu.: 54468
##  Max.   :258599   Max.   :260466   Max.   :254974   Max.   :244080
##     POP1975          POP1982          POP1990          POP1999
##  Min.   :   501   Min.   :  1501   Min.   :  1594   Min.   :  1519
##  1st Qu.: 18203   1st Qu.: 17728   1st Qu.: 18004   1st Qu.: 18105
##  Median : 28727   Median : 28580   Median : 29746   Median : 30080
##  Mean   : 43892   Mean   : 42526   Mean   : 42943   Mean   : 43107
##  3rd Qu.: 52976   3rd Qu.: 50690   3rd Qu.: 48620   3rd Qu.: 50946
##  Max.   :231301   Max.   :225596   Max.   :223940   Max.   :225362
##     POP2008
##  Min.   :  1680
##  1st Qu.: 18905
##  Median : 31854
##  Mean   : 46002
##  3rd Qu.: 54969
##  Max.   :234091
```

On remarque que la population moyenne des communes de la petite couronne et des arrondissements parisiens n'a pas augmenté de façon linéaire entre 1936 et 2008. Trois phases peuvent être définies : on observe une période de croissance allant de 1936 à 1968, puis une période de décroissance entre 1968 et 1982, et, enfin, une nouvelle période de croissance s'étendant 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, lui, décrû et passe 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 démographiquesdes 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 (puis d'interpréter) les proximités statistiques entre modalités d'une variable et individus, ainsi que de comparer les profils-lignes et profils-colonnes entre eux. Il est à 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.

Nous allons maintenant résumer l'information contenue dans le tableau de contingence `dPopEvol` en réalisant l'AFC grâce à la fonction `CA()` du *package* `FactoMineR`. Celle-ci renvoit automatiquement un graphique représentant le nuage de points des individus et des variables sur le plan principal. En outre, elle renseigne sur le résultat du test du χ^2 élaboré sur le tableau de contingence initial et sur son degré de significativité, et renvoit la liste des noms de résultats que l'objet contient.

```
resAFCpop3608 <- CA(dPopEvol)
```



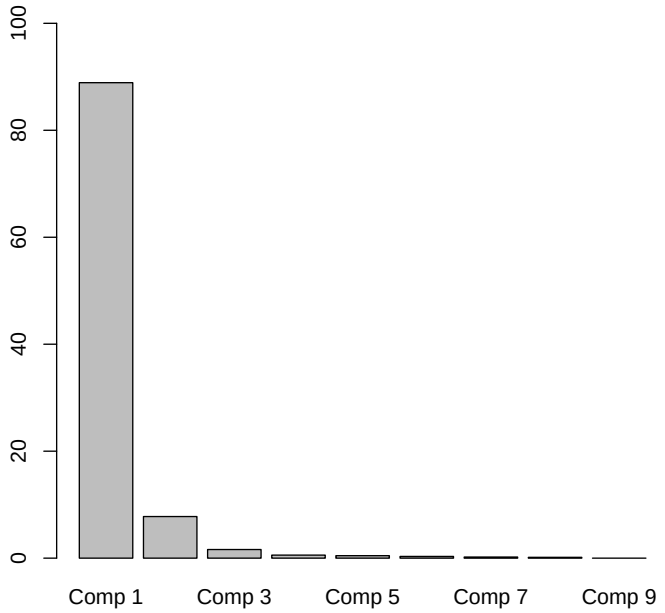

```
list(resAFCpop3608)

## [[1]]
## **Results of the Correspondence Analysis (CA)**
## The row variable has  143  categories; the column variable has 9 categories
## The chi square of independence between the two variables is equal to 2259766 (p-value =  0 ).
## *The results are available in the following objects:
##
##    name              description
## 1  "$eig"            "eigenvalues"
## 2  "$col"            "results for the columns"
## 3  "$col$coord"      "coord. for the columns"
## 4  "$col$cos2"       "cos2 for the columns"
## 5  "$col$contrib"    "contributions of the columns"
## 6  "$row"            "results for the rows"
```

```
## 7  "$row$coord"      "coord. for the rows"
## 8  "$row$cos2"       "cos2 for the rows"
## 9  "$row$contrib"    "contributions of the rows"
## 10 "$call"           "summary called parameters"
## 11 "$call$marge.col" "weights of the columns"
## 12 "$call$marge.row" "weights of the rows"
```

Sélection des axes à analyser

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

```
barplot(resAFCpop3608$eig[, 2], names = paste("Comp", 1:nrow(resAFCpop3608$eig)),
    ylim = c(0, 100))
```



L'AFC rend compte d'une structuration particulièrement marquée des données. En effet, le premier axe factoriel apparaît comme la composante qui explique la plus grande part de la variance totale contenue dans le tableau de contingence (88,90%). Le second facteur résume 7,78% de l'inertie totale du nuage de points. Les valeurs propres des autres facteurs sont peu élevées et indiquent une faible différenciation entre les individus/variables ; 96,68% de l'information étant concentrée sur les deux premiers axes.

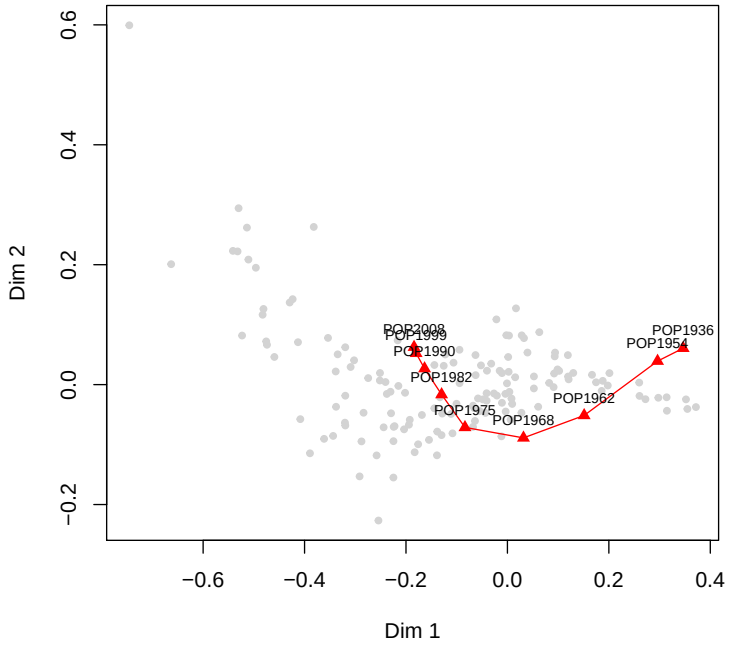
La hiérarchie des facteurs

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 [1].

1. 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* `FactoMineR` mais de représenter avec la fonction `plot()` les coordonnées des communes sur les deux premiers axes (`x=resAFCpop3608$row$coord[,1:2]`) par des cercles (`pch=20`) gris clairs (`col="light grey"`). Les projections des années d'observation sur le plan sont également ajoutées par le biais de l'option `points()`. L'option `(pch=17)` permet de les représenter par des triangles pleins reliés entre eux par une ligne droite (`type="o"`) selon l'ordre des colonnes dans le tableau de contingence initial. Enfin, il est précisé que triangles et lignes sont de couleur rouge (`col="red"`).

```
plot(x = resAFCpop3608$row$coord[, 1:2], col = "light grey", pch = 20, xlab = "Dim 1",
    ylab = "Dim 2")
points(resAFCpop3608$col$coord, type = "o", pch = 17, col = "red")
text(resAFCpop3608$col$coord, labels = row.names(resAFCpop3608$col$coord), cex = 0.7,
    pos = 3)
```



L'observation du premier plan factoriel nous apprend 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 :

```
head(round(cbind(resAFCpop3608$row$coord[, 1:2]), 2))
```

```
##       Dim 1 Dim 2
## 75101  0.37 -0.04
## 75102  0.35 -0.02
## 75103  0.31 -0.02
## 75104  0.35 -0.04
## 75105  0.26  0.00
## 75106  0.30 -0.02
```

Analyse des contributions

Nous pouvons également chercher à connaître la part que les individus et/ou variables prennent dans la dispersion des individus/variables le long de l'axe en observant leurs contributions. La somme des contributions de tous les individus (respectivement variables) est égale à 1 (ou 1000‰). L'importance de la contribution dépend de la masse des individus, deux individus pouvant avoir des coordonnées identiques sur l'axe sans avoir contribué avec la même intensité à sa formation. De la même manière, un individu éloigné de l'axe peut avoir plus contribué à la formation de l'axe qu'un individu situé à proximité.

```
vContrColF1 <- resAFCpop3608$col$contr[, 1]/sum(resAFCpop3608$col$contr[, 1]) *
    1000
summary(vContrColF1)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     3.2    51.0    81.6   111.0   111.0   317.0

vContrColF1

## POP1936 POP1954 POP1962 POP1968 POP1975 POP1982 POP1990 POP1999 POP2008
##  317.48  243.44   71.01    3.20   21.89   50.98   81.59   99.43  110.99
```

```
vContrRowF1 <- resAFCpop3608$row$contr[, 1]/sum(resAFCpop3608$row$contr[, 1]) *
    1000
summary(vContrRowF1)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    0.52    2.99    6.99   10.10   55.10

which(vContrRowF1 > 10)

## 75101 75102 75103 75104 75105 75106 75107 75108 75109 75110 75111 75112
##     1     2     3     4     5     6     7     8     9    10    11    12
## 75114 75115 75116 75117 75118 92002 92063 92078 93005 93014 93031 93050
##    14    15    16    17    18    21    48    56    58    64    69    78
## 93051 93064 93071 93073 93078 94004 94017 94019 94028 94038 94059 94060
##    79    87    90    92    95   100   104   106   109   113   127   128
```

Les entités spatiales ayant fortement contribué à la formation du premier axe sont principalement des arrondissements parisiens, des communes de Seine-Saint-Denis et du Val-de-Marne. Seules trois communes des Haut-de-Seine ont joué un rôle particulier dans le positionnement de l'axe.

Analyse des qualités de représentation

Indépendamment de la contribution d'une variable ou d'un individu à la formation d'un axe, on peut également chercher à connaître les communes et variables les mieux représentées par ces derniers. On examine alors leurs qualités de représentation sur les axes.

La fonction `CA()` renseigne sur la qualité de représentation des variables et individus sur l'ensemble des axes. La qualité de représentation correspond à la mesure de l'angle que le point relié à l'origine fait avec l'axe considéré (il s'agit du cosinus carré de cet angle). Si l'individu/variable est confondu avec l'axe factoriel, alors sa représentation sera maximale. A l'inverse, plus l'angle sera grand, moins bonne sera la qualité de sa représentation sur l'axe. La somme des qualités de représentation d'un individu ou d'une variable sur les différents axes équivaut à 1.

```
resAFCpop3608$col$cos2[, 1]

## POP1936 POP1954 POP1962 POP1968 POP1975 POP1982 POP1990 POP1999 POP2008
## 0.96317 0.97381 0.83333 0.09731 0.51479 0.88751 0.92753 0.90484 0.85789
```

```
summary(resAFCpop3608$row$cos2[, 1])

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0004  0.5500  0.8250  0.6960  0.9410  0.9980

which(resAFCpop3608$row$cos2[, 1] > 0.94)

## 75101 75102 75103 75104 75105 75106 75107 75108 75109 75110 75111 75112
##     1     2     3     4     5     6     7     8     9    10    11    12
## 75114 75116 75117 75118 92020 92063 93005 93014 93031 93047 93049 93050
##    14    16    17    18    28    48    58    64    69    75    77    78
## 93053 93064 93073 93079 94011 94015 94028 94038 94055 94059 94071 94073
##    80    87    92    96   101   102   109   113   124   127   134   135
## 94079
##   141
```

Assez logiquement en vue de l'analyse des coordonnées réalisée en amont, on remarque que les années de début et fin de la période d'observation sont celles pour lesquelles le premier facteur permet de mieux caractériser l'évolution des populations. Les hauts cœfficients associés aux dix-huit premiers arrondissements parisiens ainsi qu'à un grand nombre de communes de Seine-Saint-Denis et du Val-de-Marne indiquent que ces derniers sont particulièrement bien caractérisés par l'opposition formée entre les communes ayant connu une croissance de leur population plus forte en début ou en fin de période.

6.3 Code

```
# Préparation des données
dRef9907 <- read.table("data99_07.csv",sep = ";",dec = ".",quote = "\"",header = TRUE)
dPop3608 <- read.table("pop36_08.csv",sep = ";",dec = ".",quote = "\"",header = TRUE)
dRef9907$CODDEP <- substr(dRef9907$CODGEO, 1, 2)
vCoordPremierArr <- as.numeric(dRef9907[1, 3:4])
dRef9907$DISTCONTINUE <- sqrt((dRef9907$X - vCoordPremierArr[1]) ** 2 +
  (dRef9907$X - vCoordPremierArr[1]) ** 2) / 10
SOC07 <- data.frame(dRef9907$CODGEO, dRef9907[,23:37], row.names=1)

# Package
suppressWarnings(library(FactoMineR))

# ACP, plans et axes factoriels
SOC07$CODDEP <- as.factor(SOC07$CODDEP)
resACPSOC07 <- PCA(SOC07, quanti.sup=15, quali.sup=14, graph=FALSE)
resACPSOC07$eig

# Coordonnées ACP
resACPSOC07$var$coord[,1:2]

# Représentations ACP
resACPSOC07$var$cos2[,1:2]
summary(resACPSOC07$ind$cos2[,1])
which(resACPSOC07$ind$cos2[,1]>0.82)
```

```r
# Contributions ACP
vContrVarF1 <- resACPSOC07$var$contr[,1]/sum(resACPSOC07$var$contr[,1])*1000
summary(vContrVarF1)
vContrVarF1
vContrIndF1 <- resACPSOC07$ind$contr[,1]/sum(resACPSOC07$ind$contr[,1])*1000
summary(vContrIndF1)
which(vContrIndF1>9.24)

# Préparation des données AFC
dPopEvol <- data.frame(dPop3608[,-c(2,12)], row.names=1)
summary(dPopEvol)

# AFC, plans et axes factoriels
resAFCpop3608 <- CA(dPopEvol, graph=FALSE)

# Coordonnées AFC
dCoordAFCind <- round(cbind(resAFCpop3608$row$coord[,1:2]),2)
dCoordAFCvar <- round(cbind(resAFCpop3608$col$coord[,1:2]),2)

# Représentation AFC
resAFCpop3608$col$cos2[,1]
summary(resAFCpop3608$row$cos2[,1])
which(resAFCpop3608$row$cos2[,1]>0.94)

# Contributions AFC
vContrColF1 <- resAFCpop3608$col$contr[,1]/sum(resAFCpop3608$col$contr[,1])*1000
summary(vContrColF1)
vContrColF1

vContrRowF1 <- resAFCpop3608$row$contr[,1]/sum(resAFCpop3608$row$contr[,1])*1000
summary(vContrRowF1)
which(vContrRowF1>10)
```

Chapitre 7

Méthodes de classification

Objectifs

L'objectif de ce chapitre est de mettre en œuvre des classifications et d'identifier les différents éléments et graphiques aidant à l'interprétation. Deux classifications sont proposées sur deux jeux de données différents :

- un tableau de données hétérogènes où les communes sont décrites par la répartition de la population selon 5 catégories socio-professionnelles en %.
- un tableau de contingence croisant la population des communes au cours des différents recensements depuis 1936. C'est un tableau de contingence un peu particulier, croisant les modalités de l'espace et du temps.

Méthodes statistiques requises

Classification ascendante hiérarchique

Packages nécessaires

- `cluster` : préinstallé, pour la première classification
- `ade4` : pour la deuxième classification et l'AFC
- `FactoMineR` : pour la description des classes
- `rgdal` : pour cartographier les résultats

Données

Dans ce chapitre, on utilisera deux tableaux de données produites par l'INSEE :

- Data99_07.csv : Données de référence pour 1999 et 2007 avec différentes variables sur la composition socio-professionnelle des communes de Paris et la petite couronne.
- pop36_08.csv : Données de population sans double compte des recensements de la population de 1936 à 2008 pour Paris et la petite couronne.
- paripc_com_region.* : Délimitations communales en format shapefile nécessaires pour cartographier les classes.

Préparation des données

On travaillera ici avec deux tableaux :

- la table *data99_07* pour présenter la méthode appliquée à des données hétérogènes
- la table *pop36_08* pour illustrer la méthode appliquée à la classification des lignes d'un tableau de contingence

Lecture des deux tableaux et sélection des variables nécessaires :

```
# Chargement des packages
library('cluster')
library('ade4')
library('FactoMineR')
library('rgdal')

# Lecture des fichiers statistiques pour les communes
data99_07 <- read.csv("data/data99_07.csv",
                      sep=";",
                      dec=",",
                      quote = "\"",
                      header = TRUE,
                      encoding = "latin1")
pop36_08 <- read.csv("data/pop36_08.csv",
                      sep=";",
                      dec=",",
                      quote = "\"",
                      header = TRUE,
                      encoding = "latin1")
rownames(data99_07) <- as.character(data99_07$CODGEO)

# Lecture des fichiers géométriques (ou cartographiques)
parisPC <- readOGR("data/paripc_com_region.shp",
                   layer= "paripc_com_region",
                   input_field_name_encoding="latin1")
```

7.1 Rappels méthodologiques

On présente ici rapidement quelques éléments méthodologiques indispensables à la compréhension des entrées/sorties de la classification sous R. On considère un ensemble de n entités $\{i\}$ à classifier, décrites par p variables $\{j\}$. Différents algorithmes de classification existent. La méthode illustrée ici est la classification ascendante hiérarchique (CAH). L'algorithme est itératif. On part d'une partition en n classes, chaque classe étant constituée d'une entité. À chaque étape, on agrège les deux classes les plus « proches ». L'algorithme se déroule en $(n-1)$ étapes et produit une série de $n-1$ partitions, de la plus fine (n classes) à la plus grossière (1 classe). Les paramètres de la méthode sont la définition d'une distance entre entités et d'une distance entre classes (c'est-à-dire s'appuyant sur la distance entre entités les composant). La distance entre classes définit le critère d'agrégation.

Les paramètres de la méthode sont donc :

- une distance d pour mesurer la ressemblance entre 2 objets i et i'.
  On construit ainsi une matrice de distance $n \times n$ de termes génériques $d(i,i')$ où d désigne la distance choisie (distance euclidienne, distance du χ^2 ...).
- une règle permettant d'agréger des classes d'objets ou d'individus définissant la manière de mesurer la distance entre classes d'objets. On note $\delta(k,k')$ la distance entre les classes d'objets C_k et $C_{k'}$, qui s'appelle aussi « **critère d'agrégation** ». Selon cette logique, on réunira les classes les plus proches. Parmi les critères les plus connus, on trouve :

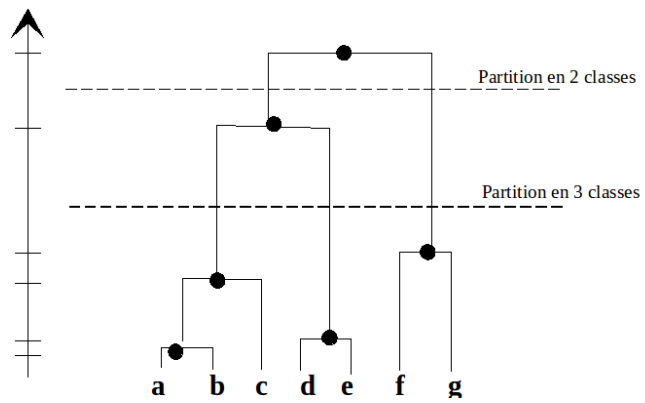
- Distance « min » : $\delta(k,k') = min\ (d(i,i'), i \in C_k, i' \in C_{k'})$
- Distance « max » : $\delta(k,k') = max\ (d(i,i'), i \in C_k, i' \in C_{k'})$
- Distance moyenne :
$$\delta(k,k') = \frac{\sum_{i \in C_k, i' \in C_{k'}} d(i,i')}{n_k \times n_{k'}}$$
- Distance entre centres de gravité : $\delta(k,k') = d(g_k, g_{k'})$

D'autres mesures sont fondées sur la compacité des classes, ou réciproquement leurs bonnes séparations. On présente ici deux critères permettant de choisir entre 2 partitions, celle qui maximisera la compacité des classe ou réciproquement, maximisera la séparabilité entre classes.

- Moyenne des distances à l'intérieur des classes :
$$IC(C) = \frac{\sum_{i',i' \in C_{k'}} d(i,i')}{n(n-1)/2}$$
- Perte d'inertie minimum (critère de Ward) : fondée sur la décomposition de l'inertie totale selon une partition en inertie intra-classes et inertie inter-classes.
$$I_N(P_k) = \sum_{l=1}^{k} \sum_{i \in C_l} m_i d^2(i, g_l) + \sum_{l=1}^{k} m_l d^2(g_l, G)$$
Ici entre 2 partitions, on préférera celle qui minimise l'inertie intra-classes, ce qui revient à maximiser l'inertie inter-classes.

où n est le nombre d'objets à classer, n_k l'effectif de la classe C_k, m_i le poids associé à l'élément i, et g_l le centre de gravité de la classe C_l

L'agorithme se déroule en $n-1$ étapes, et on obtient une hiérarchie de $n-1$ partitions qui se représente sous la forme d'un arbre de classification.



Les points noirs matérialisent les $n-1$ agrégations. Ils sont appelés **nœuds** de l'arbre.

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). La partie supérieure représente la proximité des classes (la distance se lit dans l'arbre). L'axe vertical marque les valeurs du critère d'agrégation à chaque étape de l'algorithme. Ainsi les nœuds, associés aux classes créées, sont positionnés sur cette échelle. La compacité des classes créées se lit au regard de la longueur des branches associées aux nœuds inférieurs.

On coupe ensuite l'arbre, selon un compromis, entre le nombre de classes et la qualité de la partition au sens du critère d'agrégation choisi. Plus on augmente le nombre de classes, plus celles-ci sont compactes au sens du critère choisi ; mais plus il est difficile de les différencier dans l'interprétation et dans la représentation cartographique.

On présente de manière différenciée la classification d'entités décrites par des variables hétérogènes, et la classification de lignes d'un tableau de contingence. La première, comme une analyse en composante principale, utilisera la distance euclidienne, alors que la deuxième, comme l'analyse factorielle des correspondances, nécessite de travailler avec une métrique du χ^2.

7.2 Classification d'entités décrites par des variables quantitatives hétérogènes

On cherche ici à différencier les communes en fonction de leur structure de composition en catégories professionnelles. On sélectionne ainsi les différents pourcentages d'artisans, de cadres, de professions intermédiaires, d'employés, d'ouvriers, calculés sur une même population : la population active.

```
dfCSP <- data99_07[, c("PART07", "PCAD07", "PINT07", "PEMP07", "POUV07")]
```

On travaille sur un tableau centré réduit de manière à donner le même poids à l'ensemble des variables. En conséquence, la moyenne de chaque colonne du tableau vaut 0 et l'écart-type vaut 1.

```
dfCSPStd <- scale(x = dfCSP)
```

La classification se fait avec la fonction `agnes()` (**ag**glomerative **nes**ting) du *package* `cluster`. Le premier argument est le tableau à analyser, on spécifie ici la métrique euclidienne (l'autre option possible est la métrique de Manhattan). Pour calculer la distance entre classes, on choisira la méthode « Ward »qui correspond au critère de la « perte d'inertie minimum »précédemment décrit.

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

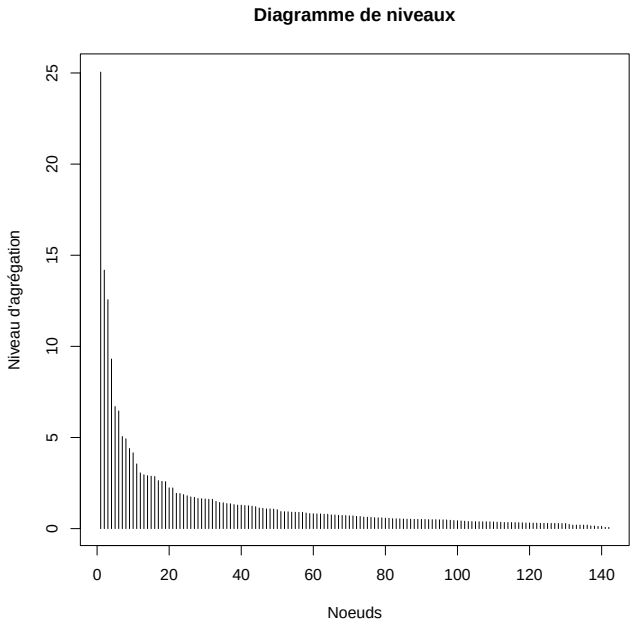
L'objet `cahCSP` est de type *agnes*. Il contient l'ensemble des informations qui vont ensuite servir pour dessiner et analyser l'arbre, puis le couper et donner des aides à l'interprétation.

Tracé des histogrammes des niveaux et du dendrogramme

On commence par transformer le type des sorties en un tableau de type *hclust* qui contient les mêmes éléments mais sous une autre forme. Il est d'usage ensuite de tracer l'histogramme des niveaux d'agrégation supérieurs, c'est-à-dire la valeur associée à la création de chacun des nœuds. Dans notre cas, il s'agit de la part de l'inertie inter-classes passée en inertie intra-classes.

On accède aux niveaux par l'attribut `$height`. Ceux-ci sont rangés par défaut par ordre décroissant. Ainsi le numéro de nœud 1 correspondra à la dernière agrégation, celle qui a conduit à réunir toutes les entités dans une même classe, le n° de nœud 2 à l'avant dernière agrégation, etc ...

