Data Processing and Analysis with R Language

Keita, Moussa

March 2017

Traitement et Analyse de Données avec le Langage R

Par
Moussa Keita, PhD
Consultant Big Data-Data Science
Umanis Consulting Group, Paris

Mars 2017
(Version 1.0)

Résumé

Depuis quelques années, nous assistons à une véritable montée en puissance des logiciels « open-sources » s'imposant peu à peu comme des alternatives solides aux logiciels « propriétaires » dans la gestion des données volumineuses. Parmi cette panoplie de logiciels libres figure le logiciel R qui se présente à la fois comme un logiciel de programmation pure mais aussi comme un logiciel de traitements et d'analyses de données statistiques. Ce manuscrit vise à proposer une introduction au langage R dans une aproche fondée exclusivement sur des exemples pratiques. Il est organisé en cinq chapitres. Le premier chapitre présente les concepts de base de la programmation avec le langage R. Le second chapitre présente les différentes méthodes de traitements et d'organisation des données. Le troisième chapitre est consacré à la mise en œuvre des méthodes d'analyses statistiques et économétriques classiques. Quant au quatrième chapitre, il est dédié à la présentation des méthodes de visualisation graphique des données. Le dernier chapitre présente les méthodes d'analyse des séries temporelles notamment les méthodes de désaisonnalisation, de filtres et lissages linéaires, de lissages expoentiels mais aussi les modélisations et prévisions par les méthodes stochastiques (AR, MA, ARMA, ARIMA et SARIMA). Le manuscrit étant toujours en progression, nous restons ouverts à toutes les critiques et suggestions de nature à améliorer son contenu.

*Ecole d'Economie, Université d'Auvergne Clermont Ferrand 1 ;
Ecole Nationale Supérieure de Statistique et d'Economie Appliquée ENSEA-C.I
Contact info: Email : keitam09@ymail.com
Codes JEL: C8
Mots clés: Programmation, langage R, traitement de données, analyses de données, data visualization.

TABLE DES MATIERES

Chapitre 1 : Les bases de la programmation sous R

Dans ce premier chapitre, nous passons en revue les différents concepts de programmation sous R. La présentation sera centrée sur la description des principaux objets de programmation qui forment le langage R à savoir : les vecteurs, les matrices, les tableaux de valeurs, les tables de données, les expressions ainsi que les fonctions. Une discussion spéciale sera aussi menée sur les variables en chaîne de caractères et sur les variables date.

1.1. Expressions mathématiques simples et opérateurs standards

R est une calculatrice avancée. Il permet d'effectuer toutes les opérations mathématiques valides exprimées sous forme d'expressions.

Exemples : Taper dans le console les opérations suivantes :

```
5+7 # renvoie 12
3^2+7 # renvoie 16
3^(2+ 7) # renvoie 19683
8/2+(8+3) # renvoie 15
cos(3*pi/4) # renvoie -0.7071068
exp(2*3) # renvoie 403.4288
log(45) # renvoie 3.806662
```

On remarque à travers les résultats que R respecte la priorité des opérateurs

Le tableau ci-dessous présente les opérateurs standards sous R ainsi que quelques exemples d'application.

Symbole opérateur	Description	Exemples
Opérateurs arithmétiques et algébriques		
+	addition,	`2+5`
-	soustraction	`11-3`
*	multiplication	`4*5`
/	division	`15/3`
^	puissance	`5^2`
%*%	produit matriciel	`A%*%B`
%%	modulo (renvoie le reste de la division)	`215 %% 11`
%/%	Partie entière d'une division	`23 %% 6`
Opérateurs logiques		
<	inférieur à	`2<3`
<=	inférieur ou égal à	`2<=3`
==	égal	`4*2==16/2`
>	supérieur à	`3>2`
> =	supérieur ou égal à	`3>=2`

!=	différent de	`6/3 != 7/2`
!	négation logique	`! x==y`
&, &&	l'opérateur logique "et"	`x==2 & y==5 / x==2`
|, ||	l'opérateur logique "ou"	`x==2 | y==5 / x==2`
xor()	Opérateur logique x ou	`xor(a,b)`
any()	Opérateur logique : chaque	`any(c(2,6,7)<5)`
all()	Opérateur logique : tout	`all(c(2,6,7)<5)`
%in%	Teste la présente d'un élément dans une	`4 %in% c(2,6,7)`
Opérateurs de sélection et d'indiçage		
:	intervalle de séquence de valeurs	`4 :12`
$	Opérateur de référencement d'éléments	`x$y`
Opérateurs d'assignation		
<- , <<-	assignation par la gauche	`x<-2 / x<<-2`
->, ->>	assignation par la droite	`2->x / 2->>x`

1.2. Définition de variables et assignation de valeurs

1.2.1. Définition de variables

Sous R, une variable se définit par assignation directe en utilisant l'une des trois formes suivantes:

- assignation par la gauche : <-
- assignation par la droite : ->
- utilisation de la fonction assign().