```
cahCSP <- as.hclust(cahCSP)
plot(rev(cahCSP$height), type = "h", xlab = "Noeuds", ylab = "Niveau d'agrégation",
    main = "Diagramme de niveaux")
```

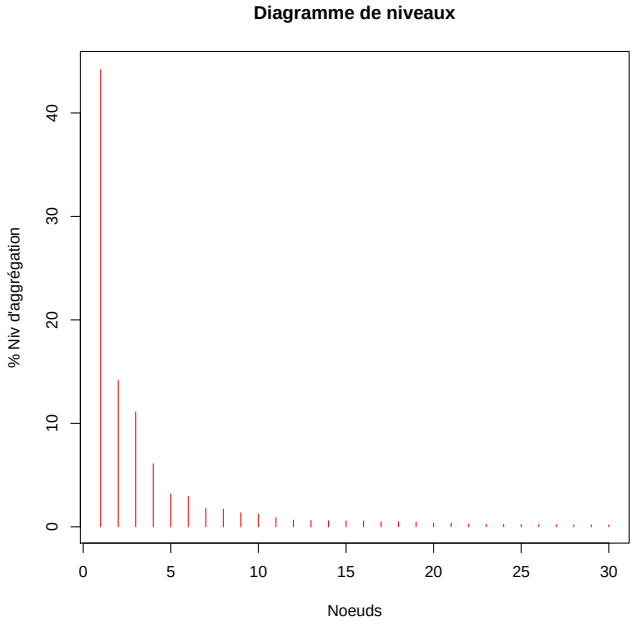


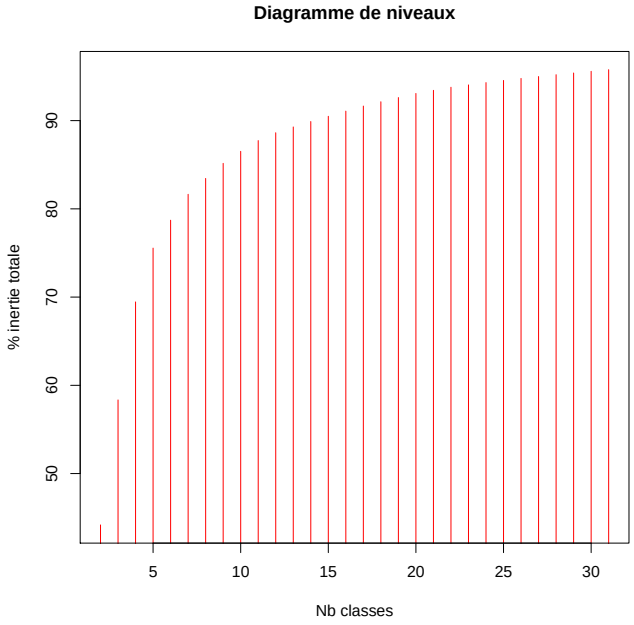
Un tel graphique permet d'ores et déjà de repérer les « sauts » dans l'histogramme et de décider à quel niveau on peut couper l'arbre. Les niveaux sont exprimés ici en absolu.

Lorsque l'on utilise ce critère d'agrégation, il est d'usage de les exprimer en part de l'inertie totale. L'inertie totale se calcule comme la somme des niveaux. On rapporte donc chaque niveau à la somme des niveaux.

En cumulant ce taux de proche en proche, on obtient la part de l'inertie restituée par la partition correspondant au « numéro du nœud + 1 » classes. En général, on ne représente que le haut de la hiérarchie : ici les 30 derniers nœuds.

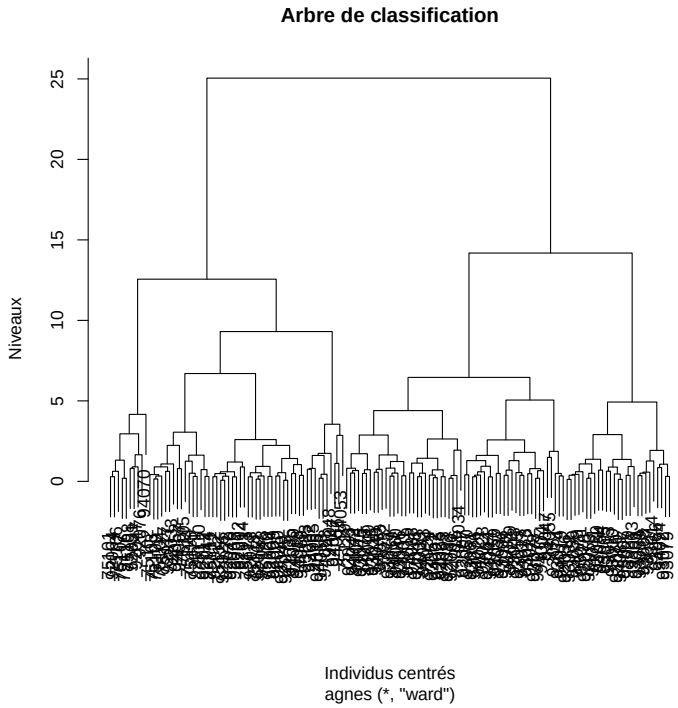
```
h2 <- as.data.frame(sort(cahCSP$height^2, decreasing = T))
colnames(h2) <- "height"
h2$tau <- h2$height * 100/sum(h2$height)
h2 <- h2[1:30, ]
plot(h2$tau, type = "h", col = "red", xlab = "Noeuds", ylab = "% Niv d'aggrégation",
    main = "Diagramme de niveaux")

h2$taucum <- cumsum(h2$tau)
h2$nbcla <- as.numeric(row.names(h2)) + 1
plot(h2$nbcla, h2$taucum, type = "h", col = "red", xlab = "Nb classes", ylab = "% inertie totale",
    main = "Diagramme de niveaux")
```





On lit ainsi à la fois les sauts (graphique de gauche) et la part de l'inertie inter-classes associée à la partition sous la coupure (graphique de droite). Une coupure en 5 classes donne une partition restituant presque 75% de l'information initiale (inertie inter / inertie totale en %). L'analyse de l'arbre confirme qu'une coupure en 3 ou 5 classes est possible.

```
plot(cahCSP, xlab = "Individus centrés", ylab = "Niveaux", main = "Arbre de classification")
```



Coupure de l'arbre et description des classes

L'objet obtenu avec la fonction `agnes()` contient l'ensemble de l'arbre ; grâce à la fonction `cutree()`, il est possible de « couper » l'arbre en énonçant le nombre de classes souhaitées (`k`). On obtient une nouvelle variable donnant pour chaque entité son numéro de classe. On peut ensuite intégrer cette variable au tableau initial pour calculer des statistiques par classe et ainsi caractériser les classes.

```
classesCAH <- cutree(tree = cahCSP, k = 5)
dfCSP$typo5Cl <- as.factor(classesCAH)
```

On décrit ensuite les classes par rapport à leur spécificité relativement au profil moyen de la région (ici les valeurs moyennes pour chacune des variables). La fonction `catdes()` du *package* `FactoMineR` permet de calculer un certain nombre « d'aides à l'interprétation »qui viennent aider à caractériser les classes.

En premier lieu, on sélectionne les variables pour lesquelles des sur-représentations ou sous-représentations significatives sont observées.

Pour cela, on teste pour chaque classe si pour chaque variable la moyenne de la classe est significativement différente de la moyenne générale. À cet effet, on calcule la *V-test*, qui est la valeur signée de cet écart réduit, et l'on teste si cet « écart-réduit » est significativement différent de 0. Ainsi pour un risque 0.05 (proba=0.05), si la *V-test* dépasse 2 (1.96) en valeur absolue, on conclura qu'il y a sur- ou sous-représentation significative de la variable concernée dans cette classe.

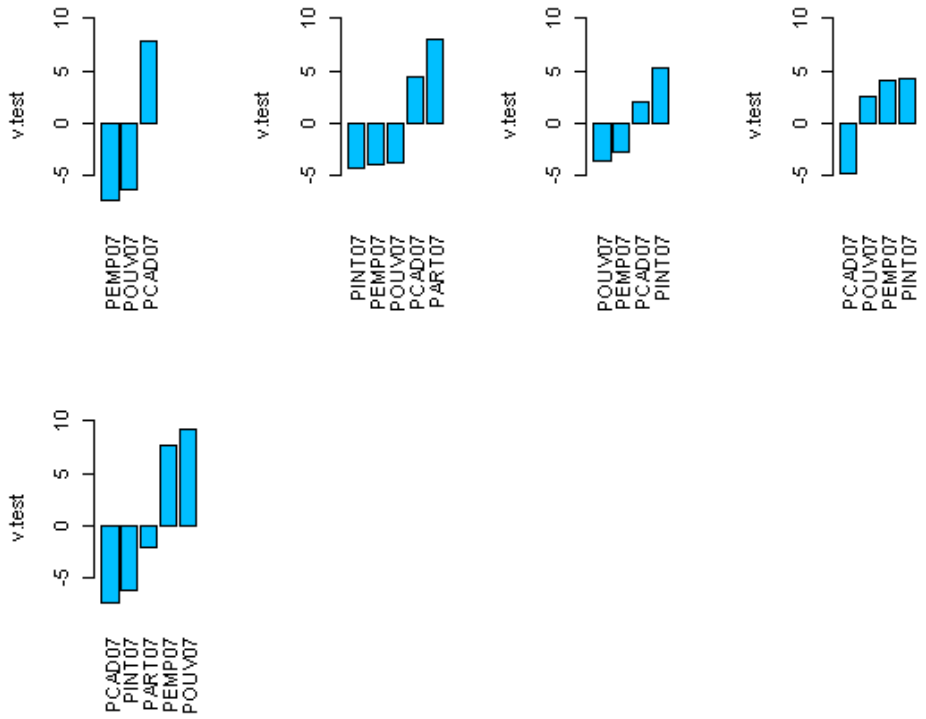
La fonction `catdes()` produit un objet récapitulant l'ensemble des calculs nécessaires à la caractérisation des classes par les variables, organisées selon la *V-test*. Le seuil de significativité est défini par défaut (`proba=0.05`, et la procédure sélectionne par classe les variables spécifiques à ce risque.

Dans un premier temps on propose de ne pas « sélectionner » les variables en fonction de la *V-test*, (`proba=1`) de manière à identifier l'ensemble des statistiques calculées, à savoir pour chaque variable : la moyenne et l'écart-type de la classe, que l'on peut comparer à la moyenne et l'écart-type de l'ensemble, la *V-test* et sa significativité (*p-value*).

```
descClasses <- catdes(donnee = dfCSP, num.var = 6, proba = 1)
descClasses$quanti
```

En relançant le calcul avec `proba=0.05`, seules sont conservées les variables qui participent significativement à la spécialisation de la classe. On peut ensuite représenter la distribution des valeurs de V-test (significativité des sur- ou sous-représentations) pour l'ensemble des variables par des histogrammes pour chaque classe. Les variables sont triées allant de la sur-représentation à la sous-représentation.

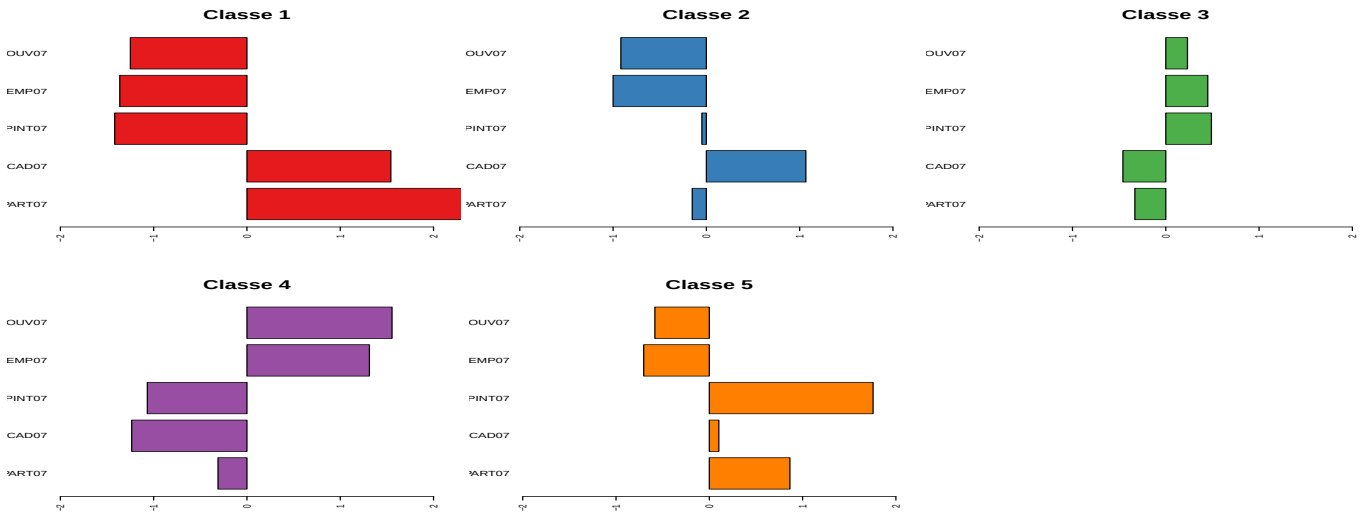
```
descClasses <- catdes(donnee = dfCSP, num.var = 6, proba = 0.05)
plot.catdes(x = descClasses)
```



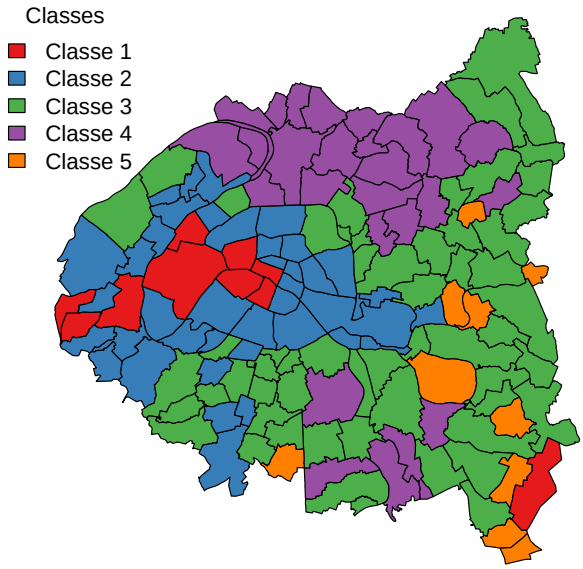
Les profils de classe et la cartographie sont produits séparément, le lien entre les deux se faisant par les couleurs associées à chacune des cartes. On peut ainsi simultanément avoir les distributions statistiques (en écart réduit au profil moyen) et les distributions spatiales. Pour plus de précision sur la cartographie, se référer au Chapitre 9.

```
# Agrégation des données centrées réduites par classe
dfCSPStd2 <- cbind.data.frame(dfCSPStd, classesCAH)
dfCSPStdcla <- aggregate(dfCSPStd2[,1:5],
                         by=list(dfCSPStd2$classesCAH), mean)

rownames(dfCSPStdcla) <- paste("CL",dfCSPStdcla$Group.1,sep="")
# Options graphiques et de palettes
library(RColorBrewer)
vPal5 <- brewer.pal(n = 5, name = "Set1")
ncla <- nrow(dfCSPStdcla)
# Transformation en matrice et représentation histogrammes par classe
CSPStdcla <- as.matrix(t(dfCSPStdcla[-1]))
for(i in 1:ncla) {
    barplot(CSPStdcla[,i],
            beside = TRUE,
            horiz = TRUE,
            col = vPal5[i],
            names.arg = rownames(CSPStdcla),
            xlim=c(-2,2),
            cex.names=1.2,
            las=2
```

```
            )
    title(paste("Classe", i, sep=" "), cex.main=2)
}
```




```
dfCSPMerge <- dfCSP
dfCSPMerge$CODCOM <- row.names(dfCSP)
parisPC@data <- merge(x = parisPC@data,
                      y = dfCSPMerge,
                      by.x = "DEPCOM",
                      by.y = "CODCOM")

parisPC@data$ClusterColors <- as.character(vPal5[parisPC@data$typo5Cl])
plot(parisPC, col=parisPC@data$ClusterColors)
legend("topleft",
    legend = paste("Classe", unique(as.character(parisPC@data$typo5Cl)), sep=" "),
    bty = "n",
    fill = vPal5,
    cex = 1.2,
    title = "Classes")
```



7.3 Classification des lignes d'un tableau de contingence

Un tableau de contingence est le résultat du croisement de deux variables qualitatives décrivent des entités élémentaires (par exemple le tableau dénombrant la population par commune et par CS provient du croisement des 2 variables qualitatives « commune de résidence » et « catégorie socio-professionnelle » au niveau individuel).

Pour éviter que seules les différences d'effectifs ne dirigent la classification, la classification des lignes d'un tableau de contingence nécessite de faire des choix différents de distance. Comme pour l'analyse factorielle des tableaux de contingence, la métrique utilisée doit être la métrique du χ^2.

Pour ce cas de figure, on trouve dans R plusieurs approches qui permettent d'utiliser des procédures génériques. La classification se fera soit sur les coordonnées factorielles issues d'une analyse multivariée (ici AFC), soit sur une matrice de distances du χ^2. Cependant, aucune de ces procédures ne correspond exactement à celles utilisées classiquement dans le cadre de l'analyse des données « à la française » [Sanders 1989, Lebart *at al.* 1995] [1].

On a vu dans le chapître précédent qu'une analyse factorielle produit une description des entités dans un nouveau système de coordonnées : celui des facteurs. Ainsi, décrire globalement les entités par les variables ou par les facteurs est équivalent lorsque l'on retient l'ensemble des facteurs, Plus spécifiquement, opérer une classification sur un tableau de contingence en utilisant la distance du χ^2 revient à faire la même classification sur l'ensemble des facteurs d'une analyse des correspondances factorielles en utilisant la distance euclidienne. On peut choisir de garder les premières coordonnées factorielles, ce qui revient à éliminer le « bruit » associé aux derniers facteurs qui sont souvent très spécifiques.

Selon les algorithmes, on peut choisir de travailler sur une matrice de coordonnées ou sur une matrice de distance. Certains algorithmes permettent de faire l'un ou l'autre.

On récapitule dans un premier temps les trois différentes manière de procéder, en fonction des *packages* utilisés. Ces trois procédures donnent des résultats identiques :

1. Procédure `hclust()` du *package* `stats`

1. Un certain nombre de tests ont été réalisés pour comparer les résultats obtenus avec d'autres logiciels (SAS, SPAD). Avec les fonctions courantes de R, le transfert du poids des entités lignes n'est pas possible, ainsi les entités sont classées ici sans que leur poids n'intervienne dans le calcul des inerties. **A la date de sortie du manuel, nous avons trouvé la procédure `ward.cluster()` du *package* `FactoClass` qui permet d'appliquer cette méthode. Nous détaillerons son utilisation dans une édition ultérieure.**

2. Procédure `agnes()` du *package* `cluster`
3. Procédure `HCPC()` du *package* `FactoMineR`

Les trois méthodes diffèrent soit par les *inputs* soit par les *outputs*. On les présente toutes les trois car elles présentent chacune un intérêt. La première (`hclust()`) ne travaille que sur une matrice de distance, la seconde travaille indifféremment sur un tableau de données ou une matrice de distance. Enfin, la dernière travaille sur des coordonnées factorielles. Les *outputs* des deux premières sont similaires. La dernière présente des aides à l'interprétation des classes intéressantes. On examinera ensuite plus attentivement la sortie de la procédure `HCPC()` qui génère automatiquement un ensemble d'aides à la description des classes. On la complétera par des représentations graphiques.

Le tableau de contingence analysé est un peu spécifique. C'est la table croisant les modalités (mailles) d'un découpage de l'espace (les communes) avec les modalités du temps (9 années de recensement). Le décompte porte sur la population résidente. Travailler sur ce tableau revient à classifier les trajectoires communales de population.

```
# Mise en forme des données
dfPopsPC <- pop36_08[, c("POP1936", "POP1954", "POP1962", "POP1968", "POP1975",
    "POP1982", "POP1990", "POP1999", "POP2008")]
row.names(dfPopsPC) <- pop36_08$CODGEO
```

L'ensemble des traitements présentés peuvent être adaptés à tout autre tableau de contingence, la spécificité de cet exemple étant le caractère ordinal des modalités de la variable temps.

7.3.1 Classification avec `hclust()`

La procédure `hclust()` travaille uniquement sur une matrice de distance. L'enchaînement qui est proposé ici intègre les fonctions du *package* `ade4` :

1. Analyse factorielle des correspondances (`dudi.coa()`)
2. Calcul d'une matrice de distance du χ^2 entre les entités (`dist.dudi()`) à partir des facteurs de l'AFC.
3. Classification des entités.
   Pour utiliser cette procédure, on construit d'abord la matrice de distance. La procédure s'applique ensuite sur cette matrice.

Pour l'analyse factorielle des correspondances (`dudi.coa()`), on garde l'ensemble des facteurs (`nf`). En AFC si l'on a n modalités, on aura $n-1$ facteurs. Pour gagner du temps, on indique ici un nombre de facteurs à conserver supérieur à leur quantité réelle, ce qui indique indirectement à la procédure de garder l'ensemble des facteurs.

```
resafc <- dudi.coa(df = dfPopsPC, scannf = FALSE, nf = ncol(dfPopsPC))
```

La deuxième opération consiste à calculer, à partir du fichier produit, la matrice de distance du χ^2 appropriée.

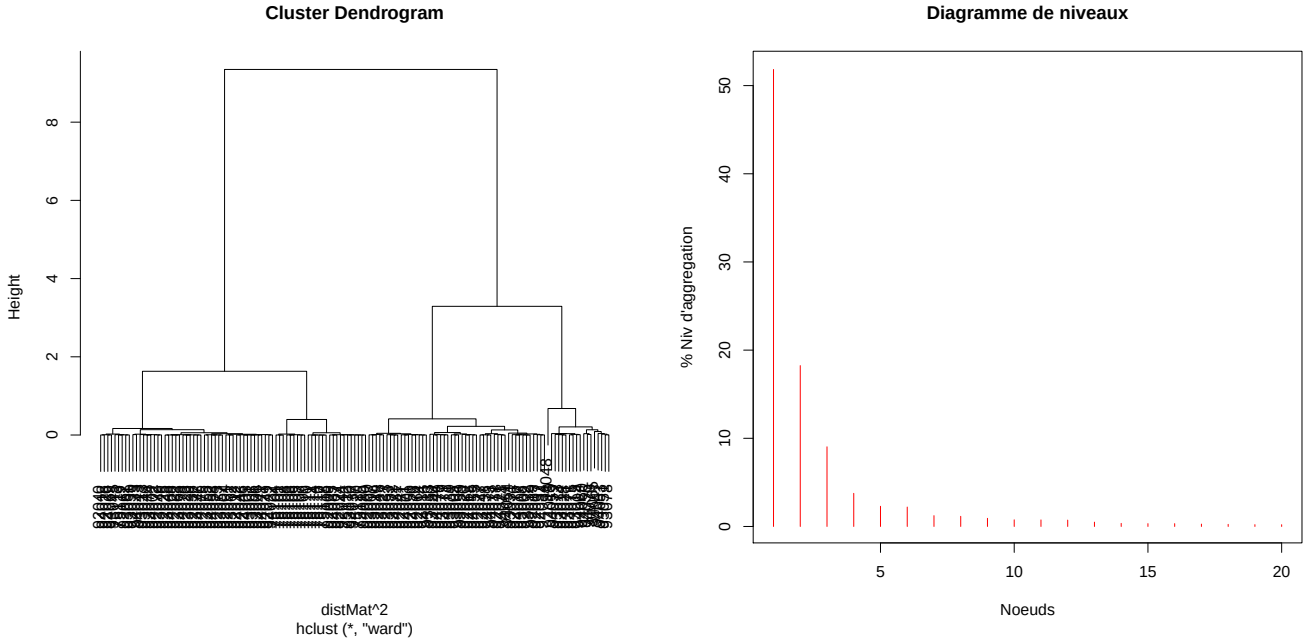
```
distMat <- dist.dudi(dudi = resafc, amongrow = TRUE)
```

La classification peut alors être effectuée à partir de cette matrice. La procédure `hclust()` travaille sur la matrice élevée au carré. Les calculs qui suivent ont déjà été présentés dans les pages précédentes, mais ici, les niveaux ne seront donc pas élevés au carré comme précédemment. L'instruction `plot()` appliquée au produit de `hclust()` permet de tracer l'ensemble de l'arbre.

```
resCAH <- hclust(distMat^2, method = "ward")
plot(resCAH)
# Création de l'histogramme des parts d'inertie inter-classes perdues à
# chaque étape
```

```
histo <- as.data.frame(sort(resCAH$height, decreasing = T))
colnames(histo) <- "height"
histo$tau <- histo$height * 100/sum(histo$height)
# Cumul des parts
histo$taucum <- cumsum(histo$tau)
plot(histo$tau[1:20], type = "h", col = "red", xlab = "Noeuds", ylab = "% Niv d'aggregation",
    main = "Diagramme de niveaux")
head(histo)
```

```
##   height    tau taucum
## 1 9.3479 51.832  51.83
## 2 3.2906 18.246  70.08
## 3 1.6297  9.036  79.11
## 4 0.6777  3.758  82.87
## 5 0.4132  2.291  85.16
## 6 0.3973  2.203  87.37
```



L'histogramme des niveaux, i.e. des parts d'inertie inter-classes transférées, permet de choisir une partition en 4 classes. Le *listing* des parts que représentent le transfert d'inertie inter-classes en inertie intra-classes vient confirmer cela.

On observe un saut entre les valeurs associées aux nœud 3 et nœud 4. On peut lire que la différenciation entre les quatre classes représente 82.9 % de la différenciation totale entre les communes.

On stocke donc dans la table de départ la partition en 4 classes.

```
dfPopsPC$res4cla <- as.factor(cutree(resCAH, k = 4))
```

7.3.2 Classification avec `agnes()`

La procédure `agnes()`, utilisée auparavant sur un tableau de données hétérogènes (nécessitant de travailler avec la distance euclidienne), est appliquée ici à un tableau de coordonnées factorielles puis à un tableau de distances.

Classification des entités directement selon leurs coordonnées factorielles : on réutilise l'AFC créée avec la fonction `dudi.coa()` de `ade4`. Les coordonnées factorielles sont contenues dans le vecteur **$li**. La fonction

travaillera directement avec la métrique euclidienne. On applique ensuite à l'objet de type *agnes* les mêmes calculs que précédemment pour construire l'histogramme des niveaux d'agrégation.

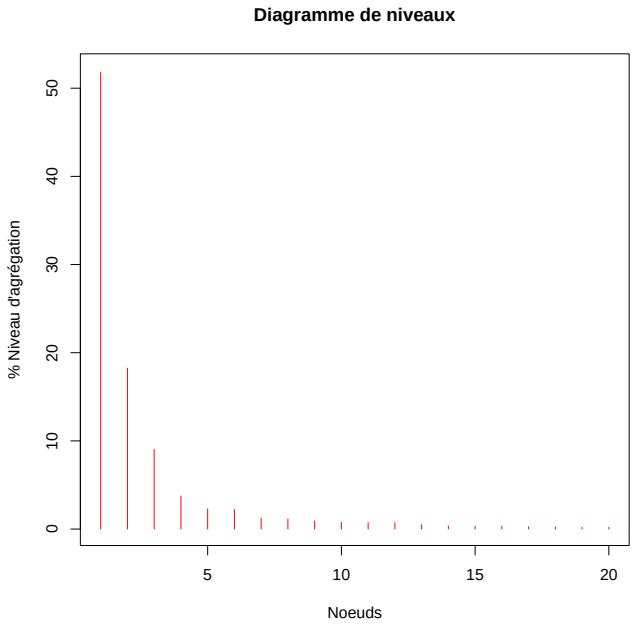
```
resCAH <- agnes(resafc$li, method = "ward")
```

Classification des entités à partir d'une matrice de distance : on propose une alternative au calcul de distance présenté à la Section subsec :hclust qui utilisait la fonction `dist.dudi()` travaillant sur un objet de type *dudi* spécifique d'`ade4`. On peut aussi calculer directement une matrice de distance euclidienne.

```
distMat2 <- dist(x = resafc$li, method = "euclidean")
resCAH <- agnes(distMat2, diss = TRUE, method = "ward")
```

Comme précédemment, on construit l'histogramme des pertes d'inertie pour évaluer le nombre de classes à garder.

```
# Histogramme des pertes d'inertie
histo <- as.data.frame(sort(resCAH$height^2, decreasing = T))
colnames(histo) <- "height"
histo$tau <- histo$height * 100/sum(histo$height)
histo$taucum <- cumsum(histo$tau)
plot(histo$tau[1:20], type = "h", col = "red", xlab = "Noeuds", ylab = "% Niveau d'agrégation",
    main = "Diagramme de niveaux")
```



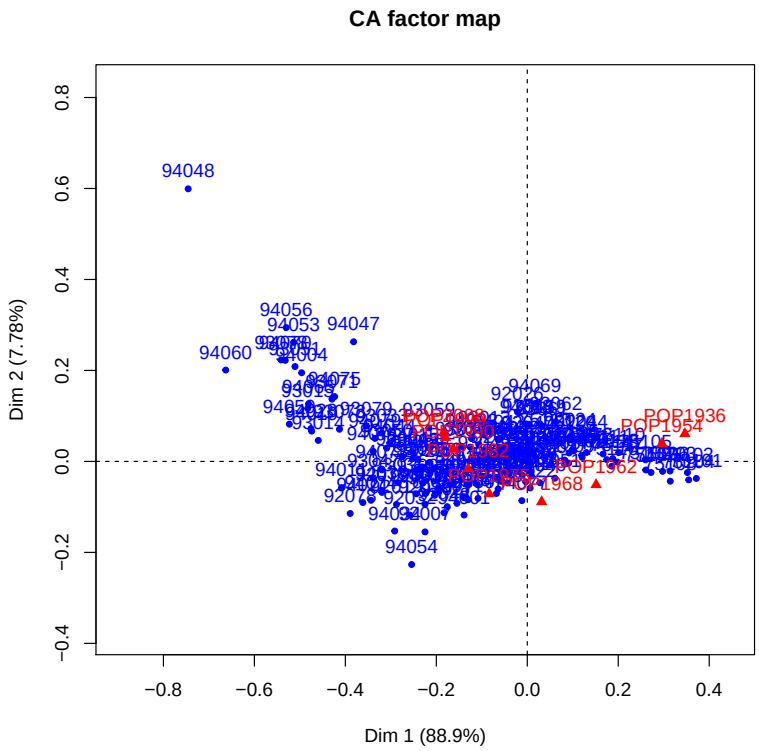
On coupe l'arbre avec la fonction `cutree()` et on stocke dans la table de départ, comme précédemment, la partition en 4 classes.

```
dfPopsPC$res4cla <- as.factor(cutree(resCAH, k = 4))
```

7.3.3 La procédure `HCPC()`

La procédure `HCPC()` travaille sur des coordonnées factorielles. Il est nécessaire de réaliser au préalable une analyse factorielle des correspondances (cf. Chapitre 6). On utilisera ici la fonction `CA()`.

```
resafc <- CA(dfPopsPC[1:9], ncp = 10)
```



Contrairement aux exemples précédents, la procédure de classification intègre directement la coupure de l'arbre en donnant le nombre de classes (`nb.clust`). Cette procédure peut être interactive : si `nb.clust=0`, alors l'utilisateur peut cliquer sur le graphique au niveau où il souhaite couper l'arbre.

```
resCAH <- HCPC(resafc, nb.clust = 0, graph = TRUE, consol = FALSE)
```

On peut aussi éviter l'étape d'interactivité en définissant *a priori* le nombre de classes.

```
resCAH <- HCPC(resafc, nb.clust = 4, graph = F, consol = FALSE)
```

Quand l'option `consol` vaut `TRUE`, la partition est consolidée avec un algorithme de type *k-means* [Lebart *at al.* 1995].

L'objet produit avec la fonction `HCPC()` contient la description des classes avec les mêmes éléments que la fonction `catdesc()`. On les atteint avec les commandes suivantes :

```
resCAH$desc.var
resCAH$desc.ind
resCAH$desc.axes
```

On renvoie au paragraphe précédent pour le détail de ces sorties, et on peut sauver de la même manière la partition :

```
resclust <- resCAH$data.clust
resclust$res4cla <- resclust$clust
dfPopsPC <- merge(dfPopsPC, resclust[2], by = "row.names")
```

On propose dans le cadre de cet exemple un peu spécifique des sorties appropriées, à savoir des trajectoires de population par classe.

7.3.4 Description des classes

Le tableau de contingence étant ici un peu spécifique puisqu'il s'agit de « trajectoires de population », on propose ici un développement approprié permettant de représenter ces trajectoires par classe.

Il s'agit d'abord de constituer un tableau agrégé donnant la somme des populations des communes appartenant à chacune des classes. On pourra alors représenter les graphiques de l'évolution du poids des classes, dans l'absolu en premier lieu (poids total des classes à chaque pas de temps), puis en pourcentage, i.e. rapportée à la population totale du système.

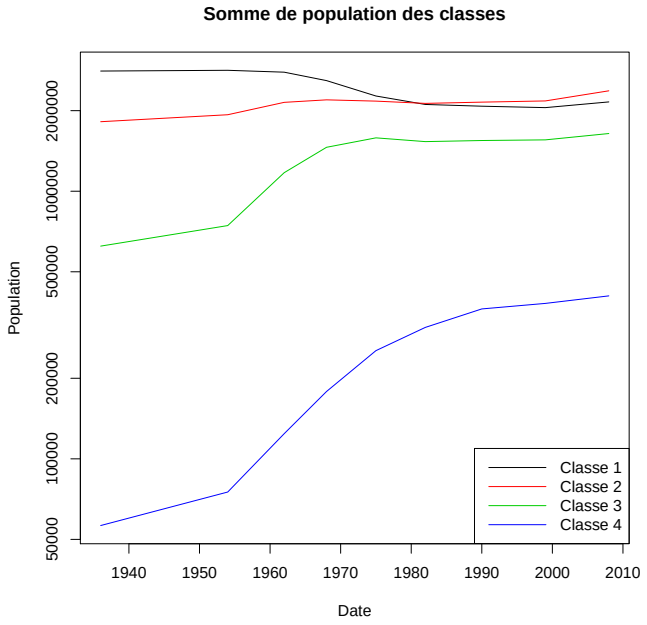
On commence par créer le tableau de données.

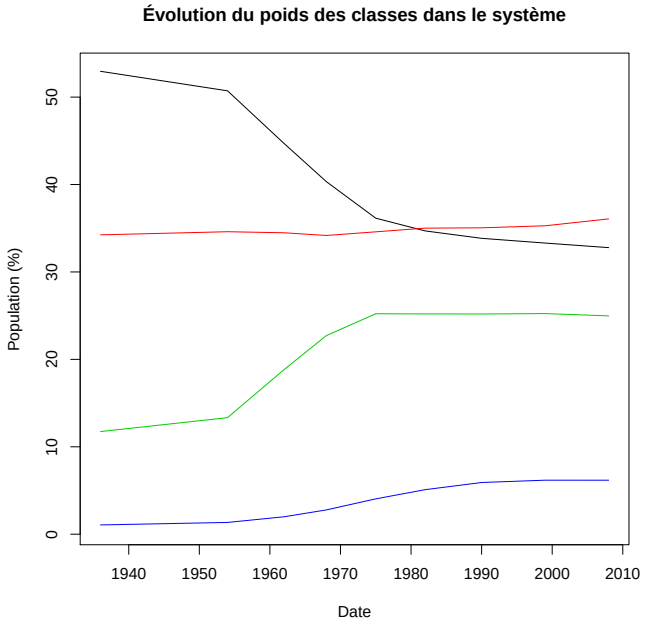
```
# On crée d'abord un objet de classe by
row.names(dfPopsPC) <- dfPopsPC$Row.names
dfPopsPC$Classe <- as.integer(as.character(dfPopsPC$res4cla))
dfPopsPC <- dfPopsPC[-c(1,11:12)]
colnames(dfPopsPC) <- sub(pattern="POP", replacement="", x=colnames(dfPopsPC))

bySumPopsPC <- by(data=dfPopsPC[1:9],
                  INDICES=dfPopsPC$Classe, FUN=colSums,
                  simplify=TRUE)
# Puis on en fait un data.frame
dfSumPopsPC <- do.call(rbind.data.frame, bySumPopsPC)
colnames(dfSumPopsPC) <- colnames(dfPopsPC[-10])
dfSumPopsPC

##      1936    1954    1962    1968    1975    1982    1990    1999    2008
## 1 2812295 2830562 2785141 2590428 2268985 2109822 2078397 2052946 2156372
## 2 1818758 1931103 2148358 2195107 2171099 2129108 2152315 2174760 2372654
## 3  623581  744056 1172923 1459288 1582830 1532476 1546906 1555673 1642877
## 4   56330   75153  124160  178492  253686  309832  363198  380859  406355
```

On peut désormais afficher les graphiques correspondants.

```
# Pour le poids total des classes
options(scipen=5)
plot(y=dfSumPopsPC[1,], x=colnames(dfSumPopsPC),
     type='l', log="y", col=1,
     ylim=c(min(dfSumPopsPC),max(dfSumPopsPC)),
     xlab="Date", ylab="Population")
lines(y=dfSumPopsPC[2,], x=colnames(dfSumPopsPC),
      type='l', col=2)
lines(y=dfSumPopsPC[3,], x=colnames(dfSumPopsPC),
      type='l', col=3)
lines(y=dfSumPopsPC[4,], x=colnames(dfSumPopsPC),
      type='l', col=4)
legend(x="bottomright", paste("Classe", 1:4, sep=" "),
       cex=1, seg.len=4, col=1:4, lty=1 )
title("Somme de population des classes")


# Puis pour le poids de chaque classe au sein de la population du système
plot(y=dfSumPopsPC[1,]/colSums(dfSumPopsPC)*100,
     x=colnames(dfSumPopsPC), type='l', col=1,
     ylim=c(min(dfSumPopsPC/colSums(dfSumPopsPC)*100),
            max(dfSumPopsPC/colSums(dfSumPopsPC)*100)),
     xlab="Date", ylab="Population (%)")
lines(y=dfSumPopsPC[2,]/colSums(dfSumPopsPC)*100,
      x=colnames(dfSumPopsPC), type='l', col=2)
lines(y=dfSumPopsPC[3,]/colSums(dfSumPopsPC)*100,
```

```
      x=colnames(dfSumPopsPC), type='l', col=3)
lines(y=dfSumPopsPC[4,]/colSums(dfSumPopsPC)*100,
      x=colnames(dfSumPopsPC), type='l', col=4)
title("Évolution du poids des classes dans le système")
```





On peut alors s'intéresser aux moyennes de classes, en les représentant, en premier lieu, en absolu, et dans un second temps, par le biais de leurs écarts à la moyenne générale de population, afin de les replacer dans le système étudié.

On crée les tableaux correspondants à ces indicateurs :

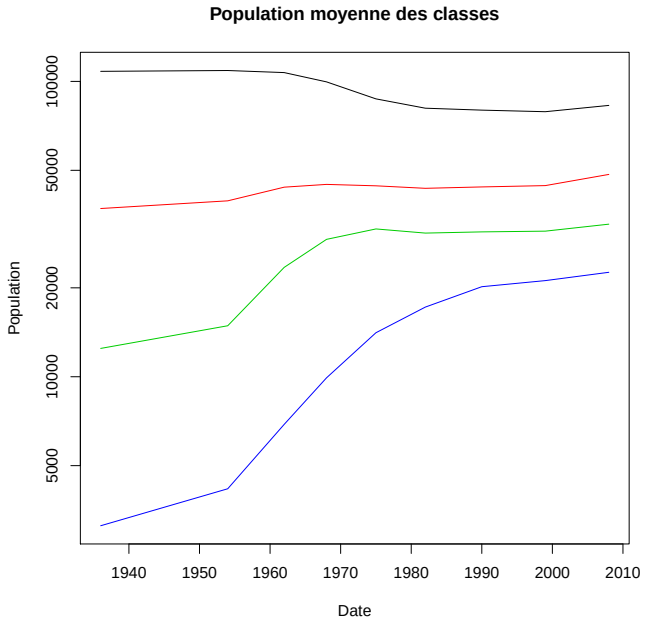
```
# Pour les populations moyennes de classes
byMeanPopsPC <- by(data=dfPopsPC[1:9],
                   INDICES=dfPopsPC$Classe, FUN=colMeans, simplify=TRUE)
dfMeanPopsPC <- do.call(rbind.data.frame, byMeanPopsPC)
colnames(dfMeanPopsPC) <- colnames(dfPopsPC[-10])
# Puis, pour chaque classe, les écarts de ces moyennes à
# la population moyenne d'une ville du système
dfDevPopsPC <- dfMeanPopsPC - colMeans(dfPopsPC[1:9])
dfDevPopsPC
```

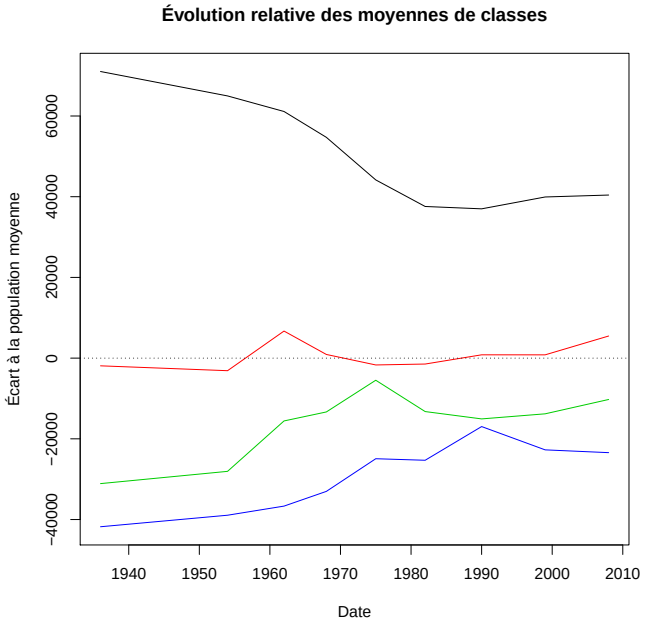
```
##      1936   1954   1962     1968   1975   1982     1990     1999   2008
## 1   71026  64975  61119  54713.6  44162  37576  36995.6  39932.4  40411
## 2   -1910  -3116   6704    905.8  -1694  -1467    818.2    812.4   5479
## 3 -31099 -28062 -15569 -13340.4  -5483 -13243 -15063.7 -13804.8 -10249
## 4 -41789 -38931 -36673 -33026.5 -24933 -25313 -16961.9 -22733.5 -23427
```

Les graphiques représentant le contenu de ces tableaux peuvent désormais être affichés. L'axe des ordonnées du premier suit une échelle logarithmique (base 10), et on représente sur le second l'écart nul ($\overline{Pop}_{k_t} = \overline{Pop}_t$) en pointillés.

```
options(scipen=5)
plot(y=dfMeanPopsPC[1,], x=colnames(dfMeanPopsPC),
     type='l', col=1, log="y",
     ylim=c(min(dfMeanPopsPC), max(dfMeanPopsPC)),
     xlab="Date", ylab="Population")
lines(y=dfMeanPopsPC[2,], x=colnames(dfMeanPopsPC),
```

```
      type='l', col=2)
lines(y=dfMeanPopsPC[3,], x=colnames(dfMeanPopsPC),
      type='l', col=3)
lines(y=dfMeanPopsPC[4,], x=colnames(dfMeanPopsPC),
      type='l', col=4)
title("Population moyenne des classes")
plot(y=dfDevPopsPC[1,], x=colnames(dfDevPopsPC),
     type='l', col=1,
     ylim=c(min(dfDevPopsPC),max(dfDevPopsPC)),
     xlab="Date", ylab="Écart à la population moyenne")
lines(y=dfDevPopsPC[2,], x=colnames(dfDevPopsPC),
      type='l', col=2)
lines(y=dfDevPopsPC[3,], x=colnames(dfDevPopsPC),
      type='l', col=3)
lines(y=dfDevPopsPC[4,], x=colnames(dfDevPopsPC),
      type='l', col=4)
abline(h = 0, lty = "dotted")
title("Évolution relative des moyennes de classes")
```





Bibliographie

Sanders Lena, 1990, L'analyse statistique des données en géographie. Montpellier, RECLUS (Coll. « Alidade »), 267 p.

Lebart L., Morineau A., Piron M. (1995) Statistique exploratoire multidimensionnelle, Dunod.

Husson F., Lê S., Pagès J., 2009, Analyse des données avec R, Dunod.

Nakache J-P., Confais J., 2004, Approche pragmatique de la classification, Technip.

7.4 Code

```
##### Chargement des packages #####
library('cluster')
library('ade4')
library('FactoMineR')
library('rgdal')
library('RColorBrewer')

###### Chargement des données #####
# Lecture des fichiers statistiques pour les communes.
data99_07 <- read.csv("data/data99_07.csv",
                      sep=";",
                      dec=",",
                      quote = "\"",
                      header = TRUE,
                      encoding = "latin1")
pop36_08 <- read.csv("data/pop36_08.csv",
                      sep=";",
                      dec=",",
                      quote = "\"",
                      header = TRUE,
                      encoding = "latin1")
rownames(data99_07) <- as.character(data99_07$CODGEO)

# Lecture des fichiers géométriques (ou cartographique)
parisPC <- readOGR("data/paripc_com_region.shp",
                   layer= "paripc_com_region",
                   input_field_name_encoding="latin1")

##### Classifications (distance euclidienne) #####

# Standardisation
dfCSP <- data99_07[, c("PART07", "PCAD07", "PINT07", "PEMP07", "POUV07")]
dfCSPStd <- scale(x=dfCSP)
# CAH
cahCSP <- agnes(x=dfCSPStd, metric="euclidean", method="ward")
cahCSP <- as.hclust(cahCSP)
#### Affichage des graphiques permettant de choisir le nombre de classes ###
plot(rev(cahCSP$height),
     type="h", xlab="Noeuds",
     ylab="Niveau d'agrégation", main="Diagramme de niveaux")

h2 <- as.data.frame(sort(cahCSP$height^2,decreasing=T))
colnames(h2) <-"height"
h2$tau <- h2$height*100 / sum(h2$height)
h2 <- h2[1:30,]

plot(h2$tau, type="h", col="red",
     xlab="Noeuds", ylab="% Niv d'aggrégation",
     main="Diagramme de niveaux")

h2$taucum <- cumsum(h2$tau)
h2$nbcla <- as.numeric(row.names(h2)) + 1

plot(h2$nbcla, h2$taucum, type="h", col="red",
     xlab="Nb classes", ylab="% inertie totale",
     main="Diagramme de niveaux")
```

```
plot(cahCSP, xlab="Individus centrés", ylab="Niveaux",
     main="Arbre de classification")

### Découpe de la CAH et description des classes ###
classesCAH <- cutree(tree=cahCSP, k=5)
dfCSP$typo5Cl <- as.factor(classesCAH)

descClasses <- catdes(donnee=dfCSP, num.var=6, proba=1)
descClasses$quanti

descClasses <- catdes(donnee=dfCSP, num.var=6, proba=0.05 )
plot.catdes(x=descClasses)

# Agrégation des données centrées réduites par classes
dfCSPStd2 <- cbind.data.frame(dfCSPStd, classesCAH)
dfCSPStdcla <- aggregate(dfCSPStd2[,1:5],
                         by=list(dfCSPStd2$classesCAH), mean)

rownames(dfCSPStdcla) <- paste("CL", dfCSPStdcla$Group.1, sep="")
# Options graphiques et de palettes
vPal5 <- brewer.pal(n = 5, name = "Set1")
ncla <- nrow(dfCSPStdcla)
# Transformation en matrice et représentation histogrammes par classe
CSPStdcla <- as.matrix(t(dfCSPStdcla[-1]))
for(i in 1:ncla) {
    barplot(CSPStdcla[,i],
            beside = TRUE,
            horiz = TRUE,
            col = vPal5[i],
            names.arg = rownames(CSPStdcla),
            xlim=c(-2,2),
            cex.names=1.2,
            las=2
            )
    title(paste("Classe", i, sep=" "), cex.main=2)
}

dfCSPMerge <- dfCSP
dfCSPMerge$CODCOM <- row.names(dfCSP)
parisPC@data <- merge(x = parisPC@data,
                      y = dfCSPMerge,
                      by.x = "DEPCOM",
                      by.y = "CODCOM")
parisPC@data$ClusterColors <- as.character(vPal5[parisPC@data$typo5Cl])
plot(parisPC, col=parisPC@data$ClusterColors)
legend("topleft",
    legend = paste("Classe",
                   unique(as.character(parisPC@data$typo5Cl)),
                   sep=" "),
    bty = "n",
    fill = vPal5,
    cex = 1.2,
    title = "Classes")

##### Classification des lignes (khi2) #####

# Mise en forme des données
dfPopsPC <- pop36_08[,c("POP1936", "POP1954", "POP1962",
```

```
                        "POP1968", "POP1975", "POP1982",
                        "POP1990", "POP1999", "POP2008")]
row.names(dfPopsPC) <- pop36_08$CODGEO

### Avec hclust() ###
resafc <- dudi.coa(df= dfPopsPC, scannf=FALSE, nf=ncol(dfPopsPC))
distMat <- dist.dudi(dudi= resafc, amongrow=TRUE)
resCAH <- hclust(distMat^2, method="ward")
plot(resCAH)
# Création de l'histogramme des parts d'inertie inter-classes perdues à chaque étape
histo <- as.data.frame(sort(resCAH$height,decreasing=T))
colnames(histo) <- "height"
histo$tau <- histo$height * 100 / sum(histo$height)
# Cumul des parts
histo$taucum <- cumsum(histo$tau)
plot(histo$tau[1:20], type="h",
     col="red",xlab="Noeuds",
     ylab="% Niv d'aggregation",
     main="Diagramme de niveaux")

head(histo)
dfPopsPC$res4cla <- as.factor(cutree(resCAH, k=4))

### Avec agnes() ###
resCAH <- agnes(resafc$li, method="ward")

distMat2 <- dist(x= resafc$li, method="euclidean")
resCAH <- agnes(distMat2, diss=TRUE, method="ward")

# Histogramme des pertes d'inertie
histo <- as.data.frame(sort(resCAH$height^2, decreasing=T))
colnames(histo) <- "height"
histo$tau <- histo$height * 100 / sum(histo$height)
histo$taucum <- cumsum(histo$tau)
plot(histo$tau[1:20], type="h",
     col="red", xlab="Noeuds",
     ylab="% Niv d'aggregation",
     main="Diagramme de niveaux")

dfPopsPC$res4cla <- as.factor(cutree(resCAH, k=4))

### Avec HCPC() ###
resafc <- CA(dfPopsPC[1:9], ncp=10)
resCAH <- HCPC(resafc, nb.clust=0, graph=TRUE, consol=FALSE)
resCAH <- HCPC(resafc, nb.clust=4, graph=F, consol=FALSE)

resCAH$desc.var
resCAH$desc.ind
resCAH$desc.axes

resclust <- resCAH$data.clust
resclust$res4cla <- resclust$clust
dfPopsPC <- merge(dfPopsPC, resclust[2], by="row.names")

#### Description des classes ####

# On crée d'abord un objet de classe by
row.names(dfPopsPC) <- dfPopsPC$Row.names
```

```
dfPopsPC$Classe <- as.integer(as.character(dfPopsPC$res4cla))
dfPopsPC <- dfPopsPC[-c(1,11:12)]
colnames(dfPopsPC) <- sub(pattern="POP", replacement="", x=colnames(dfPopsPC))
bySumPopsPC <- by(data=dfPopsPC[1:9],
                  INDICES=dfPopsPC$Classe,
                  FUN=colSums,
                  simplify=TRUE)