Exemples :

Assignation par la gauche:

```
a<- 5 # définit une variable nommée « a » en lui assignant la valeur 5
print(a)
```

Assignation par la droite:

```
7-> b # définit une variable nommée « b » en lui assignant la valeur 7
print(b)
```

Utilisation de la fonction assign()

```
assign('c',11)   # définit une variable nommée « c » en lui assignant la valeur 11
print(c)
```

Attention à l'utilisation des quotes qui entourent le nom de la variable

Noter aussi qu'on peut définir une variable à partir d'autres variables.

Exemples:

```
a<-7 # définit la variable « a » qui prend la valeur 7
b<- 2*a+5 #  définit la variable « b » comme le double de « a » auquel on ajoute 5.
print(b)
```

1.2.2. Regroupement de plusieurs instructions d'assignation

On peut regrouper plusieurs commandes en une seule expression en les entourant d'accolades { }.

Exemple :

```
{
a <- 2 + 3
b <- a
b
}
```

Par défaut, le résultat renvoyé par le regroupement des instructions est la valeur de la dernière instruction. Dans l'exemple ci-dessus, seul le résultat de la ligne de commande b est renvoyée.

Mais si le regroupement se termine par une assignation (voir exemple ci-dessous), aucune valeur n'est retournée ni affichée à l'écran :

```
 {
a <- 2 + 3
b <- a
}
```

Le regroupement de commande est surtout utile dans le cadre de la définition de fonctions (nous y reviendrons).

NB : On peut spécifier plusieurs instructions sur la même ligne en utilisant le symbôle « ; ». Exemple :

```
a <- 2 + 3 ; b <- a ; print(a) ; print(b)
```

1.2.3. Quelques règles sur le choix du nom de la variable

Ci-dessous quelques règles pour le choix du nom des variables sous R.

1.2.3.1. Choix des caractères

- Sous R, le nom d'une variable doit être composé uniquement de lettres minuscules et majuscules a-zA-Z, de chiffres 0-9, le point « _ » et l'underscore « _ ».
- Aucun autre caractère spécial n'est autorisé.
- Les noms ne peuvent commencer par un chiffre.

- Si le nom commence par un point, celui-ci ne peut être suivi par un chiffre.

1.2.3.2. Choix de la composition du nom

Pour la composition du nom d'une variable, on peut choisir l'une des formes suivantes :

- Tous les caractères en minuscule : myvariable ;
- séparation par un point : my.variable ;
- séparation par un underscore : my_variable ;
- première lettre en minuscule et lettres de transition en majuscule : myVariable
- première lettre en majuscule : MyVariable.

1.2.3.3. Choix de la casse

- Attention à la casse de la variable: myvar ≠ mYvar

- Eviter d'utiliser les mots réservés. Ex: diff, length, mean, pi, range, var, break, else, for, function, if, in, next, repeat, return, while, TRUE, FALSE, Inf, NA, NaN, NULL, NA_integer_, NA_real_, NA_complex_, NA_character_, etc...

- Pour vos programmes personnels, utiliser plutôt des noms composés tels que: myvar, mydata, myfunction. Cela permet à un débutant de distinguer facilement vos objets des autres objets de R.

1.2.4. Les attributs d'une variable

1.2.4.1. Le mode

Le mode d'une variable (d'un objet) est le type de valeur que prend cette variable (cet objet). Pour connaitre le type d'un objet, on utiliser la fonction mode().