# Puis on en fait un data.frame
dfSumPopsPC <- do.call(rbind.data.frame, bySumPopsPC)
colnames(dfSumPopsPC) <- colnames(dfPopsPC[-10])
dfSumPopsPC

# Pour le poids total des classes
options(scipen=5)
plot(y=dfSumPopsPC[1,], x=colnames(dfSumPopsPC),
     type='l', log="y", col=1,
     ylim=c(min(dfSumPopsPC),max(dfSumPopsPC)),
     xlab="Date", ylab="Population")
lines(y=dfSumPopsPC[2,], x=colnames(dfSumPopsPC),
      type='l', col=2)
lines(y=dfSumPopsPC[3,], x=colnames(dfSumPopsPC),
      type='l', col=3)
lines(y=dfSumPopsPC[4,], x=colnames(dfSumPopsPC),
      type='l', col=4)

legend(x="bottomright", paste("Classe", 1:4, sep=" "),
       cex=1, seg.len=4, col=1:4, lty=1 )
title("Somme de population des classes")

# Puis pour le poids de chaque classe dans la population du système
plot(y=dfSumPopsPC[1,]/colSums(dfSumPopsPC)*100,
     x=colnames(dfSumPopsPC), type='l', col=1,
     ylim=c(min(dfSumPopsPC/colSums(dfSumPopsPC)*100),
            max(dfSumPopsPC/colSums(dfSumPopsPC)*100)),
     xlab="Date", ylab="Population (%)")
lines(y=dfSumPopsPC[2,]/colSums(dfSumPopsPC)*100,
      x=colnames(dfSumPopsPC), type='l', col=2)
lines(y=dfSumPopsPC[3,]/colSums(dfSumPopsPC)*100,
      x=colnames(dfSumPopsPC), type='l', col=3)
lines(y=dfSumPopsPC[4,]/colSums(dfSumPopsPC)*100,
      x=colnames(dfSumPopsPC), type='l', col=4)
title("Évolution du poids des classes dans le système")

byMeanPopsPC <- by(data=dfPopsPC[1:9],
                   INDICES=dfPopsPC$Classe,
                   FUN=colMeans,simplify=TRUE)
dfMeanPopsPC <- do.call(rbind.data.frame, byMeanPopsPC)

colnames(dfMeanPopsPC) <- colnames(dfPopsPC[-10])
# Puis, pour chaque classe, les écarts de ces moyennes à
# la population moyenne d'une ville du système
dfDevPopsPC <- dfMeanPopsPC - colMeans(dfPopsPC[1:9])
dfDevPopsPC

options(scipen=5)
plot(y=dfMeanPopsPC[1,], x=colnames(dfMeanPopsPC),
     type='l', col=1, log="y",
```

```
     ylim=c(min(dfMeanPopsPC), max(dfMeanPopsPC)),
     xlab="Date", ylab="Population")
lines(y=dfMeanPopsPC[2,], x=colnames(dfMeanPopsPC),
      type='l', col=2)
lines(y=dfMeanPopsPC[3,], x=colnames(dfMeanPopsPC),
      type='l', col=3)
lines(y=dfMeanPopsPC[4,], x=colnames(dfMeanPopsPC),
      type='l', col=4)
title("Population moyenne des classes")

plot(y=dfDevPopsPC[1,], x=colnames(dfDevPopsPC),
     type='l', col=1,
     ylim=c(min(dfDevPopsPC),max(dfDevPopsPC)),
     xlab="Date", ylab="Écart à la population moyenne")
lines(y=dfDevPopsPC[2,], x=colnames(dfDevPopsPC),
      type='l', col=2)
lines(y=dfDevPopsPC[3,], x=colnames(dfDevPopsPC),
      type='l', col=3)
lines(y=dfDevPopsPC[4,], x=colnames(dfDevPopsPC),
      type='l', col=4)
abline(h = 0, lty = "dotted")
title("Évolution relative des moyennes de classes")
```

Chapitre 8

Analyse de graphes

Objectifs

Ce chapitre traite de manière générique de graphes, c'est-à-dire d'objets composés de sommets et de liens, chacun pouvant être valué. Selon le domaine, de tels objets peuvent aussi s'appeler « réseaux » : réseaux sociaux, réseaux d'acteurs, réseaux de lieux ... Ce chapitre s'appuie sur un graphe de relations entre les communes, défini sur la base des flux de résidents migrant d'une commune à une autre entre 2 recensements. Les liens sont donc orientés, d'une origine vers une destination. La représentation informatique de ces graphes diffère selon les fonctions utilisées. On propose ici une initiation à l'analyse de graphes avec R : importation et transformation des données, mesures usuelles et options de visualisation. Cela ne prétend nullement à l'exhaustivité et seule une petite partie des possibilités de chaque package *est ici exposée. Pour aller plus loin, voir la documentation de chacun des* packages *évoqués ici et les ressources signalées à la fin du chapitre.*

Prérequis

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

Packages utilisés

Trois packages sont utilisés : `statnet`, `igraph` et `tnet`. Le premier [1], développé par des sociologues, se prête particulièrement à la modélisation statistique des flux (modèles ERGM - *Exponential Random Graph Models*). Il utilise des objets de type *network*. Le second, dévelopéppé par des physiciens, propose toutes les mesures classiques et celles récemment apparues en analyse de graphe (*clustering coefficient*, modularité etc.), il utilise des objets de type *igraph*. Enfin, le dernier développé par l'informaticien Tore Opshal, permet de calculer des indicateurs sur des graphes valués et utilise des objets de type *matrix*.

Les versions utilisées ici sont les suivantes :

- `statnet` : version 3.0-1
- `igraph` : version 0.6-2
- `tnet` : version 3.0.8

Les deux premiers sont conçus pour analyser des graphes booléens (1 si existence d'un lien entre deux sommets, 0 sinon). L'intensité des flux peut être importée commme attributs des liens afin de faire varier l'épaisseur des liens, mais elle ne sera pas prise en compte pour le calcul des indicateurs. Le plus simple est donc d'analyser les flux (voir le chapitre consacré à l'analyse univariée), de créer des graphes correspondant aux seuils choisis et d'analyser séparément ces différents graphes.

Il est déconseillé d'utiliser en même temps les *packages* `statnet` et `igraph`, certaines fonctions - et notamment les fonctions graphiques - étant incompatibles. C'est pourquoi dans ce chapitre il arrive de « détacher » les *packages* incompatibles avec la fonction `detach()`.

1. `statnet` est en réalité un module de modules incluant divers *packages* spécifiques tels `sna`, `network`, `ergm` etc.

Données nécéssaires

- `dMobResid2008.txt` : fichier de mobilités résidentielles entre communes de la petite couronne en 2008 (source INSEE).

Le fichier de départ liste les flux résidentiels entre les communes de la petite couronne. Il comprend 5 colonnes et 15 179 lignes, la première ligne est celle des noms de colonnes, séparateur tabulation ("\t"), décimale ",". Les cinq colonnes sont les suivantes :

- `CODGEO` : code INSEE de la commune d'origine ;
- `LIBGEO` : nom de la commune d'origine ;
- `DCRAN` : code INSEE de la commune de destination ;
- `L_DCRAN` : nom de la commune de destination ;
- `NBFLUX` : volume du flux de personnes ayant déménagé de la commune d'origine vers la commune de destination.

8.1 Préparation des données

La première étape consiste à importer le fichier, à sélectionner les colonnes utiles puis à transformer le *data.frame*, en objet *network* pour l'analyser avec `statnet`, en objet `igraph` pour l'analyser avec `igraph` ou en objet *matrix* pour l'analyser avec `tnet`.

Le script suivant permet l'importation des données, la sélection des colonnes (origine, destination, intensité du flux) et la suppression des boucles (liens intra-communaux). Afin de faciliter les lectures des résultats, seuls les flux majeurs seront considérés : il s'agit de prendre pour chaque commune le flux émis le plus important.

```
library(statnet)
library(igraph)
library(tnet)
```

```
# importation des données
dMob <- read.table("data/dMobResid2008.txt", sep = "\t", dec = ",", header = TRUE,
    fill = TRUE)
```

On sélectionne ensuite les colonnes origine, destination et flux et on supprime les lignes correspondant à des boucles (origine égale destination).

```
# sélection des colonnes origine, destination et flux
dMobResid <- dMob[, -c(2, 4)]

# suppression boucle
dMob <- subset(dMobResid, dMobResid$CODGEO != dMobResid$DCRAN)
```

L'étape suivante consiste à ne sélectionner que le flux majeur, c'est à dire le plus important sortant de chacune des communes d'origine.

```
# sélection du flux le plus important de i

# tri par ordre descendant
dMobOrd1 <- dMob[with(dMob, order(as.numeric(CODGEO), NBFLUX, decreasing = TRUE)),
    ]
```

```
# codage binaire de la première occurrence du champ CODGEO
dMobOrd1$First <- !duplicated(dMobOrd1$CODGEO)

# sélection de la première occurrence, c'est à dire du premier flux
dMobOrd2 <- subset(dMobOrd1, dMobOrd1$First == "TRUE")
```

A partir de ces données, il est possible de créer un graphe : objet de type *network* avec le package `statnet` et de type `igraph` avec le package `igraph`. Il est nécessaire en vue d'utiliser les fonctionnalités du *package* `statnet` de transformer la table de forme « origine - destination », en type `network` (ce qui nécessite au préalable une transformation en type `matrix`).

```
library(statnet)

gMobR <- as.network(as.matrix(dMobOrd2[, c(1:2)]))
gMobR


##  Network attributes:
##   vertices = 138
##   directed = TRUE
##   hyper = FALSE
##   loops = FALSE
##   multiple = FALSE
##   bipartite = FALSE
##   total edges= 136
##     missing edges= 0
##     non-missing edges= 136
##
##  Vertex attribute names:
##     vertex.names
```

En soumettant le nom de l'objet `network` nouvellement créé, on obtient les informations élémentaires de cet objet : nombre de sommets et de liens, caractéristiques du graphe (orienté ou non, présence de boucle, de liens multiples etc.).

Pour utiliser les fonctionnalités du *package* `igraph`, il est également nécessaire de procéder à une transformation de la table origine-destination en objet `igraph`. Les informations fournies par défaut par `igraph` sont beaucoup plus restreintes.

```
library(igraph)

gIMobR <- graph.data.frame(dMobOrd2[, c(1:2)], directed = TRUE)

summary(gIMobR)


## IGRAPH DN-- 138 136 --
## attr: name (v/c)
```

8.2 Description des graphes à l'aide de mesures globales

Les mesures globales (i.e. portant sur le graphe dans son ensemble) les plus couramment utilisées [2] pour décrire un graphe sont :

- l'ordre : nombre de sommets ;

2. La liste suivante est simultanément subjective et non exhaustive...

TABLE 8.1 – Mesures globales d'un graphe g

	statnet	igraph	tnet
Ordre	g	summary(g)	-
Taille	g	summary(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)	clustering_w(g)

* Une option permet de préciser, dans le cas d'un graphe orienté, si on souhaite rechercher les composantes fortement connexes (s'il existe un chemin de i à j, il existe un chemin passant par les mêmes sommets de j à i) ou faiblement connexes (on ne tient pas compte de l'orientation des liens).
**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.

- la taille : nombre de liens ;
- la densité : nombre de liens présents sur nombre de liens possibles ;
- le nombre de composantes connexes : nombre de sous-graphes connexes ;
- le diamètre : longueur du plus long des plus courts chemins ;
- la transitivité ou *global clustering coefficient* : proportion de triangles fermés.

Ces mesures sont particulièrement utiles pour comparer des graphes entre eux. Le tableau « Mesures globales d'un graphe g » donne les noms des fonctions dans les *packages* étudiés ici.

On donne ci-après quelques exemples de calcul de ces indicateurs sur le graphe des flux majeurs `gIMobR` de type `igraph`. Ce graphe est très spécifique puisqu'un seul lien n'émane de chaque nœud. La densité des liens est donc très faible ; le nombre de composantes connexes est très élevé.

```
summary(gIMobR)

## IGRAPH DN-- 138 136 --
## attr: name (v/c)

graph.density(gIMobR)

## [1] 0.007193

clusters(gIMobR)

## $membership
##   [1]  1  2  3  4  5  1  4  1  6  7  2  6  2  8  6  3  2  1  6  2  6  6  1
##  [24]  6  1  4  4  8  2  6  1  8  6  2  6  4  2  6  6  1  6  5  4  2  2  2
##  [47]  4  9  4  4  1 10  4  4  9  4  1  3  2  2  1 11  9  1  4  4  2  4  4
##  [70] 12  4  9 11 12  1 12 12  1 12  4  4  4  4  1  1  2  4  2  2  1  4  2
##  [93]  2  1  2  3  1  4  1  4 12  4  4  4  4  4  4  4  4  8  8  2  4  4  4
## [116]  8  1  1  4  4  2  1  1  1  2  1  4  4  6  4  4  4  6  1  1  1  7 10
##
## $csize
##  [1] 29 23  4 43  2 15  2  6  4  2  2  6
##
## $no
## [1] 12
```

```
# 3 sorties avec cette commande $no : le nombre de composantes connexes
# $csize : leur taille respective en nombre de sommets $membership :
# l'appartenance de chaque sommet aux composantes connexes
diameter(gIMobR)

## [1] 9

dyad.census(gIMobR)

## $mut
## [1] 10
##
## $asym
## [1] 116
##
## $null
## [1] 9327

# nombre de liens mutuels (a vers b et b vers a) asymétriques et nuls
```

D'autres mesures plus complexes peuvent être produites. A titre d'exemple, la fonction `triad.census()` fait référence à une classification (MAN) que nous ne présenterons pas dans le détail ici. Elle renvoie une classification en 16 catégories des triades possibles. Par exemple, le code MAN « 003 » correspond à 0 lien mutuel, 0 lien asymétrique, 3 liens nuls. On renvoie à des manuels spécialisés pour l'interprétation, par exemple [Wasserman et Faust 1994].

```
triad.census(gIMobR)

##  [1] 411619  15364   1334      0    105     88     26      0      0      0
## [11]      0      0      0      0      0      0
```

8.3 Description des graphes à l'aide de mesures locales

Les mesures locales (i.e. mesurées sur chaque sommet ou chaque lien) les plus courantes sont :

- le degré : nombre de liens entrant et sortant pour un graphe orienté ;
- la centralité de proximité (*closeness* ;
- la centralité d'intermédiarité (*betweenness*) ;
- le *local clustering coefficient* ;
- le plus court chemin entre un sommet et les autres sommmets du graphe (si aucun chemin n'existe, la distance entre les deux sommets est considérée infinie).

Les fonctions permettant de calculer ces différents indicateurs sont listées dans le tableau « Mesures locales d'un graphe g ». Attention, toutes ces mesures classiques ne sont pas nécessairement pertinentes pour tous les types de graphes : calculer la centralité d'intermédiarité pour une matrice de flux résidentiels est possible, l'interprétation du résultat sera plus délicate...

Les mesures non pondérées donnent les résultats suivants [3] pour les degrés, se décomposant en degré entrant et degré sortant.

3. J'utilise ici **igraph** mais le script situé en fin de manuel fournit successivement les mesures avec **statnet**, **igraph** et **tnet**

TABLE 8.2 – Mesures locales d'un graphe g

	statnet	igraph	tnet
Degré	`degree(g)`	`degree(g)`	`degree_w(g)`
Proximité	`closeness(g)`	`closeness(g)`	`closeness_w(g)`
Intermédiarité	`betweenness(g)`	`betweenness(g)`	`betweenness_w(g)`
Local clustering coef.	-	`transitivity(g, type="local")`	`clustering_local_w(g)`
Plus court chemins	`geodist(g)`	`shortest.paths(g)`	

```
# degré total
deg <- degree(gIMobR)

# degré entrant
degin <- degree(gIMobR, mode = c("in"))

# degré sortant
degout <- degree(gIMobR, mode = c("out"))
```

Puis les mesures de centralité de proximité (*closeness*) et d'intermédiarité (*betweenness*).

```
# mesures de centralité
clo <- closeness(gIMobR)
betw <- betweenness(gIMobR)

# tableau des résultats
dRes <- data.frame(cbind(gIMobR$CODGEO, deg, degin, degout, clo, betw))
summary(dRes[, -1])
```

```
##      degin           degout           clo                betw
##  Min.   :0.000   Min.   :0.000   Min.   :0.0000529   Min.   : 0.00
##  1st Qu.:0.000   1st Qu.:1.000   1st Qu.:0.0000537   1st Qu.: 0.00
##  Median :0.000   Median :1.000   Median :0.0000541   Median : 0.00
##  Mean   :0.986   Mean   :0.986   Mean   :0.0000544   Mean   : 7.88
##  3rd Qu.:1.000   3rd Qu.:1.000   3rd Qu.:0.0000553   3rd Qu.: 6.00
##  Max.   :6.000   Max.   :1.000   Max.   :0.0000565   Max.   :99.00
```

ll est intéressant de calculer les corrélations entre indicateurs afin de savoir, par exemple, si les communes recevant beaucoup sont aussi celles qui émettent beaucoup. Cette méthode courante en analyse de graphes ne peut être appliquée ici que si on pondère les flux par la population concernée. En effet, les degrés sortant calculés ont un degré de 1 pour toutes les commmunes considérées.

Cette pondération peut être effectuée à l'aide du *package* `tnet`. Les différentes fonctions de `tnet` nécessitent de déterminer un cœfficient α compris entre 0 et 1 : si α vaut 0, l'intensité des liens n'est pas prise en compte ; quand α vaut 1, la somme des valeurs des liens est totalement utilisée.

8.4 Cliques et communautés

L'un des passe-temps favoris de l'analyste de graphes est de chercher à l'intérieur d'un graphe les cliques (sociologues), les communautés (physiciens) ou les *clusters* (informaticiens), à savoir les sous-graphes fortement connexes. De nombreuses méthodes existent pour détecter ces sous-graphes et on n'évoquera ici que les principales.

Une clique désignent un ensemble maximal de sommets entre lesquels tous les liens possibles sont présents.

Une définition plus souple est celle des *k-cores*, à savoir les ensembles d'acteurs entre lesquels tous les liens possibles moins k sont présents.

Les commandes disponibles dans le *package* `statnet` sont les suivantes :

- clique : `clique.census(g)`
- *k-cores* : `kcores(g)`

La commande `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. Pour la fonction `kcores()`, si le graphe est orienté, il faut préciser le degré (entrant, sortant ou total) utilisé pour déterminer les *kcores* à l'aide de l'option `cmode = ""` (`indegree`, `outdegree` ou `freeman`) - les résultats seront évidemment très différents dans les trois cas.

Le *package* `igraph` propose les fonctions suivantes :

- cliques : `cliques(g)`
- *k-cores* : `graph.coreness(g)`

La densité du graphe étudié ici est telle que cette recherche a ici peu d'intérêt... Les commandes proposées dans le script en fin de chapitre sont donc uniquement là pour information. Attention, la recherche des cliques est particulièrement lente avec `igraph`.

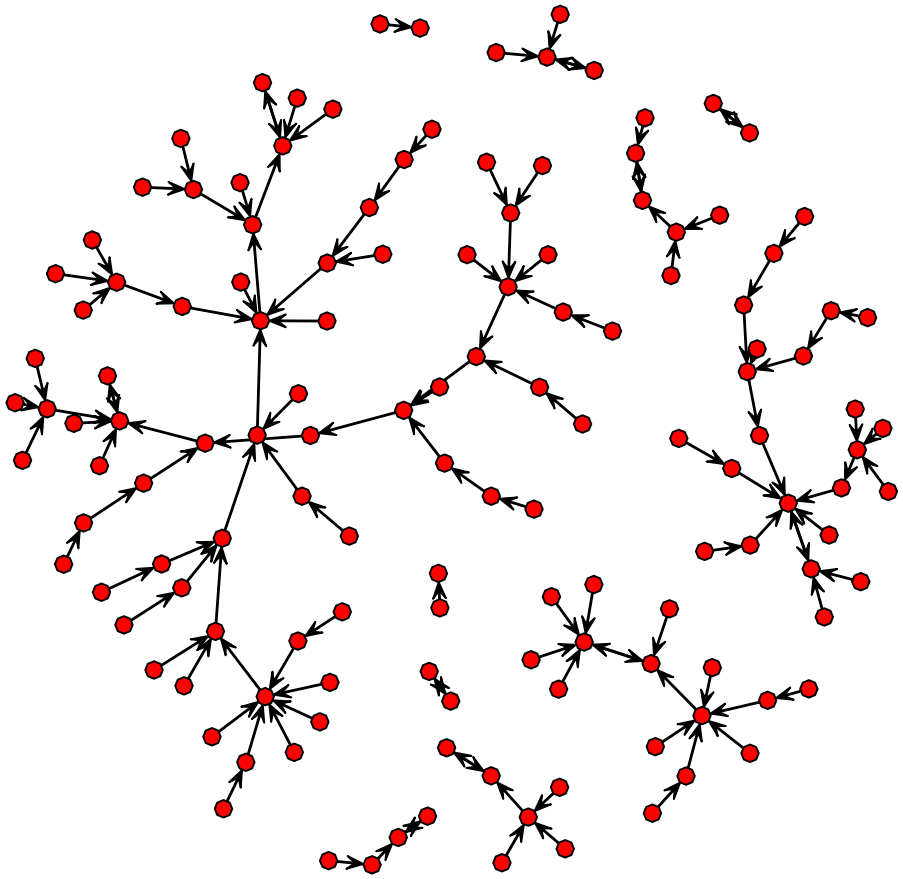
8.5 Visualisations

Visualiser les graphes est l'un des réflexes les plus courants - et peut-être l'un des plus trompeurs [4] - en analyse de graphes. Les deux *packages* proposent des outils et options comparables.

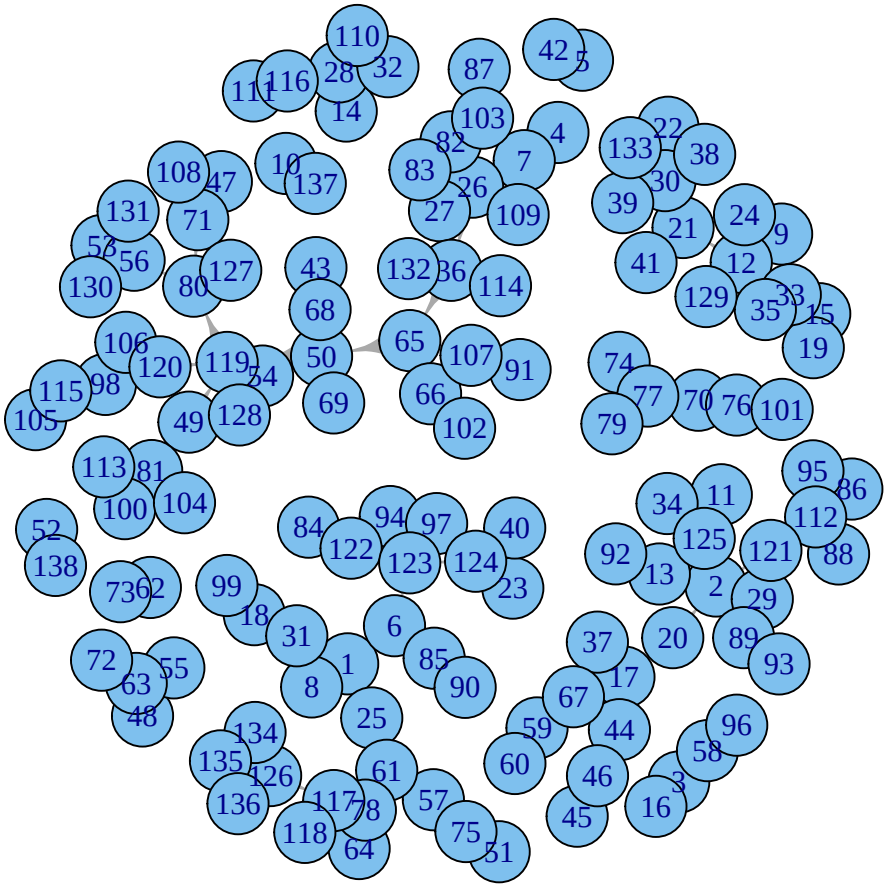
Le résultat par défaut est plus lisible avec `statnet` qu'avec `igraph`, comme le montre les deux figures suivantes.

```
detach(package:igraph, force = TRUE)
library(statnet)
gplot(gMobR)
```

4. Commenter une image sans connaître les principes de l'algorithme utilisé pour la visualisation revient à commenter une carte choroplète dont on ne connaitrait pas la méthode de discrétisation utilisée...



```
detach(package:statnet, force = TRUE)
detach(package:sna, force = TRUE)
detach(package:network, force = TRUE)
library(igraph)
plot(gIMobR)
```



Les deux *packages* proposent une version interactive (i.e. possibilité de déplacer les sommets à sa guise) et une version 3D (fausse pour `igraph`).

Pour `statnet`, les fonctions seront :

- `gplot(g)` ;
- `gplot(g, interactive=TRUE)` ;
- `gplot3d(g)`.

Et pour `igraph` :

- `plot(g)` ;
- `tkplot(g)` ;
- `rglplot(g)` ;

Plusieurs algorithmes de visualisation existent avec de très nombreux paramétrages possibles, qu'il s'agisse des sommets, des étiquettes, des liens ou du fond. Consulter l'aide de ces fonctions est recommandé, en particulier les fonctions `gplot.layout()` pour le package `snatnet` et `layout()` pour le pakage `igraph`.

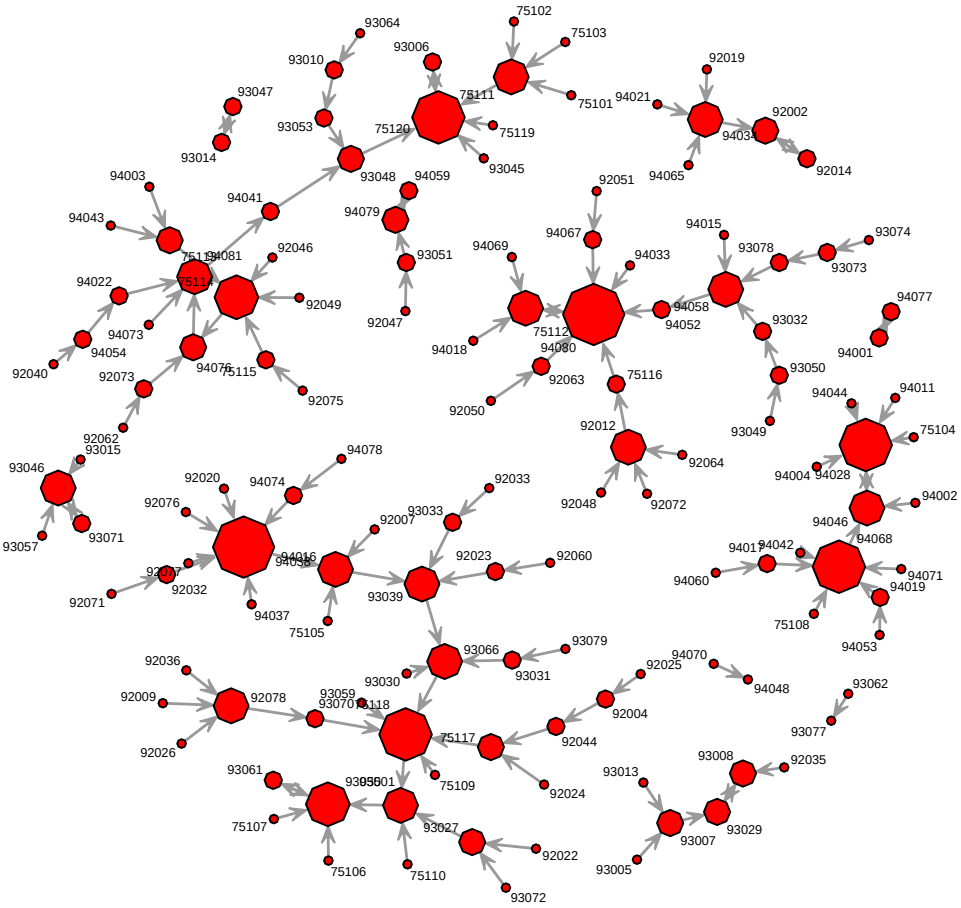
Voici un exemple de script personnalisé avec `statnet` : la couleur des liens est modifiée, la taille des sommets est proportionnelle au degré.

```
detach(package:igraph, force = TRUE)
library(statnet)
```

```
class(gMobR)

## [1] "network"

gplot(gMobR, displayisolates = FALSE,
     vertex.cex = degree(gMobR)/2,
     label = network.vertex.names(gMobR),
     label.cex = 0.4,
     label.pos = 0,
     vertex.col = "red",
     edge.col = "grey60",
     mode= "fruchtermanreingold",
     main="Flux résidentiels majeurs en 2008\n
     Paris et petite couronne",
     sub="Source : INSEE")
```

Flux résidentiels majeurs en 2008

Paris et petite couronne



Source : INSEE

8.6 Code

```
#importation des données
dMob <- read.delim("data/dMobResid2008.txt", dec=",")

#sélection des colonnes origine, destination et flux
dMobResid <- dMob[,-c(2,4)]

#suppression boucle
dMob <- subset(dMobResid, dMobResid$CODGEO != dMobResid$DCRAN)

#sélection du flux le plus important de i

#tri par ordre descendant
dMobOrd1 <- dMob[with(dMob, order(as.numeric(CODGEO),
```

```
NBFLUX, decreasing=TRUE)), ]

#test pour contrôler la redondance
dMobOrd1$F <- !duplicated(dMobOrd1$CODGEO)

#suppresion des flux non majeurs
dMobOrd2 <- subset(dMobOrd1, dMobOrd1$F == "TRUE")

#avec le package statnet
#transformation en objet network

library(statnet)

gMobR <- as.network(as.matrix(dMobOrd2[,c(1:2)]))

#résumé des informations
gMobR

#indices globaux
#densité
gden(gMobR)

#commposantes connexes
components(gMobR)

#diades
dyad.census(gMobR)

#triades
triad.census(gMobR)

#transitivité
gtrans(gMobR)

#indices locaux
#degré
degree(gMobR)

#degré entrant
degree(gMobR ,cmode="indegree")

#degré sortant
degree(gMobR ,cmode="outdegree")

#centralité de proximité
closeness(gMobR)

#centralité d'intermédiarité
betweenness(gMobR)

#sous-graphes fortement connexes
clique.census(gMobR, mode = "digraph", tabulate.by.vertex=TRUE)
kcores(gMobR, cmode = "indegree")
kcores(gMobR, cmode = "outdegree")
kcores(gMobR, cmode = "freeman")

#visualisation
#par défaut
```

```
gplot(gMobR)
#interactive
gplot(gMobR, interactive=TRUE)
#3D
gplot3d(gMobR)

#algorithmes de visualisation
help(package="sna", "gplot.layout")

#personnalisée
#choix des couleurs
#liens courbes
gplot(gMobR, displayisolates=FALSE,
label=network.vertex.names(gMobR),
label.cex=0.4,
label.pos=0,
vertex.col="red",
vertex.cex=degree(gMobR)/2,
edge.col="grey60",
mode="fruchtermanreingold",
main="Flux résidentiels dominants en 2008\n Paris et petite coutonne",
sub="Source : INSEE")

########################
#avec le package igraph
########################

library(igraph)
gIMobR <- graph.data.frame(dMobOrd2, directed=TRUE)

#caractéristiques principales
summary(gIMobR)

#mesures globales
#densité
graph.density(gIMobR)

#diamètre
diameter(gIMobR)

#composantes connexes
clusters(gIMobR)

#diades
dyad.census(gIMobR)

#triades
triad.census(gIMobR)

#transitivité
transitivity(gIMobR)

#mesures locales
#degré entrant, sortant et total
degree(gIMobR, mode = c("in"))
degree(gIMobR, mode = c("out"))
degree(gIMobR)
```

```
closeness(gIMobR)
betweenness(gIMobR)

#clustering coefficient local
transitivity(gIMobR, type = c("local"))

#sous-graphes fortement connexes
#attention : la fonction cliques est très lente dans igraph
cliques(gIMobR)
graph.coreness(gIMobR)

#visualisation
#par défaut
plot(gIMobR)
#interactive

tkplot(gIMobR)
#(fausse) 3D

rglplot(gIMobR)
#personnalisée

plot(gIMobR,
layout=layout.fruchterman.reingold,
vertex.size=4,
vertex.label.dist=0.5,
vertex.color="orange",
edge.arrow.size=0.5,
edge.color="palegreen")

##############
#package tnet
##############

#transformation de gIMobR en "objet" tnet (matrix)
library(tnet)
mTnet <- cbind(get.edgelist(gIMobR, names=FALSE)+1, dMobOrd2$NBFLUX)
#degré entrant avec 3 alpha différents
degin0 <- degree_w(net=mTnet, measure=c("degree", "output", "alpha"),
alpha=0, type="in")
degin05 <- degree_w(net=mTnet, measure=c("degree", "output", "alpha"),
alpha=0.5, type="in")
degin1 <- degree_w(net=mTnet, measure=c("degree", "output", "alpha"),
alpha=1, type="in")
degin <- data.frame(cbind(degin0, degin05, degin1))
head(degin, 8)
```

8.7 Pour aller plus loin

Pour aller plus loin dans l'analyse de graphe avec R, voir les sites respectifs des *packages* évoqués ici [5] :

- statnet : http ://csde.washington.edu/statnet/
- igraph : http ://igraph.sourceforge.net/
- tnet : http ://toreopsahl.com/tnet/

La *mailing list* de `statnet` est consacrée quasi exclusivement aux modèles ERGM, celle d'`igraph` est plus généraliste.

Ressources internet en français :

- tutoriel de Julien Barnier : http ://alea.fr.eu.org/pages/reseaux
- tutoriel de Laurent Beauguitte : http ://cel.archives-ouvertes.fr/
- billets du blog fmr : http ://groupefmr.hypotheses.org/

Ressources internet en anglais :

- Workshop de la Sunbelt : http ://bojan.3e.pl/bojanorama/doku.php ?id=snar :start
- Présentation de D. Conway : http ://www.drewconway.com/

Sans oublier bien entendu l'indispensable R-bloggers http ://www.r-bloggers.com/.

Sur l'analyse de graphes telle qu'elle est abordée ici, voir notamment [Wasserman et Faust 1994], [Lazéga 2007], [Newman 2010] ainsi que les différentes synthèses produites par le groupe fmr (flux, matrices, réseaux) et disponibles sur halshs [Groupe fmr].

5. Tous ces sites ont été visités en août 2012.

Chapitre 9

Cartographie

9.1 Objectifs et prérequis

Objectifs

*Ce chapitre vise à réaliser des cartes simples d'objets vectoriels (points et polygones) et d'images raster. Il est en effet possible de lire et de manipuler tous types de formats de données spatiales et d'obtenir des sorties cartographiques de bonne qualité. Si, pour faire une simple carte choroplèthe, il est certainement plus rapide et plus simple d'utiliser 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) trois situations dans lesquelles R s'avèrera avantageux :*

- 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 réaliser manuellement des ensembles de nombreuses cartes ;
- pour intégrer une carte dans un flux de travail « classique » (analyse de données classiques, i.e. non spatiales) entièrement réalisé 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 quelconque, pour réexporter les résultats à cartographier, etc. ;
- pour intégrer la cartographie 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.

Prérequis

Discrétisation ; formats d'objets spatiaux (vectoriel, raster) ; sémiologie graphique.

Packages nécessaires

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 [Bivand *at al.* 2008].

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. Il est utilisé ici pour importer les fichiers en format MapInfo ;
- `raster`, permet de manipuler des fichiers raster ;
- `rgeos`, interface avec la bibliothèque Geometry Engine - Open Source (GEOS) qui permet de manipuler la géométrie des objets ;
- `mapproj`, permet de convertir des coordonnées géographiques en coordonnées projetées ;
- `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. On l'utilise pour faire des cartes choroplèthes ;
- `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 ;
- `reshape`, contient un ensemble de fonctions permettant de restructurer l'information d'un tableau et notamment de transformer des matrices longues en matrices larges et *vice versa*.

Données nécessaires

Plusieurs types de fichiers sont utilisés 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 `readOGR()` permet d'importer tous les formats de la bibliothèque OGR (PostGIS, kml, Idrisi, etc.). Les fichiers nécessaires sont les suivants :

- Fichier `pop36_08.csv` : fichier des populations dans les communes de Paris et la petite couronne de 1936 à 2008 ;
- Fichier `paripccomregion.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 ;
- Fichier `paripchopitauxfin.tab` : établissements hospitaliers de Paris et petite couronne, données vectorielles (points) au format MapInfo constitué de quatre fichiers avec les extensions .ind, .id, .mif, .map ;
- Fichier `Paris_Densite.tif` : grille de population, données raster au format .tif.

Pour une description plus précise du contenu des fichiers, voir la Section 1.6.

Préparation des données

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*. Le préfixe `s-` est ajouté, comme pour l'objet `sCommunes`, pour signifier que ce sont des données spatiales.

La fonction `readOGR()` peut lire des formats contenant plusieurs couches, c'est pourquoi il ne suffit pas de préciser le nom du fichier (premier argument de la fonction), mais également le nom de la couche (`layer =`). Dans le cas des fichiers ESRI (.shp) et MapInfo (.TAB), ils ne contiennent qu'une seule couche (du même nom que le fichier) qu'il faut préciser avec l'option `layer`. L'option `encoding` permet de préciser l'encodage, de la même façon que dans l'importation de simples tableaux.

Pour importer l'image raster, la fonction `readGDAL()` du *package* `raster` est utilisée.

```
library(sp)
library(rgdal)
library(raster)
library(ggplot2)
library(scales)
library(rgeos)
library(mapproj)
library(RColorBrewer)
library(classInt)
library(reshape)
library(maptools)
```

L'option `stringsAsFactors` sert à préciser si les champs stockés comme texte doivent être conservés comme tels ou bien transformés en facteurs (cf. Chapitre 2). Lors de l'importation de données non spatiales ce choix est libre et dépend de chaque utilisateur. En revanche, lors de l'importation de données spatiales, il faut absolument spécifier `stringsAsFactors = FALSE`. En effet, si l'identifiant des objets spatiaux (ici le code INSEE des communes) est un champ textuel, le type *factor* peut entraîner des confusions lors des jointures, car il combine pour un même enregistrement une valeur (*level*) et une étiquette de valeur (*label*). En d'autres termes, le *factor* est utile pour des variables qualitatives classiques dont les modalités regroupent plusieurs cas (n modalités $<$ n cas), mais pas pour des variables qualitatives exhaustives ou identifiants (n modalités $=$ n cas).

```
sCommunes <- readOGR("data/paripc_com_region.shp",
                     layer = "paripc_com_region",
                     encoding = "latin1",
                     stringsAsFactors = FALSE)

sHopitaux <- readOGR("data/paripc_hopitaux_fin.TAB",
                     layer = "paripc_hopitaux_fin",
                     encoding = "latin1",
                     stringsAsFactors = FALSE)

sDensite <- readGDAL("data/Paris_Densite.tif")
```

9.2 Prise en main des données spatiales

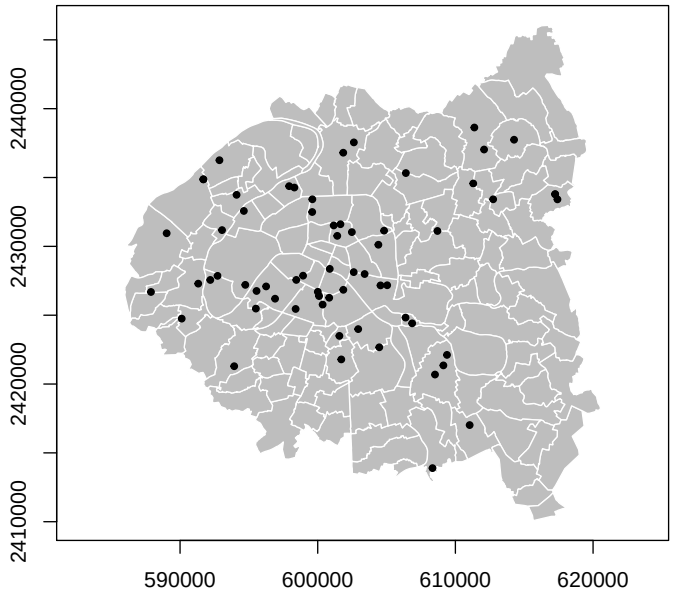
Il est intéressant d'examiner le type et la structure de ces données spatiales grâce aux fonctions `class()` et `str()`. La fonction `str()` renvoie les attributs, le type de données spatiales, elle précise l'étendue des données (*bbox*) et le système de coordonnées (*proj4string*).

```
class(sCommunes)
class(sHopitaux)

str(sCommunes)
str(sHopitaux)
```

Premières visualisations

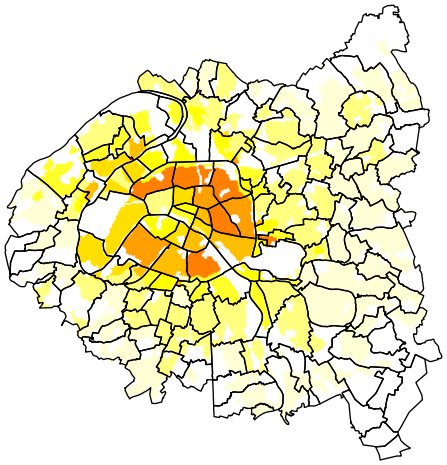
Une fois les données importées, elles peuvent être visualisées avec la fonction `plot()`. On utilise l'option `col =` pour préciser la couleur de remplissage, l'option `border =` pour la couleur du trait, l'option `pch =` pour le type de point, et finalement l'option `add = TRUE` pour superposer chaque nouvelle couche à la précédente :

```
plot(sCommunes, col = "grey", border = "white", axes = TRUE)
plot(sHopitaux, pch = 20, col = "black", add = TRUE)
```



Pour visualiser le raster, on commence par créer une palette de couleurs (montée en intensité et blanc pour les valeurs nulles) puis la fonction `image()` permet de représenter le raster. Pour travailler directement sur les valeurs des pixels, il est possible de transformer l'objet raster (*SpatialGridDataFrame*) en une matrice de pixels avec la fonction `as.matrix()`.

```
pal <- c("white", rev(heat.colors(12)))
image(sDensite, col = pal)
plot(sCommunes, axes = T, add = T)
```



```
mPixels <- as.matrix(sDensite)
```

La conception de l'échelle et de la légende est détaillée dans la section suivante.

Modifier le système de projection

Jusqu'à présent, les données spatiales ont été importées et visualisées sans se soucier de leur système de projection. 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. Le résultat n'est pas très parlant et il faut traduire cette information dans la liste de codes EPSG. Le site Internet `http://prj2epsg.org/` permet de faire cette traduction en récupérant l'information contenue dans le fichier d'extension .prj du fichier ESRI (shapefile) d'origine et en donnant le code EPSG équivalent. La projection de l'objet `sCommunes` semble mal référencée et sera convertie en « Lambert II étendu ». 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 systèmes de projection avec leurs identifiants. La fonction `grep()` permet de rechercher les expressions régulières, c'est-à-dire les chaînes de caractères répétés (motifs). Dans ce *data.frame*, on trouve les 68 systèmes de projection de la famille Lambert. Finalement, le code EPSG correspondant au Lambert II étendu est repéré, c'est le code 27572.

```
strwrap(proj4string(sCommunes))

## [1] "+proj=lcc +lat_1=45.90287723937 +lat_2=47.69712276063 +lat_0=46.8"
## [2] "+lon_0=2.337229104484 +x_0=600000 +y_0=2200000 +ellps=clrk80"
## [3] "+units=m +no_defs"

dEPSG <- make_EPSG()
dLambert <- dEPSG[grep("Lambert", dEPSG$note), 1:2]
dLambert[59:68, ]

##       code                                        note
## 2892 21500 # Belge 1950 (Brussels) / Belge Lambert 50
## 2954 22770              # Deir ez Zor / Syria Lambert
## 3062 24600                                # KOC Lambert
## 3408 27561        # NTF (Paris) / Lambert Nord France
## 3409 27562      # NTF (Paris) / Lambert Centre France
## 3410 27563         # NTF (Paris) / Lambert Sud France
## 3411 27564              # NTF (Paris) / Lambert Corse
## 3412 27571             # NTF (Paris) / Lambert zone I
## 3413 27572            # NTF (Paris) / Lambert zone II
## 3414 27573           # NTF (Paris) / Lambert zone III
```

Il s'agit maintenant de définir le système de projection des données spatiales. Un objet `sCommunesL2` est créé en définissant son système (Lambert II étendu) grâce à la fonction `spTransform()` qui ajoute au système initial le nouveau code EPSG. Ensuite l'opération est répétée pour l'objet `sHopitaux`, et enfin on vérifie que le code EPSG a bien été modifié :

```
sCommunesL2 <- spTransform(sCommunes, CRS("+init=epsg:27572"))
sHopitauxL2 <- spTransform(sHopitaux, CRS("+init=epsg:27572"))
proj4string(sCommunesL2)
proj4string(sHopitauxL2)
```

Exportations

Au cours du travail cartographique, on peut souhaiter exporter deux types d'objets : des résultats (des images) ou bien des objets spatiaux. R permet d'utiliser des dispositifs de sorties graphiques pour exporter les résultats

en plusieurs formats avec les fonctions `pdf()`, `png()`, `svg()`, `jpeg()` etc. Il est aussi possible d'utiliser l'interface graphique de RStudio, avec le bouton « Export » de l'onglet « Plot ».

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 OGR (`http://www.gdal.org/ogr/`) recense tous les formats disponibles. Dans l'exemple qui suit, le fichier de communes est exporté au format ESRI (.shp) dans un dossier intitulé « MonDossier » (si ce dossier n'existe pas, il est créé au moment de l'exportation) :

```
writeOGR(sCommunes, dsn = "MonDossier",
         layer = "paripc_com_region",
         driver="ESRI Shapefile")
```

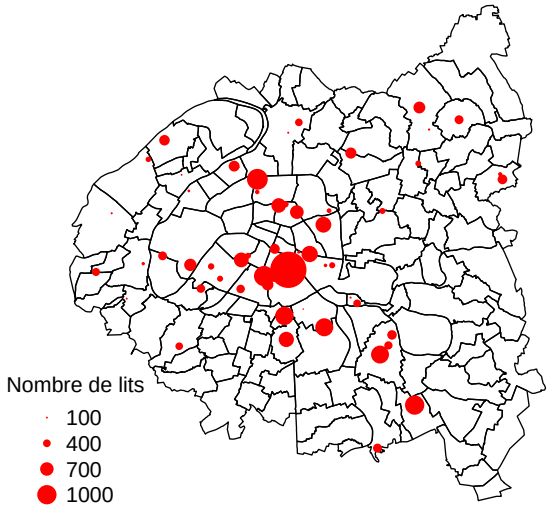
9.3 Cartographie des objets ponctuels : cercles proportionnels

Les hôpitaux peuvent être représentés pour faire apparaître l'attribut « Capacité moyenne totale des hôpitaux » en cercles proportionnels. La fonction générique `plot()` est utilisée en faisant varier la taille des symboles (option `cex =`, *character expansion factor*) selon la capacité des hôpitaux. On note au passage que l'argument `cex =` accepte une fonction. Ici la taille du cercle est la capacité de l'hôpital divisée par 500. Il serait aussi possible d'utiliser une fonction qui lisse les très grandes valeurs, par exemple une racine carrée ou un logarithme.

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 options prédéfinies (`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 options `pch =`, `col =` et `pt.cex =` sont utilisées pour spécifier respectivement le type de figuré, la couleur et la taille des points, et l'option `bty =` permet de dessiner ou non un encadré de la légende.

```
plot(sCommunesL2)

plot(sHopitauxL2,
     pch = 16,
     cex = sHopitauxL2$capacite_moy_totale / 500,
     col = "red", add = T)

title("Capacité moyenne des hôpitaux à Paris - Petite couronne")

vValeursLegende <- c(100, 400, 700, 1000)

legend("bottomleft",
       legend = vValeursLegende,
       pch = 16,
       col = "red",
       pt.cex = vValeursLegende / 500,
       bty = "n",
       title = "Nombre de lits")
```

Capacité moyenne des hôpitaux à Paris – Petite couronne



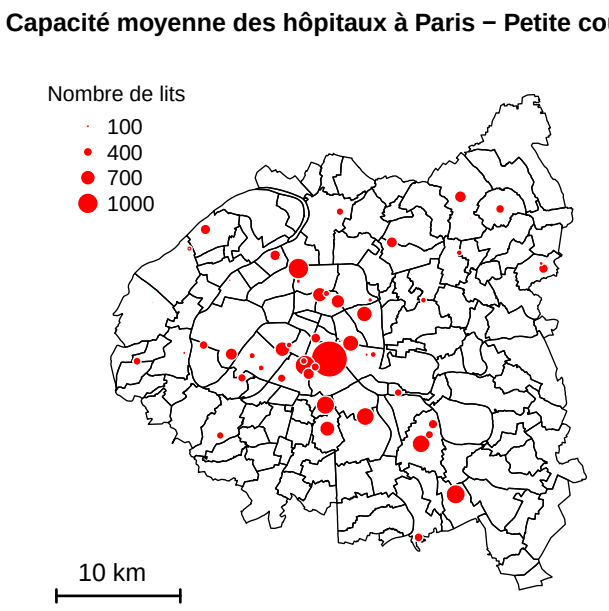
Pour ajouter une échelle, il y a deux solutions, qui demandent toutes deux de spécifier manuellement la localisation du trait, la longueur du trait, l'épaisseur et le texte. Dans le cas ci-dessous sont utilisées les fonctions `arrows()` pour le trait de l'échelle, puis `text()` pour le texte de l'échelle. 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'option `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'option `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).

```
plot(sCommunesL2)

sHopitauxL2Tri <- sHopitauxL2[order(sHopitauxL2@data$capacite_moy_totale, decreasing = TRUE), ]

plot(sHopitauxL2Tri,
     pch = 21,
     cex = sHopitauxL2Tri$capacite_moy_totale / 500,
```

```
     col = "white",
     bg = "red",
     add = T)

title("Capacité moyenne des hôpitaux à Paris - Petite couronne")

vValeursLegende <- c(100, 400, 700, 1000)

legend("topleft",
       legend = vValeursLegende,
       pch = 16,
       col = "red",
       pt.cex = vValeursLegende / 500,
       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)
```



9.4 Cartographie des objets zonaux : cartes choroplèthes

A la date de préparation de ce manuel, il existe principalement deux façons de réaliser 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 réalisation d'une carte choroplèthe de la densité de population par commune en 2008.

Pour commencer, il faut importer les données externes contenant les variables de population, puis calculer la densité à partir des données de population et de la surface de la commune.

```
dPop3608 <- read.csv("data/pop36_08.csv",
                     sep = ";",
                     stringsAsFactor = FALSE,
                     encoding = "latin1")

dPop3608$DENSITE2008 <- dPop3608$POP2008 / dPop3608$SURF
```

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.

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. La fonction `display.brewer.all()` permet de visualiser les palettes disponibles. Si on ne précise pas le type souhaité (`type =`), la fonction renvoie l'ensemble des palettes, sinon on peut choisir entre trois types : divergente (`div`), avec montée de valeurs (`seq`) ou avec variation de couleurs (`qual`).

Deux palettes de couleurs sont créées, la première est une palette divergente à quatre couleurs pour la discrétisation en quartiles, avec couleurs froides en-dessous de la médiane et couleurs chaudes au-dessus. La seconde est une palette continue avec cinq couleurs de même ton et d'intensité variable (montée de valeurs) pour la discrétisation en cinq classes par l'algorithme de Jenks :

```
vPal4 <- rev(brewer.pal(n = 4, name = "RdYlBu"))
vPal5 <- brewer.pal(n = 5, name = "Reds")
```

Ensuite on fait une jointure entre les données attributaires de l'objet spatial et les données externes contenues dans le *data.frame* `dPop3608` grâce à la fonction `match()`. Il y a plusieurs précautions à prendre avant de faire cette jointure, la première concerne le type de la variable de jointure. Il est utile de vérifier avec la fonction class() que le champ des données attributaires et celui des données externes sont bien du même type. Ici, le code INSEE est de type texte, et ce format a bien été conservé lors de l'importation des données grâce à l'option `stringsAsFactors = FALSE`.

La seconde précaution concerne l'intégrité de la table attributaire de l'objet spatial : elle ne doit être 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 `merge()` peut être source d'erreur pour réaliser ce type de jointure, et il faut donc trouver des moyens de garantir que la table attributaire ne subit aucune modification. Le script suivant se lit de la façon suivante : la table attributaire finale (`sCommunesL2@data`) est une combinaison (`data.frame()`) de la table attributaire initiale et du tableau externe (`dPop3608`). Les lignes du tableau externe sont ajoutées quand il y a correspondance des identifiants, et l'ajout conserve l'ordre et le nombre du permier argument de la fonction `match()`, à savoir la table attributaire de l'objet spatial.

Une dernière précaution à observer, qui ne concerne pas spécifiquement R mais tous les logiciels de SIG qui permettent ce genre de manipulations. Lorsque le tableau de données externes contient plusieurs cas pour un même objet géométrique, la fonction `match()` ira chercher tous les cas qui correspondent à l'identifiant de la table attributaire et le nombre de lignes de cette table sera modifié. Cette erreur n'est pas propre à R ni à la fonction utilisée, c'est une erreur de conception préalable à l'analyse : un seul champ d'un seul objet géométrique ne peut recevoir qu'une seule valeur.

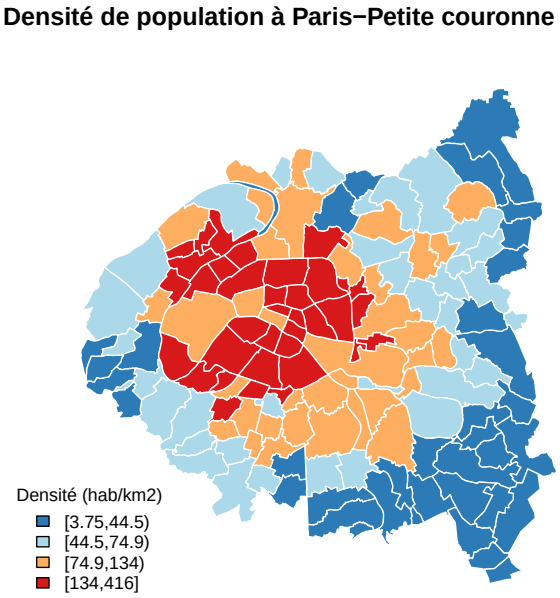
```
sCommunesL2@data <- data.frame(sCommunesL2@data,
                               dPop3608[match(sCommunesL2@data[, "DEPCOM"],
                                              dPop3608[, "CODGEO"]), ])
```

Finalement, la jointure réalisée est de type `left join`, elle pourrait également être réalisée avec des commandes SQL grâce au *package* `sqldf` (cf. Section 2.3). Une fois la jointure réalisée, on peut travailler directement sur les données attributaires, les discrétiser et les représenter. Notons que 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 4) et un label (étiquette de valeur) qui prend ici la couleur de la palette créée précédemment. L'option `break =` est utilisée pour préciser les seuils, l'option `labels =` pour préciser les étiquettes des valeurs, les options `include.lowest = TRUE` et `right = FALSE` pour créer des intervalles fermés à gauche et ouverts à droite ([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 (DENSQ4) dans la table attributaire contiendra donc le code couleur utilisé par la suite avec la fonction `plot()`.

```
vQuartile2008 <- quantile(sCommunesL2@data$DENSITE2008, names = TRUE)

sCommunesL2@data$DENSQ4 <- as.character(cut(sCommunesL2@data$DENSITE2008,
                                            breaks = vQuartile2008,
                                            labels = vPal4,
                                            include.lowest = TRUE,
                                            right = FALSE))

vLegendBoxQ4 <- as.character(levels(cut(sCommunesL2@data$DENSITE2008,
                                        breaks = vQuartile2008,
                                        include.lowest = TRUE,
                                        right = FALSE)))
```

Enfin, on peut passer à la représentation cartographique proprement dite :

```
plot(sCommunesL2, col = sCommunesL2@data$DENSQ4, border = "white")

legend("bottomleft",
       legend = vLegendBoxQ4,
       bty = "n",
       fill = vPal4,
       cex = 0.8,
       title = "Densité (hab/km2)")

title(main="Densité de population à Paris-Petite couronne (2008)")
```

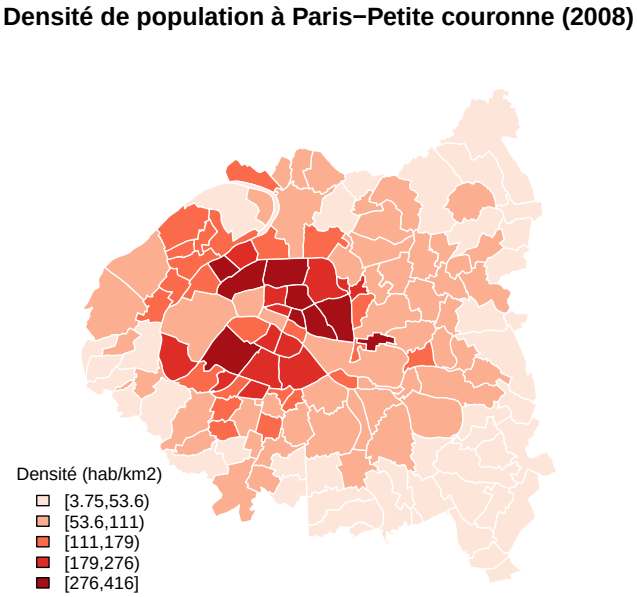


On répète les mêmes opérations, mais cette fois-ci en discrétisant avec l'algorithme de Jenks et en utilisant la palette à cinq couleurs :

```
lJenks2008 <- classIntervals(var = sCommunesL2@data$DENSITE2008,
                             n = 5,
                             style = "jenks")

vJenks2008 <- lJenks2008$brks

sCommunesL2@data$DENSJ5 <- as.character(cut(sCommunesL2@data$DENSITE2008,
                                            breaks = vJenks2008,
                                            labels = vPal5,
                                            include.lowest = TRUE,
                                            right = FALSE))

vLegendBoxJ5 <- as.character(levels(cut(sCommunesL2@data$DENSITE2008,
                                        breaks = vJenks2008,
                                        include.lowest = TRUE,
                                        right = FALSE)))
```

Finalement la carte choroplèthe en cinq catégories de densité :

```
plot(sCommunesL2,
     col = sCommunesL2@data$DENSJ5,
     border = "white")

legend("bottomleft",
       legend = vLegendBoxJ5,
       bty = "n",
       fill = vPal5,
       cex = 0.8,
       title = "Densité (hab/km2)")

title(main = "Densité de population à Paris-Petite couronne (2008)")
```



Cartographie avec la fonction `ggplot()`

La fonction `ggplot()` est une fonction du *package* `ggplot2` très utile pour tout type de représentations graphiques. L'idée générale de `ggplot2` est de s'appuyer sur une « grammaire des graphiques » pour généraliser et simplifier la production d'objets graphiques et cartographiques. La fonction `ggplot()` permet de personnaliser les aspects de la représentation plus facilement et 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. Aussi bien cette transformation que la fonction de représentation elle-même sont plus lentes que la fonction `plot()`.

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. Pour le présent jeu de données, composé de 143 communes et arrondissements, cela ne pose pas de problème, mais il est pour le moment impossible de faire, sur un ordinateur de bureau, des cartes des 36 000 communes françaises avec `ggplot()`.

Les mêmes discrétisations que dans la section précédente sont réalisées, mais ici les variables sont discrétisées dans le tableau de données externes et non dans le tableau de données attributaires de l'objet spatial :

```
dPop3608$DENS2008Q4 <- cut(dPop3608$DENSITE2008,
                           breaks = vQuartile2008,
                           include.lowest = TRUE,
                           right = FALSE,
                           labels = c("Q1", "Q2", "Q3", "Q4"))

dPop3608$DENS2008JENKS <- cut(dPop3608$DENSITE2008,
                              breaks = vJenks2008,
                              include.lowest = TRUE,
                              right = FALSE,
                              labels = c("G1", "G2", "G3", "G4", "G5"))
```

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*. La fonction `fortify()` du *package* `ggplot2` réalise 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 que l'identifiant du polygone (id). On fait ensuite une jointure sur l'objet « fortifié » des données externes avec les variables discrétisées. On examine la structure de cet objet « fortifié », il s'agit d'un *data.frame* classique où chaque ligne est un sommet définissant un polygone. On passe d'une structure où un polygone équivaut à une ligne à une structure où un polygone équivaut à n lignes, n étant le nombre de sommets du polygone.

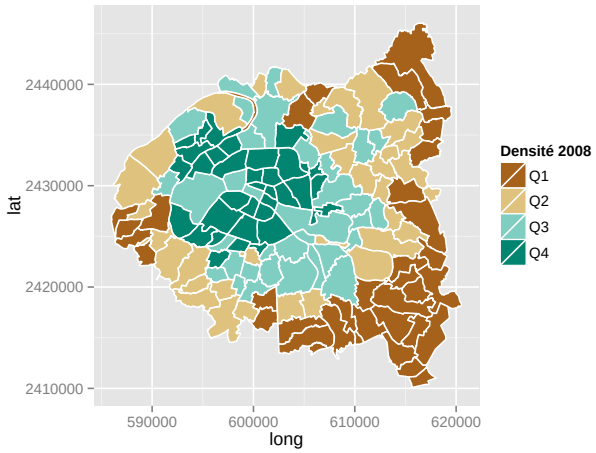
La fonction `geom_polygon()` sert à graphier des objets zonaux, l'option `group =` désigne l'identifiant des polygones et l'option `fill =` désigne la couleur de remplissage, contenue dans le champ `DENS2008Q4` créé précédemment. Finalement on modifie l'objet graphique (`pMap`) avec la fonction `coord_equal()` 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 (ici une palette divergente).

```
dCommunes <- fortify(sCommunesL2, region="DEPCOM")
head(dCommunes)
```

```
##     long     lat order  hole piece   group    id
## 1 601021 2429340     1 FALSE     1 75101.1 75101
## 2 601003 2429291     2 FALSE     1 75101.1 75101
## 3 601001 2429286     3 FALSE     1 75101.1 75101
## 4 600971 2429206     4 FALSE     1 75101.1 75101
## 5 600946 2429140     5 FALSE     1 75101.1 75101
## 6 600944 2429135     6 FALSE     1 75101.1 75101
```

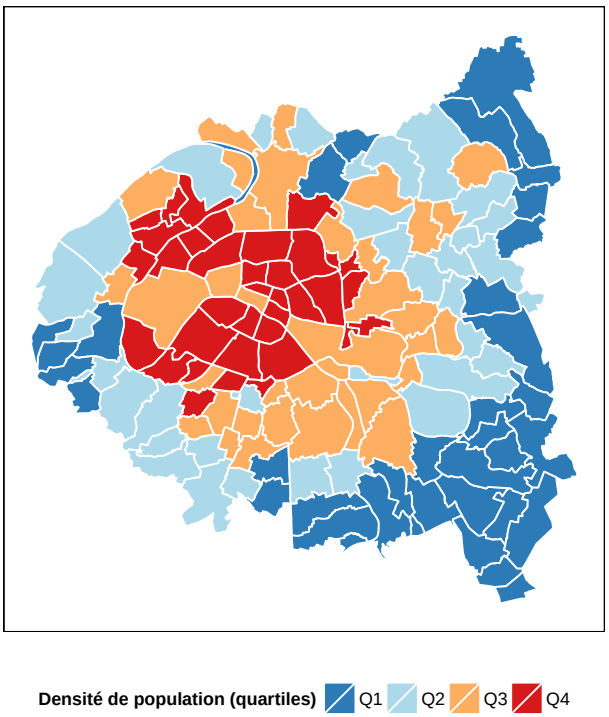
```
dCommunes <- merge(x = dCommunes,
                   y = dPop3608,
                   by.x = "id",
                   by.y = "CODGEO",
                   all.x = TRUE,
                   sort = FALSE)

pMap <- ggplot() + geom_polygon(data = dCommunes,
                                aes(long, lat, group = group, fill = DENS2008Q4),
                                colour = "white")

pMap <- pMap + coord_equal() + scale_fill_brewer(name = "Densité 2008", type = "div")
```

```
pMap
```



Le fond gris et la palette de couleurs ne conviennent pas. En effet, les graphiques réalisés avec `ggplot()` contiennent par défaut une trame grise et des axes. Pour faire la carte, on préfère un fond blanc avec un simple encadré sans axes, on personnalise donc le thème par défaut, et on récupère par la même occasion la palette de couleurs créée précédemment :

```
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")

pMap + scale_fill_manual(name = "Densité de population (quartiles)", values = vPal4)
```



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 réaliser 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 :

- intégrer toutes les cartes dans une même page ;
- stocker toutes les cartes dans 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. Ces fonctions prennent comme argument une

boucle qui cartographie les données à chaque date du recensement. La troisième option nécessite l'utilisation du *package* `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. Les deux options sont montrées successivement.

Il s'agit d'abord d'introduire dans l'objet fortifié `dCommunes` (celui que `ggplot()` prend comme argument) les densités pour chaque année du recensement.

```
dDensites <- dPop3608[ , 3:11] / dPop3608$SURF

mQuartiles <- apply(dDensites, 2, quantile, names = TRUE)

mClassDens <- matrix(nrow = 143, ncol = 9)

for(i in 1:9){
  mClassDens[ ,i] <- as.character(cut(dDensites[ , i],
                                      breaks = mQuartiles[ , i],
                                      include.lowest = TRUE,
                                      right = FALSE,
                                      labels = c("Q1", "Q2", "Q3", "Q4")))
}

dClassDens <- as.data.frame(cbind(dPop3608$CODGEO, mClassDens))

colnames(dClassDens) <- c("CODGEO", "DENS1936", "DENS1954", "DENS1962",
                          "DENS1968", "DENS1975", "DENS1982",
                          "DENS1990", "DENS1999", "DENS2008")

dCommunesDens <- merge(x = dCommunes,
                       y = dClassDens,
                       by.x = "id",
                       by.y = "CODGEO",
                       all.x = TRUE,
                       sort = FALSE)
```

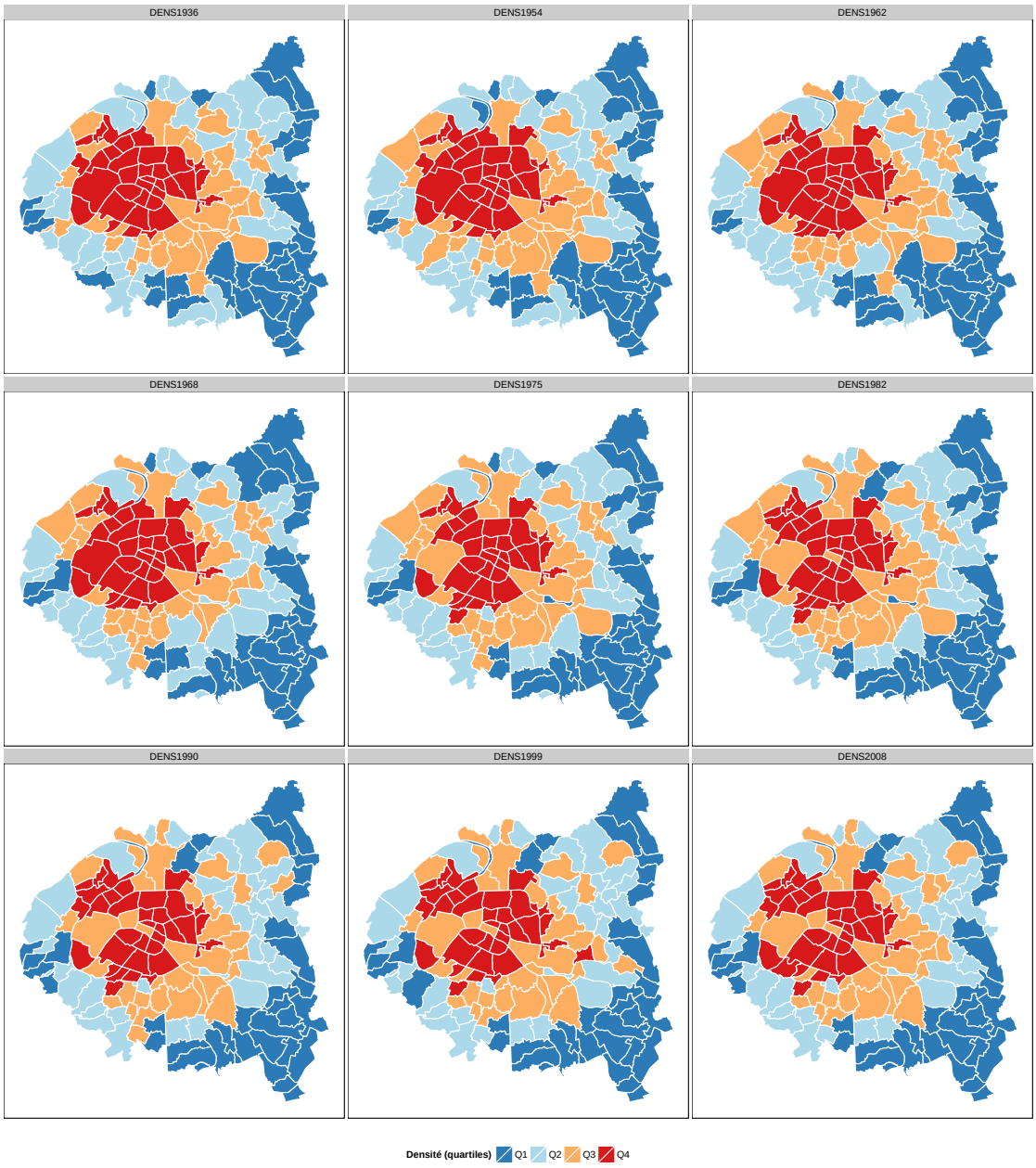
Une fois calculées et introduites les densités dans l'objet `dCommunes`, l'objet est transformé avec la fonction `melt()` du *package* `reshape` (cf. Section 2.3). L'objet `dCommunes` est une « matrice large » contenant neuf champs de densité, un champ pour chaque année du recensement. Il faudrait en faire une matrice longue qui n'aurait que deux champs à graphier : un champ contenant une variable qualitative indiquant l'année du recensement et un champ contenant la variable à représenter (ici des classes de densités) :

```
dCommunesMatriceLarge <- dCommunesDens[ , c(1:7, 22:30)]
dCommunesMatriceLongue <- melt(dCommunesMatriceLarge,
                               id = c("id", "long", "lat", "order",
                                      "hole", "piece", "group"))
```

Cet objet peut être cartographié, en utilisant l'option `facet_wrap` pour intégrer l'ensemble des cartes dans une seule sortie. Cette option intègre dans une même sortie n cartes, une pour chaque modalité de la variable donnée comme argument (`facets =`).

```
pMapDens <- ggplot() + geom_polygon(data = dCommunesMatriceLongue,
                                    aes(long, lat, group = group, fill = value),
                                    colour = "white")
```

```
pMapDens +
    facet_wrap(facets=~variable) +
    scale_fill_manual(name = "Densité (quartiles)", values = vPal4) +
    coord_equal()
```



Il s'agit maintenant de réaliser le même ensemble de cartes, mais en discrétisant sur l'ensemble des données de 1936 à 2008. Pour cela on récupère l'objet `dDensites`, *data.frame* qui contient pour toutes les communes et arrondissement (143 cas) les valeurs de tous les recensements depuis 1936 (9 champs). Il n'est pas nécessaire de joindre toutes ces données dans un seul champ pour calculer les quartiles : la fonction `quantile()` peut être appliquée à l'ensemble du tableau, à condition de le transformer en matrice (l'objet `matrix` est un vecteur bidimensionnel à la différence de l'objet *data.frame* qui se rapproche plutôt de l'objet *list*).

```
vQuartiles <- quantile(as.matrix(dDensites), names = TRUE)
mClassDensEnsemble <- apply(dDensites, 2, cut,
                            breaks = vQuartiles,
                            include.lowest = TRUE,
                            right = FALSE,
```

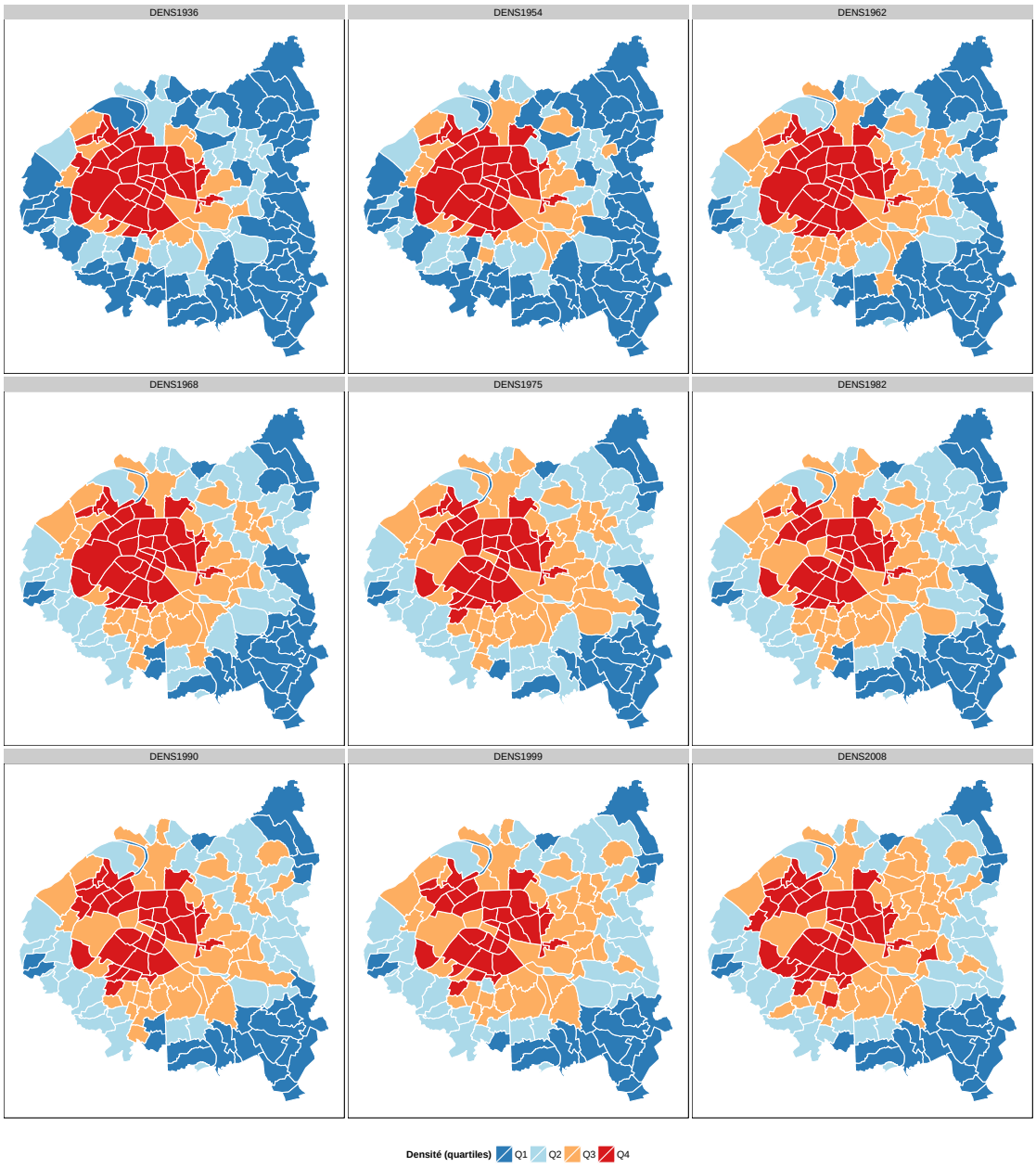
```
                            labels = c("Q1", "Q2", "Q3", "Q4"))

dClassDensEnsemble <- as.data.frame(cbind(dPop3608$CODGEO, mClassDensEnsemble))

colnames(dClassDensEnsemble) <- c("CODGEO", "DENS1936", "DENS1954", "DENS1962",
                                  "DENS1968", "DENS1975", "DENS1982",
                                  "DENS1990", "DENS1999", "DENS2008")

dCommunesDensEnsemble <- merge(x = dCommunes,
                               y = dClassDensEnsemble,
                               by.x = "id",
                               by.y = "CODGEO",
                               all.x = TRUE,
                               sort = FALSE)
```

Même opération que précédemment, on construit une matrice longue avec la fonction `melt()` :

```
dCommunesMatriceLargeEns <- dCommunesDensEnsemble[ , c(1:7, 22:30)]

dCommunesMatriceLongueEns <- melt(dCommunesMatriceLargeEns,
                                  id = c("id", "long", "lat", "order",
                                         "hole", "piece", "group"))
```

Puis on cartographie les données :

```
pMapDensEns <- ggplot() + geom_polygon(data = dCommunesMatriceLongueEns,
                                       aes(long, lat, group = group, fill = value),
                                       colour = "white")

pMapDensEns +
    facet_wrap(facets = ~variable) +
    scale_fill_manual(name = "Densité (quartiles)", values = vPal4) +
    coord_equal()
```



9.5 Code

```
### CHARGEMENT DES PACKAGES

library(sp)
library(rgdal)
library(raster)
library(ggplot2)
library(scales)
library(rgeos)
library(mapproj)
library(RColorBrewer)
library(classInt)
library(reshape)
library(maptools)
```

```
### IMPORTATION

sCommunes <- readOGR("paripc_com_region.shp",
                     layer = "paripc_com_region",
                     input_field_name_encoding = "latin1")

sHopitaux <- readOGR("paripc_hopitaux_fin.TAB",
                     layer = "paripc_hopitaux_fin",
                     input_field_name_encoding = "latin1")

sDensite <- readGDAL("Paris_Densite.tif")

### PRISE EN MAIN DES OBJETS SPATIAUX

class(sCommunes)
class(sHopitaux)

str(sCommunes)
str(sHopitaux)

plot(sCommunes, col = "grey", border = "white", axes = TRUE)
plot(sHopitaux, pch = 20, col = "black", add = TRUE)


### RASTER

pal <- c("white", rev(heat.colors(12)))
image(sDensite, col = pal)
plot(sCommunes, axes = T, add = T)

mPixels <- as.matrix(sDensite)


### PROJECTION

strwrap(proj4string(sCommunes))
dEPSG <- make_EPSG()
dLambert <- dEPSG[grep("Lambert", dEPSG$note), 1:2]
dLambert[59:68, ]

sCommunesL2 <- spTransform(sCommunes, CRS("+init=epsg:27572"))
sHopitauxL2 <- spTransform(sHopitaux, CRS("+init=epsg:27572"))
proj4string(sCommunesL2)
proj4string(sHopitauxL2)


### EXPORTATION

writeOGR(sCommunes, dsn = "MonDossier",
         layer = "paripc_com_region",
         driver="ESRI Shapefile")


### CARTOGRAPHIE DES OBJETS PONCTUELS

plot(sCommunesL2)

plot(sHopitauxL2,
```

```
     pch = 16,
     cex = sHopitauxL2$capacite_moy_totale / 500,
     col = "red", add = T)

title("Capacité moyenne des hôpitaux à Paris - Petite couronne")

# légende

vValeursLegende <- c(100, 400, 700, 1000)

legend("bottomleft",
       legend = vValeursLegende,
       pch = 16,
       col = "red",
       pt.cex = vValeursLegende / 500,
       bty = "n",
       title = "Nombre de lits")


plot(sCommunesL2)

# carte finale des hôpitaux

sHopitauxL2Tri <- sHopitauxL2[order(sHopitauxL2@data$capacite_moy_totale, decreasing = TRUE), ]

plot(sHopitauxL2Tri,
     pch = 21,
     cex = sHopitauxL2Tri$capacite_moy_totale / 500,
     col = "white",
     bg = "red",
     add = T)

title("Capacité moyenne des hôpitaux à Paris - Petite couronne")

vValeursLegende <- c(100, 400, 700, 1000)

legend("topleft",
       legend = vValeursLegende,
       pch = 16,
       col = "red",
       pt.cex = vValeursLegende / 500,
       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)


### CARTOGRAPHIE DES OBJETS ZONAUX
```

```
# importation des données

dPop3608 <- read.csv("data/pop36_08.csv",
                     sep = ";",
                     stringsAsFactor = FALSE,
                     encoding = "latin1")

dPop3608$DENSITE2008 <- dPop3608$POP2008 / dPop3608$SURF

# création des palettes de couleurs

vPal4 <- rev(brewer.pal(n = 4, name = "RdYlBu"))
vPal5 <- brewer.pal(n = 5, name = "Reds")


# jointure sur table attributaire

sCommunesL2@data <- merge(x = sCommunesL2@data,
                          y = dPop3608,
                          by.x = "DEPCOM",
                          by.y = "CODGEO",
                          all.x = TRUE)


# discrétisation en quantiles

vQuartile2008 <- quantile(sCommunesL2@data$DENSITE2008, names = TRUE)

sCommunesL2@data$DENSQ4 <- as.character(cut(sCommunesL2@data$DENSITE2008,
                                            breaks = vQuartile2008,
                                            labels = vPal4,
                                            include.lowest = TRUE,
                                            right = FALSE))

vLegendBoxQ4 <- as.character(levels(cut(sCommunesL2@data$DENSITE2008,
                                        breaks = vQuartile2008,
                                        include.lowest = TRUE,
                                        right = FALSE)))


# légende et titre

plot(sCommunesL2, col = sCommunesL2@data$DENSQ4, border = "white")

legend("bottomleft",
       legend = vLegendBoxQ4,
       bty = "n",
       fill = vPal4,
       cex = 0.8,
       title = "Densité (hab/km2)")

title(main="Densité de population à Paris-Petite couronne (2008)")


# discrétisation selon algorithme de Jenks

lJenks2008 <- classIntervals(var = sCommunesL2@data$DENSITE2008,
                             n = 5,
```

```
                              style = "jenks")

vJenks2008 <- lJenks2008$brks

sCommunesL2@data$DENSJ5 <- as.character(cut(sCommunesL2@data$DENSITE2008,
                                            breaks = vJenks2008,
                                            labels = vPal5,
                                            include.lowest = TRUE,
                                            right = FALSE))

vLegendBoxJ5 <- as.character(levels(cut(sCommunesL2@data$DENSITE2008,
                                        breaks = vJenks2008,
                                        include.lowest = TRUE,
                                        right = FALSE)))

# affichage de la carte choroplèthe

plot(sCommunesL2,
     col = sCommunesL2@data$DENSJ5,
     border = "white")

legend("bottomleft",
       legend = vLegendBoxJ5,
       bty = "n",
       fill = vPal5,
       cex = 0.8,
       title = "Densité (hab/km2)")

title(main = "Densité de population à Paris-Petite couronne (2008)")


### CARTOGRAPHIE AVEC ggplot()

# discrétisation de la variable DENSITE

dPop3608$DENS2008Q4 <- cut(dPop3608$DENSITE2008,
                           breaks = vQuartile2008,
                           include.lowest = TRUE,
                           right = FALSE,
                           labels = c("Q1", "Q2", "Q3", "Q4"))

dPop3608$DENS2008JENKS <- cut(dPop3608$DENSITE2008,
                              breaks = vJenks2008,
                              include.lowest = TRUE,
                              right = FALSE,
                              labels = c("G1", "G2", "G3", "G4", "G5"))

# fortification de l'objet spatial

dCommunes <- fortify(sCommunesL2, region="DEPCOM")
head(dCommunes)

# jointure sur la table attributaire

dCommunes <- merge(x = dCommunes,
                   y = dPop3608,
                   by.x = "id",
                   by.y = "CODGEO",
```

```
                    all.x = TRUE,
                    sort = FALSE)

pMap <- ggplot() + geom_polygon(data = dCommunes,
                                aes(long, lat,
                                    group = group,
                                    fill = DENS2008Q4),
                                colour = "white")

pMap <- pMap + coord_equal() + scale_fill_brewer(name = "Densité 2008", type = "div")

# modification du thème (axes, titres, 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")

pMap + scale_fill_manual(name = "Densité de population (quartiles)", values = vPal4)

### CARTOGRAPHIE A LA CHAINE

# calcul des densités
dDensites <- dPop3608[ , 3:11] / dPop3608$SURF

mQuartiles <- apply(dDensites, 2, quantile, names = TRUE)

mClassDens <- matrix(nrow = 143, ncol = 9)

for(i in 1:9){
  mClassDens[ ,i] <- as.character(cut(dDensites[ , i],
                                      breaks = mQuartiles[ , i],
                                      include.lowest = TRUE,
                                      right = FALSE,
                                      labels = c("Q1", "Q2", "Q3", "Q4")))
}

dClassDens <- as.data.frame(cbind(dPop3608$CODGEO, mClassDens))

colnames(dClassDens) <- c("CODGEO", "DENS1936", "DENS1954", "DENS1962",
                          "DENS1968", "DENS1975", "DENS1982",
                          "DENS1990", "DENS1999", "DENS2008")

dCommunesDens <- merge(x = dCommunes,
                       y = dClassDens,
                       by.x = "id",
                       by.y = "CODGEO",
                       all.x = TRUE,
                       sort = FALSE)

### DISCRETISATION SUR CHAQUE ANNEE DE LA SERIE TEMPORELLE

# transformation en matrice longue
```

```
dCommunesMatriceLarge <- dCommunesDens[ , c(1:7, 22:30)]
dCommunesMatriceLongue <- melt(dCommunesMatriceLarge,
                               id = c("id", "long", "lat", "order",
                                      "hole", "piece", "group"))

# carte finale
pMapDens <- ggplot() + geom_polygon(data = dCommunesMatriceLongue,
                                    aes(long, lat, group = group, fill = value),
                                    colour = "white")

pMapDens +
    facet_wrap(facets=~variable) +
    scale_fill_manual(name = "Densité (quartiles)", values = vPal4) +
    coord_equal()


### DISCRETISATION SUR L'ENSEMBLE DE LA SERIE TEMPORELLE
vQuartiles <- quantile(as.matrix(dDensites), names = TRUE)
mClassDensEnsemble <- apply(dDensites, 2, cut,
                            breaks = vQuartiles,
                            include.lowest = TRUE,
                            right = FALSE,
                            labels = c("Q1", "Q2", "Q3", "Q4"))

dClassDensEnsemble <- as.data.frame(cbind(dPop3608$CODGEO, mClassDensEnsemble))

colnames(dClassDensEnsemble) <- c("CODGEO", "DENS1936", "DENS1954", "DENS1962",
                                  "DENS1968", "DENS1975", "DENS1982",
                                  "DENS1990", "DENS1999", "DENS2008")

dCommunesDensEnsemble <- merge(x = dCommunes,
                               y = dClassDensEnsemble,
                               by.x = "id",
                               by.y = "CODGEO",
                               all.x = TRUE,
                               sort = FALSE)

# transformation en matrice longue
dCommunesMatriceLargeEns <- dCommunesDensEnsemble[ , c(1:7, 22:30)]

dCommunesMatriceLongueEns <- melt(dCommunesMatriceLargeEns,
                                  id = c("id", "long", "lat", "order",
                                         "hole", "piece", "group"))

# carte finale
pMapDensEns <- ggplot() + geom_polygon(data = dCommunesMatriceLongueEns,
                                       aes(long, lat, group = group, fill = value),
                                       colour = "white")

pMapDensEns +
    facet_wrap(facets = ~variable) +
    scale_fill_manual(name = "Densité (quartiles)", values = vPal4) +
    coord_equal()
```

Chapitre 10

Initiation aux statistiques spatiales

Objectif

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

Analyse des semis de points, autocorrélation spatiale

Packages R nécessaires

- `rgeos` : permet de manipuler la géométrie des objets
- `rgdal` : pour l'importation de données géographiques
- `maptools` : autre *package* d'importation
- `reshape` : pour les conversions de format de données
- `spdep` : pour les calculs de statistiques spatiales

Données

Dans ce chapitre, nous utiliserons des variables géométriques et attributaires, données recensées dans les tableaux suivants, dont la description est dans le chapitre d'introduction :

- `Data99_07.csv` Données pour communes Paris PC en 1999 et 2007
- `paripc_com_region.*` Données géométriques des communes
- `paripc_hopitaux_fin.*` Établissements hospitaliers sur Paris et PC

Préparation des données

On donne rapidement les lignes de programme pour la lecture des différents fichiers nécessaires. On renvoie aux chapitres précédents pour leur compréhension.

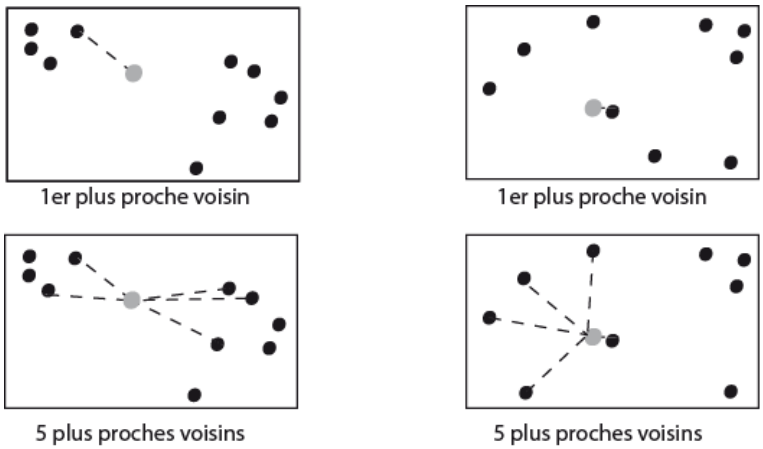
```
# Chargement des packages
require('maptools', quietly=TRUE)
require('rgdal', quietly=TRUE)
require('rgeos', quietly=TRUE)
require('reshape', quietly=TRUE)
require('spdep', quietly=TRUE)

# Lecture des fichiers communes
sdCommunes <- readShapePoly('data/paripc_com_region.shp')
class(sdCommunes)
summary(sdCommunes)
plot(sdCommunes, axes=T)
sdCommunes$DEPCOM <- as.character(sdCommunes$DEPCOM)

# Lecture des fichiers hôpitaux
sdHopito <- readOGR("data/paripc_hopitaux_fin.TAB",
                    layer="paripc_hopitaux_fin",
                    input_field_name_encoding="latin1")
plot(sdHopito, col="red", add=T)
rownames(sdHopito@data) <- as.character(sdHopito$CodHop)

# Lecture des fichiers statistiques pour les communes
data99_07 <- read.csv("data/data99_07.csv",
                      sep=";",
                      stringsAsFactor=FALSE)
rownames(data99_07) <- as.character(data99_07$CODGEO)
```

10.1 Mesurer les espacements d'une distribution ponctuelle

10.1.1 Rappels méthodologiques : les notions 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 des espacements entre les points, résumés à l'ensemble du semis.
Dans ces approches, la distance joue un rôle fondamental. On travaillera ici en distance euclidienne « à vol d'oiseau ». Pour certaines questions, il peut être nécessaire d'utiliser une distance plus appropriée. Dans ce cas, il faut construire un objet « matrice de distance » qui pourrait être importé d'un SIG (distance sur réseau, distance à une frontière, entre deux points, etc) ou calculée selon une autre formule à partir de métriques mathématiques (distance de Manhattan, famille de distances de Minkowski...) (cf. Section 2.3). Cette matrice sert de base à la description des espacements entre points du semis. Le plus souvent, les méthodes développées travaillent sur la notion de « distance au plus proche voisin ».

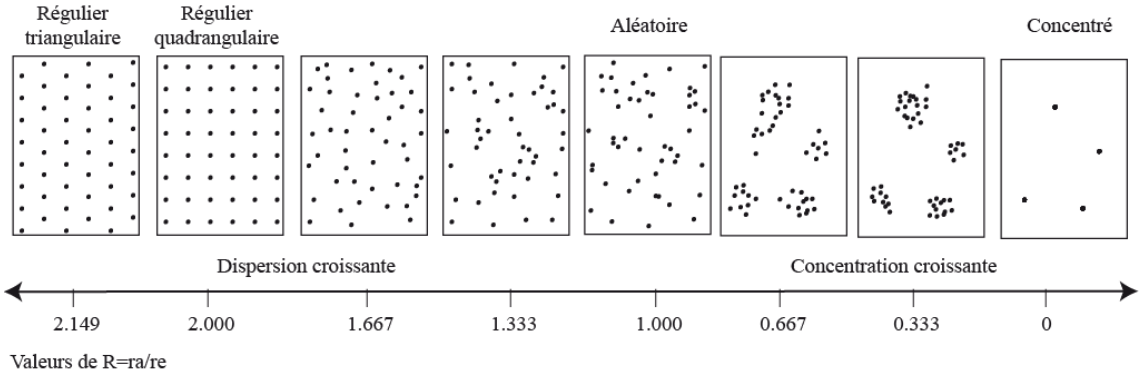
La distance au plus proche voisin est utilisée dans un cadre descriptif : pour décrire des contextes locaux, ou pour être intégrée dans d'autres analyses, comme variable explicative (par exemple) dans une régression multiple. Dans ce cas, elle peut aussi être généralisée à des ordres supérieurs comme par exemple la distance moyenne aux 5 plus proches voisins.



La « distance moyenne au plus proche voisin » (d_{ppv}) est aussi la statistique utilisée dans certaines méthodes pour caractériser l'organisation du semis (concentré, régulier, aléatoire). On se place dans ce cas dans un cadre de statistique inférentielle, en comparant cette mesure, à une mesure théorique obtenue dans le cadre d'une expérience aléatoire de Poisson, qui génère un semis dit « aléatoire » [Pumain et Saint-Julien 1997, Zaninetti 2005]. Dans ce cadre, on peut calculer la distance moyenne théorique d'un point à son plus proche voisin :

$$d_{théo} = \frac{1}{2} \times \sqrt{\frac{S}{M}}$$

où S est la superficie de l'espace étudié et M le nombre de points localisés sur cet espace.
La figure suivante donne des illustrations des différents cas valeurs du rapport $d_{ppv}/d_{théo}$:
Il vaut 1 lorsque le semis est « aléatoire ». Il vaut 2,15 pour un semis régulier. Il vaut 0 pour un semis concentré où un grand nombre de points sont superposés.



Source : [Pumain et Saint-Julien 1997]

10.1.2 L'exemple du semis des hôpitaux

On étudiera ici le semis des hôpitaux sur l'espace de Paris et la petite couronne. On travaille avec un objet de type *SpatialPointDataframe* (cf. 9).

```
str(head(sdHopito))
```

```
## 'data.frame': 6 obs. of  8 variables:
##  $ CodHop             : Factor w/ 64 levels "185","186","187",..: 64 63 62 61 60 59
##  $ Numéro_FINESS      : Factor w/ 64 levels "750000481","750000499",..: 63 62 57 55 61 60
##  $ Raison_sociale     : Factor w/ 63 levels "CENTRE DE LONG SEJOUR LES ORMES",..: 49 36 7 47 34 30
##  $ Code_postal        : Factor w/ 46 levels "75004","75007",..: 46 45 44 44 43 42
##  $ CodHop_2           : Factor w/ 64 levels "185","186","187",..: 64 63 62 61 60 59
##  $ x                  : num  601714 611037 606860 606380 601575 ...
##  $ Y                  : num  2421788 2417020 2424408 2424826 2423496 ...
##  $ capacite_moy_totale: num  791 1000 400 110 957 934
```

On observera que la partie `@data` de l'objet spatial comprend les coordonnées des hôpitaux. Mais elles sont ici considérées comme des attributs. La partie « géométrique » à proprement parler est contenue dans la partie `@coords` qui peut être appelée comme un vecteur. Ou, si l'on veut plus spécifiquement travailler sur la coordonnée x ou la coordonnée y, il faudra les appeler *via* la partie `coord` (ici `coord.x1` ou `coord.x2`).

```
row.names(sdHopito) <- as.character(sdHopito$CodHop)
coordsHop <- coordinates(sdHopito)
```

L'objectif est alors de mesurer la distance moyenne au plus proche voisin pour chacun des hôpitaux parisiens recensés dans le tableau de données afin de caractériser la dispersion du semis. Pour pouvoir la comparer à une distance théorique, on commence par calculer ce que serait cette distance dans le cadre d'un semis théorique résultant d'un processus aléatoire de Poisson.
On prêtera attention aux différents systèmes d'expression des coordonnées et unités de surface. Les hôpitaux ont des coordonnées exprimées en mètres, tandis que l'aire de l'espace étudié est calculée à partir de la variable `SURF` du tableau `data99_07` exprimée en hectares. On choisit de travailler en mètres.

```
Distppvtheo <- 0.5 * sqrt(sum(data99_07$SURF) * 10000/nrow(sdHopito))
```

Le calcul de la distance moyenne au plus proche voisin peut-être traité de différentes manières et nous présentons ici chacune d'elles :

1. Calculer au préalable la distance entre chaque hôpital (distancier, cf. Section 2.3). Cette méthode a l'avantage de créer un objet sur lequel on va pouvoir travailler et tester un certain nombre d'indicateurs (distance au plus proche voisin, distance moyenne au 5 plus proches voisins, distance au $3^{\text{ème}}$ plus proche voisin etc.). Elle a l'inconvénient, si l'on a un grand nombre d'objets, de créer un objet sur-dimensionné par rapport à l'utilisation que l'on peut en faire. Ce serait le cas si l'on cherche par exemple à identifier à l'échelle d'une région métropolitaine, la dimension « locale » des choix de collèges des élèves. Seules les proximités de moins de quelques kilomètres sont à prendre en compte relativement à la distance maximale entre les 2 collèges les plus éloignés qui peut dépasser une centaine de km.
2. Utiliser une suite de fonctions préexistantes dans le `package spdep` spécialisé dans les statistiques spatiales. L'inconvénient est l'enchaînement de fonctions dont certaines servent essentiellement à changer de format pour l'utilisation de fonctions différentes.

Démarche *via* le calcul d'une matrice de distance Le distancier se calcule à partir de colonnes (x,y) contenues dans une table de données :
On construit la matrice de distance entre hôpitaux avec la fonction `dist()`. Nous choisissons ici la méthode euclidienne, mais plusieurs options peuvent être choisies (`euclidean`, `maximum`, `manhattan`, `canberra`, `binary` ou `minkowski`).

```
distHop <- dist(coordsHop, method = "euclidean", diag = FALSE)
str(distHop)
```

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. Les deux derniers arguments sont les arguments par défaut.

Il faut ensuite transformer cet objet en un objet matrice $n \times n$ en utilisant la fonction `as.matrix()`.

```
matdistHop <- as.matrix(distHop)
str(matdistHop)
matdistHop
```

On effectue ensuite le calcul de la distance au plus proche voisin pour chaque hôpital (`mindistHI`), 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.

```
matdistHop[matdistHop == 0] <- NA
mindistHI <- apply(matdistHop, 1, min, na.rm = TRUE)
mean(mindistHI)
RIndex <- mean(mindistHI)/Distppvtheo
RIndex
```

La valeur moyenne de la distance au plus proche voisin est de 1393 mètres tandis que la valeur théorique est de 1730 mètres. Le rapport vaut 0.8, le semis des hôpitaux serait plutôt concentré. Par ailleurs, cette valeur peut être testée statistiquement.

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. Le programme suivant permet de créer une table à n lignes et 7 colonnes pour les différents indicateurs créés par `summary()`. Il est nécessaire de transposer la première table obtenue, puis de l'ordonner.

```
summarydistI <- apply(matdistHop, 1, summary, na.rm = TRUE)
summarydistI2 <- t(summarydistI)
summarydistI3 <- summarydistI2[order(rownames(summarydistI2)), ]
```

Lorsque l'on a un grand nombre d'objets, il peut être intéressant de travailler sur la matrice de distance dans un autre format : un vecteur $n^2 \times 3$ donnant pour l'ensemble des couples (i, j) la valeur $d(i, j)$ associée en 3 colonnes. On l'obtient avec la fonction `melt()` du `package reshape` (cf. Section 2.3). Dans cette nouvelle table, les variables s'appellent par défaut : `X1`, `X2`, `value`.

```
coupledistHop <- melt(matdistHop)
mindistbis <- tapply(coupledistHop$value,
                     as.factor(coupledistHop$X1),
                     min,
                     na.rm=TRUE)
```

Démarche en utilisant le *package* `spdep` 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. Ici $k = 1$. Si l'on travaille sur un objet de type *SpatialPoint*, la fonction récupère directement l'ensemble des informations nécessaires aux calculs et travaille sur le vecteur textttt$coords. La proximité est ici entendue au sens de la distance euclidienne. Si on spécifie l'argument `longlat=TRUE`, la fonction fait le calcul sur les coordonnées, comme si c'était des longitudes/latitudes en degré décimaux ; la distance est exprimée en kilomètres.

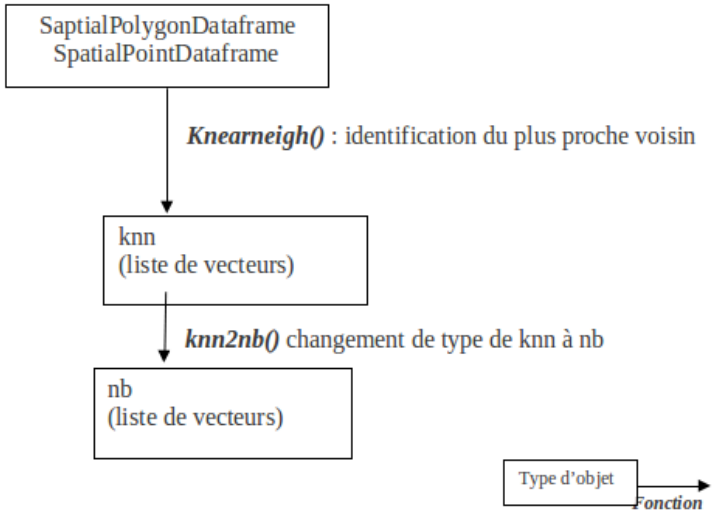
```
lppv <- knearneigh(sdHopito@coords, k = 1)
str(lppv)

## List of 5
##  $ nn       : int [1:64, 1] 5 8 4 3 7 7 5 2 10 11 ...
##  $ np       : int 64
##  $ k        : num 1
##  $ dimension: int 2
##  $ x        : num [1:64, 1:2] 601714 611037 606860 606380 601575 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:64] "310" "309" "308" "307" ...
##   .. ..$ : chr [1:2] "coords.x1" "coords.x2"
##  - attr(*, "class")= chr "knn"
##  - attr(*, "call")= language knearneigh(x = sdHopito@coords, k = 1)
```

L'objet `lppv` est une liste de type *knn* dont le premier élément est le vecteur des plus proches voisins.

```
summary(lppv)
```

```
##           Length Class  Mode
## nn         64    -none- numeric
## np          1    -none- numeric
## k           1    -none- numeric
## dimension   1    -none- numeric
## x         128    -none- numeric
```

Ce schéma résume les différentes fonctions et leur usages en fonction des objets traités avec cette seconde méthode.



10.1.3 Visualisations d'autres voisinages

Description du semis sur la base des 5 plus proches voisins On peut décider de travailler non pas sur le plus proche voisin, mais sur les 5 plus proches. On utilisera la même fonction `knearneigh()` pour construire la liste des 5 plus proches.

```
lppv5 <- knearneigh(sdHopito@coords, k = 5)
```

On obtient une liste dont le premier élément est pour chaque hôpital la liste des cinq hôpitaux les plus proches. On calcule ensuite la distance entre chaque hôpital et ses 5 plus proches voisins en utilisant cette liste de type `nn` et le vecteur `@coords` contenant les coordonnées des hôpitaux. La fonction `knearneigh()` ne renvoie pas la matrice des distances. Pour ce faire, il faut utiliser la fonction `nbdists()` qui nécessite de travailler à partir d'un objet de type *nb*. Il faut donc au préalable transformer cet objet `lppv5` par la fonction `knn2nb()`.

```
lppv5b <- knn2nb(lppv5, row.names = sdHopito$CodHop)
str(head(lppv5b))
```

```
## List of 6
##  $ : int [1:5] 5 6 7 42 55
##  $ : int [1:5] 3 8 9 10 11
##  $ : int [1:5] 4 6 11 43 59
##  $ : int [1:5] 3 6 7 43 59
##  $ : int [1:5] 1 6 7 42 55
##  $ : int [1:5] 1 3 4 5 7
```

```
summary(lppv5b)
```

```
## Neighbour list object:
## Number of regions: 64
## Number of nonzero links: 320
## Percentage nonzero weights: 7.812
## Average number of links: 5
## Non-symmetric neighbours list
## Link number distribution:
##
##  5
## 64
## 64 least connected regions:
## 310 309 308 307 306 305 304 303 302 301 300 299 298 297 296 295 294 293 292 291 290 289 288 287 286
## 64 most connected regions:
## 310 309 308 307 306 305 304 303 302 301 300 299 298 297 296 295 294 293 292 291 290 289 288 287 286
```

On remarquera que la fonction `summary()` appliquée à un objet « liste de voisins » donne des indications sur la connectivité : les hopitaux sont considérés au travers du graphe associé à la relation « est plus proche voisin de ».

On calcule ensuite les distances à partir de cette liste.

```
distppv5 <- nbdists(lppv5b, sdHopito@coords)
str(head(distppv5))
```

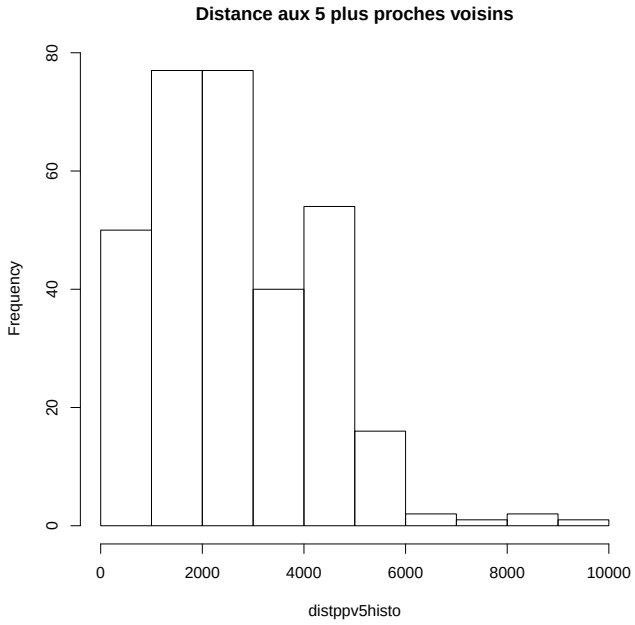
```
## List of 6
##  $ : num [1:5] 1714 2896 2521 4206 4564
##  $ : num [1:5] 8488 4130 4450 4735 5356
##  $ : num [1:5] 636 2954 3414 3307 3589
##  $ : num [1:5] 636 2881 3540 2700 2962
##  $ : num [1:5] 1714 3013 1453 2579 2867
##  $ : num [1:5] 2896 2954 2881 3013 2022
```

En utilisant la fonction `unlist()`, on crée une liste de l'ensemble de ces distances. On obtient une nouvelle variable décrivant tous les couples d'hôpitaux voisins au sens de « 5 plus proches voisins ». On peut calculer le minimum, le maximum et la moyenne.

```
distppv5histo <- unlist(nbdists(lppv5b, sdHopito@coords))
summary(distppv5histo)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0    1550    2450    2700    3890    9590
```

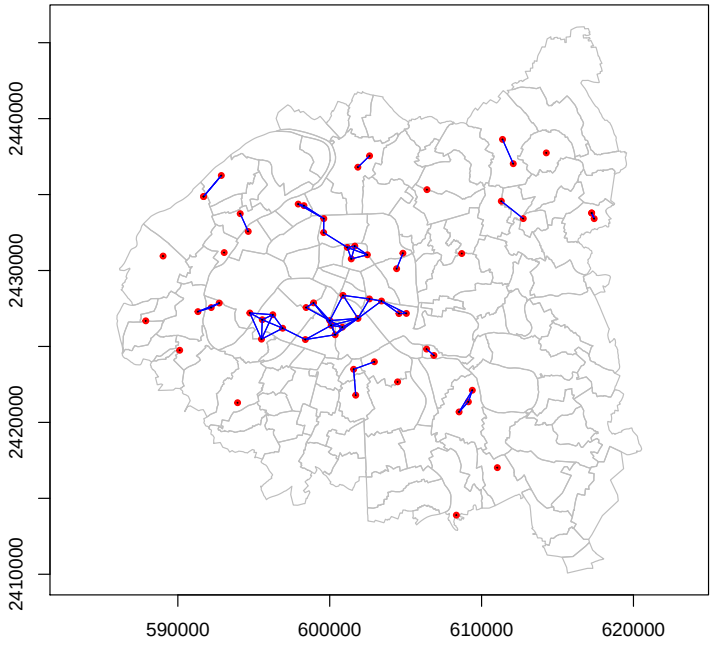
```
hist(distppv5histo, main = "Distance aux 5 plus proches voisins")
```



Sélectionner les couples en fonction d'une distance On peut aussi avoir la démarche inverse, et définir le voisinage en absolu (distance $<$ seuil) plutôt qu'en relatif (plus proche). On peut par exemple identifier les couples d'hôpitaux qui sont à moins de 2 km et les visualiser par le programme suivant :

```
hopi2km_nb <- dnearneigh(sdHopito@coords, d1 = 0, d2 = 2000, row.names = sdHopito$CodHop)

str(hopi2km_nb)
plot(sdCommunes, cex = 0.1, axes = T, border = "grey")
plot(sdHopito, col = "red", add = T, pch = 20)
plot(hopi2km_nb, sdHopito@coords, pch = 20, cex = 0.1, col = "blue", add = TRUE)
```

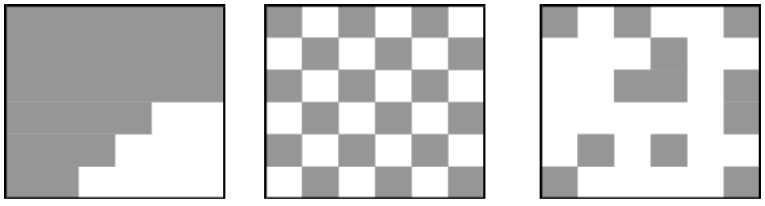
Les dernières instructions permettent de cartographier les « liens » identifés dans $hopi2km\_nb$: les hôpitaux qui sont à moins de 2km sont reliés par un lien.



10.2 Mesure de l'autocorrélation spatiale et des ressemblances locales

10.2.1 Introduction aux notions d'autocorrélation spatiale

L'autocorrélation spatiale est au cœur de la première loi en géographie telle que l'énonçait W. Tobler : « everything is related to everything else, but near things are more related than distant things » [Tobler 1970]. L'autocorrélation spatiale peut être mesurée. A l'origine, les mesures ont été développées pour des phénomènes observés sur des maillages territoriaux.



Autocorrélation
L'apparition en un lieu dépend de ce qui se passe dans les lieux voisins

Absence d'autocorrélation

Positive
2 lieux proches se ressemblent plus que 2 lieux éloignés

Négative
2 lieux proches se ressemblent moins que 2 lieux éloignés

(d'après [Pumain et Saint-Julien 1997])

Il s'agit ainsi, pour un phénomène donné, de 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. Par exemple pour une variable X, la variance totale s'écrit :

$$var_T(X) = \frac{1}{n}\sum_i (X_i - \bar{X})^2 = \frac{1}{2 \times n \times (n-1)} \sum_i \sum_{i'} (X_i - X_{i'})^2$$

La variance locale, par analogie, s'écrira :

$$var_L(X) = \frac{1}{2K} \sum_i \sum_{i' \in v(i)} (X_i - X_{i'})^2$$

où $v(i)$ désigne un voisinage de i et K le nombre de couples (i, i') voisins.

Les premiers développements concernaient des voisinages définis par la contiguïté sur un maillage. La variance locale peut alors s'écrire comme une variance totale pondérée par un poids W affecté à chacun des couples (i, i') : $wii' = 1$ si i et i' sont des zones contiguës, $= 0$ sinon. Ce système peut être étendu à la notion de voisinage au sens large. On peut ainsi considérer comme voisines des unités distantes de moins de 5km, de moins d'une heure, appartenant à la même région administrative...Dans ce cas : $wii' = 1$ si i et i' sont des zones voisines, $= 0$ sinon. En anglais, la matrice W s'appelle une matrice de « spatial weights » que l'on peut traduire par matrice de voisinage.

Nous présentons très brièvement les deux indices d'autocorrélation les plus utilisés, en renvoyant le lecteur aux ouvrages cités pour de plus amples développements sur ces questions. L'indice de Geary, fondé sur la variance locale et l'indice de Moran fondé sur la covariance locale.

$$\text{Indice de Geary : } C = \frac{(N-1)[\sum_i \sum_j W_{i,j}(X_j - \bar{X})]}{2(\sum_i \sum_j W_{i,j}) \sum_i (X_i - \bar{X})^2}$$

$$\text{Indice de Moran : } I = \frac{N \sum_i \sum_j W_{i,j}(X_i - \bar{X})(X_j - \bar{X})}{(\sum_i \sum_j W_{i,j}) \sum_i (X_i - \bar{X})^2}$$

Avec N le nombre d'individus, X_i et X_j valeur de la variable X en i et j, $\bar{X}$ est la moyenne de la variable X et $W_{i,j}$ est une pondération illustrant la notion de voisinage (i, j).

L'indice de Moran est souvent préféré car il varie comme un cœfficient de corrélation tandis que l'indice de Geary est toujours positif et une faible valeur signale une forte autocorrélation spatiale. On travaille d'ailleurs en général avec l'indice 1 - C.

Par ailleurs, comme pour un cœfficient de corrélation, on doit tester la significativité de ces cœfficients. Par exemple pour l'indice de Moran, on construira la statistique Z :
$Z(I) = (I - E(I)/S(E(I))$ qui suit une loi normale de paramètres (0,1).

où $E(I)$ est l'espérance de la variable I,
$S(E(I))$ est son erreur standard et se calcule ainsi $S(E(I)) = sqrt(S1/S2)$
avec

$$S1 = N^2 \sum_{i,j} W_{i,j} + 3 \times (\sum_{i,j} W_{i,j})^2 - N \times \sum_i \sum_j W_{i,j}^2$$

et

$$S2 = (N^2 - 1)(\sum_{i,j} W_{i,j})^2$$

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 (Local Index of Spatial Autocorrelation, LISA, voir [Anselin 1995])
L'indice local de Moran correspond à la contribution de chaque unité à l'autocorrélation globale. Il s'écrit :

$$I_i = \frac{(X_i - \bar{X})[\sum_{j \in V(i)} W_{i,j}(X_j - \bar{X})]}{var(X)}$$

Ainsi on peut 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.

10.2.2 L'exemple des cadres et des ouvriers

On va chercher à identifier les zones de fortes concentrations de ces deux catégories dans la région étudiée. La fonction `moran.test()` du *package* `spdep` travaille directement sur les objets de type `nb` qui donnent la liste des unités « en relation » que l'on peut appeler aussi graphe de relation ou de voisinage. Si l'on souhaite travailler sur un graphe de contiguité, il faut au préalable transformer l'objet *SpatialPolygonDataFrame* en ce type de liste. On utilise la fonction `poly2nb()` en spécifiant les identifiants. L'option `Queen` (en rappel au déplacements des pièces du jeu d'échecs) détermine si l'on considère que un point commun suffit pour que 2 zones soient contigues (`TRUE`) ou s'il faut plusieurs points (`FALSE`). L'option `snap` permet de gérer la résolution (distance en dessous de laquelle deux points d'un polygone sont considérés superposés).

```
nbCom <- poly2nb(pl = sdCommunes, row.names = sdCommunes$DEPCOM, snap = 50,
    queen = TRUE)
# str(nbCom)
moran.test(x = data99_07$PCAD99, listw = nb2listw(nbCom))


##
##  Moran's I test under randomisation
##
## data:  data99_07$PCAD99
## weights: nb2listw(nbCom)
##
## Moran I statistic standard deviate = 14.26, p-value < 2.2e-16
## alternative hypothesis: greater
## sample estimates:
## Moran I statistic       Expectation          Variance
##          0.735489         -0.007042          0.002712

moran.test(x = data99_07$PCAD07, listw = nb2listw(nbCom))


##
##  Moran's I test under randomisation
##
## data:  data99_07$PCAD07
## weights: nb2listw(nbCom)
##
## Moran I statistic standard deviate = 14.59, p-value < 2.2e-16
## alternative hypothesis: greater
## sample estimates:
## Moran I statistic       Expectation          Variance
##          0.753495         -0.007042          0.002717

moran.test(x = data99_07$POUV99, listw = nb2listw(nbCom))


##
##  Moran's I test under randomisation
##
## data:  data99_07$POUV99
## weights: nb2listw(nbCom)
##
## Moran I statistic standard deviate = 13.02, p-value < 2.2e-16
## alternative hypothesis: greater
## sample estimates:
## Moran I statistic       Expectation          Variance
##          0.671305         -0.007042          0.002713

moran.test(x = data99_07$POUV07, listw = nb2listw(nbCom))
```

```
##
##  Moran's I test under randomisation
##
## data:  data99_07$POUV07
## weights: nb2listw(nbCom)
##
## Moran I statistic standard deviate = 13.43, p-value < 2.2e-16
## alternative hypothesis: greater
## sample estimates:
## Moran I statistic       Expectation          Variance
##          0.691736         -0.007042          0.002709
```

En prenant exemple sur l'analyse de la proportion d'ouvriers dans les communes en 2007, les sorties permettent d'identifier :

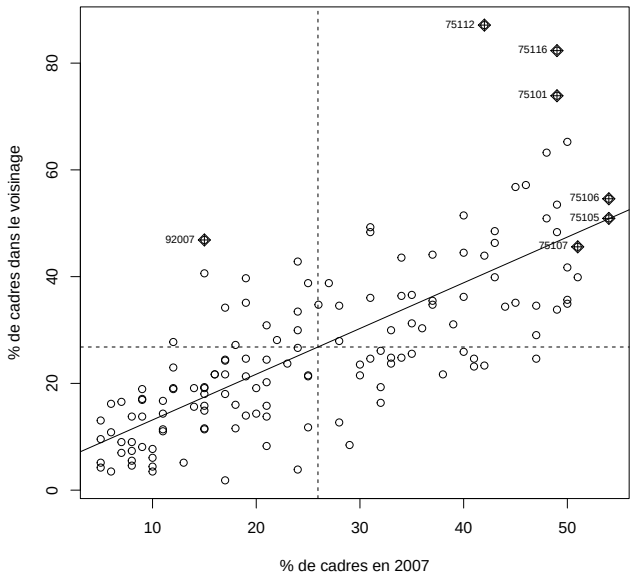
- la valeur de l'autocorrélation (0.69) ;
- la valeur de l'écart standardisé (Moran I statistic standard deviate = 13.42) ;
- la probabilité de dépasser cette valeur sous l'hypothèse où il n'y a pas d'autocorrélation (ici $2.2e^{-16}$).

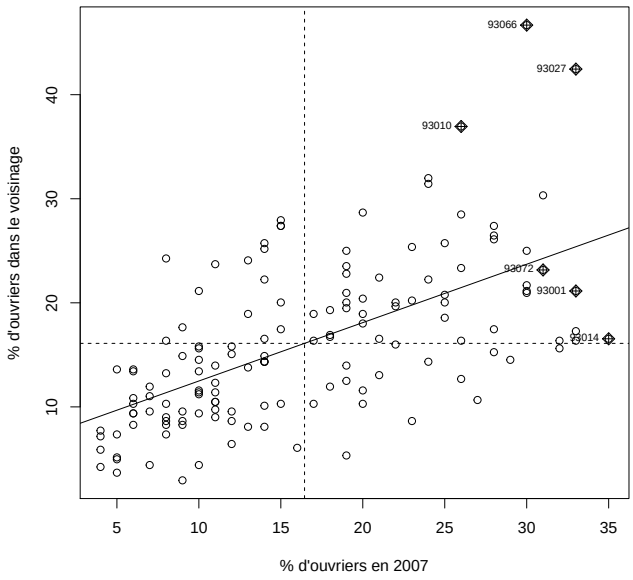
On peut donc conclure que la valeur 0.69 illustre une situation significative d'une autocorrélation spatiale globale positive.

La proportion de cadres et la proportion d'ouvriers présentent une autocorrélation spatiale positive, synonyme dans ce cas de ségrégation dans l'espace. Cette configuration est stable pour les cadres entre 1999 et 2007 (0.75) et quasi-stable pour les ouvriers (0.67 en 1999 ; 0.69 en 2007).

Comme pour l'analyse des corrélations, on peut représenter la relation dans l'espace grâce à un nuage de points croisant les valeurs en i avec les valeurs au voisinage de i pour tous les couples d'unités voisines.

```
moran.plot(x=data99_07$PCAD07, listw=nb2listw(nbCom, style="C"),
           xlab="% de cadres en 2007",
           ylab="% de cadres dans le voisinage")
moran.plot(x=data99_07$POUV99, listw=nb2listw(nbCom, style="C"),
           xlab="% d'ouvriers en 2007",
           ylab="% d'ouvriers dans le voisinage")
```





La figure représente le nuage de points croisant les valeurs en un point et les valeurs moyennes au voisinage. La droite représente le modèle de régression des valeurs du voisinage en fonction des valeurs des individus en faisant l'hypothèse d'une autocorrélation spatiale significative. Les individus qui contribuent significativement à la pente de la droite sont identifiés et représentés en noir.

Conclusion

L'intégration d'objets spatiaux dans R permet de construire des objets de voisinage et de rentrer dans une démarche d'analyse spatiale, exploratoire ou confirmatoire, intégrant les notions de voisinage. Nous donnons ici une bibliographie pour tou-te-s celles et ceux qui veulent approfondir les aspects méthodologiques. Pour approfondir les aspects de traitement sous R, nous les renvoyons à l'ouvrage très complet de [Bivand *at al.* 2008] qui est l'auteur du *package* `spdep`.

10.3 Code

```
##### Chargement des packages et données #####
# Chargement des packages
require('maptools', quietly=TRUE)
require('rgdal', quietly=TRUE)
require('rgeos', quietly=TRUE)
require('reshape', quietly=TRUE)
require('spdep', quietly=TRUE)

# Lecture des fichiers communes
sdCommunes <- readShapePoly('data/paripc_com_region.shp')
class(sdCommunes)
summary(sdCommunes)
plot(sdCommunes, axes=T)
sdCommunes$DEPCOM <- as.character(sdCommunes$DEPCOM)

# Lecture des fichiers hôpitaux
sdHopito <- readOGR("data/paripc_hopitaux_fin.TAB",
                    layer="paripc_hopitaux_fin",
                    input_field_name_encoding="latin1")
plot(sdHopito,col="red",add=T)
rownames(sdHopito@data) <- as.character(sdHopito$CodHop)

# Lecture des fichiers statistiques pour les communes
data99_07 <- read.csv("data/data99_07.csv",
                      sep=";",
                      stringsAsFactor=FALSE)
rownames(data99_07) <- as.character(data99_07$CODGEO)

##### Espacements d'un semis de points #####

str(head(sdHopito))
str(head(data99_07))
row.names(sdHopito) <- as.character(sdHopito$CodHop)
coordsHop <- coordinates(sdHopito)
Distppvtheo <- 0.5 * sqrt(sum(data99_07$SURF) * 10000 / nrow(sdHopito))

### Démarche générique ###
distHop <- dist(coordsHop, method="euclidean", diag=FALSE)
str(distHop)

matdistHop <- as.matrix(distHop)
str(matdistHop)
matdistHop

matdistHop[matdistHop==0] <- NA
mindistHI <- apply(matdistHop, 1, min, na.rm=TRUE)
mean(mindistHI)
RIndex <- mean(mindistHI) / Distppvtheo
RIndex

summarydistI <- apply(matdistHop, 1, summary, na.rm=TRUE)
summarydistI2 <- t(summarydistI)
summarydistI3 <- summarydistI2[order(rownames(summarydistI2)), ]

coupledistHop <- melt(matdistHop)
mindistbis <- tapply(coupledistHop$value,
                     as.factor(coupledistHop$X1),
```

```
                    min,
                    na.rm=TRUE)

### Avec spdep ###

lppv <- knearneigh(sdHopito@coords, k=1)
str(lppv)
summary(lppv)
lppv5 <- knearneigh(sdHopito@coords, k=5)
lppv5b <- knn2nb(knearneigh(sdHopito@coords, k=5),
                 row.names=sdHopito$CodHop)
str(head(lppv5b))
summary(lppv5b)

distppv5 <- nbdists(lppv5b, sdHopito@coords)
str(head(distppv5))

distppv5histo <- unlist(nbdists(lppv5b, sdHopito@coords))
summary(distppv5histo)
hist(distppv5histo, main="Distance aux 5 plus proches voisins")

hopi2km_nb <- dnearneigh(sdHopito@coords, d1=0, d2=2000,
                         row.names=sdHopito$CodHop)
str(hopi2km_nb)
plot(sdCommunes, cex=0.1, axes=T, border="grey")
plot(sdHopito,col="red", add=T, pch=20)
plot(hopi2km_nb, sdHopito@coords, pch=20,
     cex=0.1, col="blue", add=TRUE)

plot(sdCommunes, cex=0.1, axes=T, border="grey")
plot(sdHopito,col="red", add=T, pch=20)
plot(hopi2km_nb, sdHopito@coords, pch=20,
     cex=0.1, col="blue", add=TRUE)

##### Mesure de l'autocorrélation spatiale #####
nbCom <- poly2nb(pl=sdCommunes,
                 row.names=sdCommunes$DEPCOM,
                 snap=50, queen=TRUE)
str(nbCom)
moran.test(x=data99_07$PCAD99, listw=nb2listw(nbCom))
moran.test(x=data99_07$PCAD07, listw=nb2listw(nbCom))
moran.test(x=data99_07$POUV99, listw=nb2listw(nbCom))
moran.test(x=data99_07$POUV07, listw=nb2listw(nbCom))

moran.plot(x=data99_07$PCAD07, listw=nb2listw(nbCom, style="C"),
           xlab="% de cadres en 2007",
           ylab="% de cadres dans le voisinage")
moran.plot(x=data99_07$POUV99, listw=nb2listw(nbCom, style="C"),
           xlab="% d'ouvriers en 2007",
           ylab="% d'ouvriers dans le voisinage")
```

Bibliographie

[Anselin 1995] Anselin L. (1995) "Local indicators of spatial association - LISA", *Geographical analysis*, 27/2, pp. 93-115.

[Baath 2012] Baath R. (2012) "The state of naming conventions in R", *The R Journal*, Vol. 4/2, pp.74-75.

[Bivand *at al.* 2008] Bivand R.S., Pebesma E.J., Gómez-Rubio V. (2008) *Applied spatial data analysis with R*, Springer.

[Bonnel 2002] Bonnel P. (2002) *Prévision de la demande de transport*, HDR, Université Lumière - Lyon II.

[Groupe fmr] Groupe fmr (2010-2013) *Synthèses*, http ://halshs.archives-ouvertes.fr/FMR/.

[Husson *at al.* 2009] Husson F., Lê S., Pagès J. (2009) *Analyse de données avec R*, Presses Universitaires de Rennes.

[Lazéga 2007] Lazéga E. (2007) *Réseaux sociaux et structures relationnelles*, Paris, PUF.

[Lebart *at al.* 1995] Lebart L., Morineau A., Piron M. (1995) *Statistique exploratoire multidimensionnelle*, Dunod.

[Muenchen 2008] Muenchen R. (2008) *R for SAS and SPSS users*, Springer.

[Muenchen 2012] Muenchen R. (2012) *R for Stata users*, Springer.

[Newman 2010] Newman M. (2010) *Networks : an introduction*, Oxford, Oxford University Press.

[Pumain et Saint-Julien 1997] Pumain D., Saint-Julien Th. (1997) *L'analyse spatiale*, Paris, Armand Colin.

[Sanders 1989] Sanders L. (1989) *L'analyse statistique des données en géographie*, Montpellier, Alidade-RECLUS.

[Wasserman et Faust 1994] Wasserman S., Faust K. (1994), *Social network analysis. Methods and applications*, Cambridge, Cambridge University Press.

[Zaninetti 2005] Zaninetti J.-M. (2005) *Statistique spatiale : méthodes et applications géomatiques*, Hermès Science Publications.

[Tobler 1970] Tobler W. (1970), "A computer movie simulating urban growth in the Detroit region", *Economic geography*, 46, pp. 234-240.