Exemples:

```
x<-3
mode(x) # renvoie  "numeric"
y<-'Je teste'
mode(y) # Renvoie "character"
```

On peut aussi utiliser la fonctin typeof()

```
typeof(x) #Renvoie "double"
typeof(y) # Renvoie "character"
```

Les principaux modes disponibles sous R sont:

- numeric (nombres réels)
- complex (nombres complexes)
- logical (valeurs booléennes TRUE et FALSE)

- character (chaînes de caractères)
- function (objet fonction)
- list (objet pouvant contenir des données quelconques)
- expression (formules non évaluées)

1.2.4.2. Détails sur les valeurs numériques (numeric)

On distingue deux types de mode numérique, à savoir les integers (entiers) et les double ou real (réels). Le type double est le type de base des valeurs numériques sous R. Mais on peut le convertir en d'autres types.

Exemples:

```
a <- 2.0 # valeur de type double
typeof(a) ## [1] "double"
is.integer(a) # Teste si a est un réel. Renvoie FALSE
b <- 2
typeof(b) ## [1] "double"
c <- as.integer(b) # Convertit en type integer
typeof(b) ## [1] "integer"
is.numeric(c) # Teste si c est bien un numérique. Renvoie TRUE
```

Les valeurs numériques spéciales sous R: les valeurs NULL, NA, Inf, -Inf et NaN

L'objet NULL représente est un objet dont le contenu est vide. Il ne doit pas être confondu ni avec une valeur manquante, ni avec la valeur 0

```
a<-NULL # Crée la variable a et lui attribue la valeur NULL
```

La fonction is.null() teste si un objet est NULL ou non. Ex :

```
x<-2
is.null(x) #Renvoie FALSE
y<-NULL
is.null(y) #Renvoie TRUE
```

La valeur NA représente une valeur manquante en particulier dans les données numériques.

Pour tester si les éléments d'un objet sont NA ou non, on utilise la fonction is.na() . Exemple:

```
x<-2
is.na(x) #Renvoie FALSE
y<-NA
is.na(y) #Renvoie TRUE
```

Les valeurs infinies sont représentées par les mots Inf et -Inf qui désigent respectivement + l'infini et - l'infini.

Pour tester si une valeur est infinie, on utilise la fonction infinite(). Exemple:

```
x<-2
is.infinite(x) #Renvoie FALSE
y<- Inf
is.infinite(y) #Renvoie TRUE
z<- -Inf
is.infinite(z) #Renvoie TRUE
```

Les valeurs indéterminées sont représentée par le symble NaN(). Le mot NaN signifie Not a Number.

Pour determiner si une valeur NaN, on utilise la fonction is.nan() comme suit:

```
x<-2
is.nan(x) #Renvoie FALSE
y<-NaN
is.nan(y) #Renvoie TRUE
```

1.2.4.3. Détails sur les valeurs en caractères (character)

Les chaînes de caractères sont placées entre les guillemets simples ' ou doubles ".

```
a <- "Hello world!"
typeof(a) # "character"
```

1.2.4.4. Détails sur les valeurs logiques ou booléennes (logical)

Les données de type logique peuvent prendre deux valeurs : TRUE ou FALSE. Elles répondent à une condition logique. Exemple:

```
a <- 1 ; b <- 2
a < b # Teste si a est inférieur à b. renvoie [1] TRUE
a == 1 # Teste si a est égale à 1. TRUE
a != 1 # Test d'inégalité [1] FALSE
is.character(a) # renvoie [1] FALSE
(a <- TRUE) # renvoie [1] TRUE
(a <- T) # Renvoie [1] TRUE
```

Comme le montre l'exemple ci-dessus, il est possible d'abréger TRUE en T ; il en est de même pour FALSE, qui peut s'abréger en F.

Il peut parfois être pratique d'utiliser le fait que TRUE peut être automatiquement converti en 1 et FALSE en 0. Ex:

```
x<-TRUE + TRUE + FALSE + TRUE*TRUE
print(x) # renvoie 3
```

1.2.4.5. Détails sur les variables de type complexe (complex)

Les nombres complexes sont caractérisés par la présence d'une partie réelle et d'une partie imaginaire. On crée un nombre imaginaire à l'aide de la lettre i.

Exemples :

```
x <- 2+3i # définit nombre complexe
y <- 2 # nombre complexe de partie imaginaire nulle
z <- 3i # nombre complexe de partie réelle nulle
```

On peut obtenir la partie réelle d'un nombre complexe à l'aide de la fonction Re() ; et par leur partie imaginaire, que l'on obtient grâce à la fonction Im(). Pour calculer le module et l'argument d'un nombre complexe, on utilise respectivement Mod() et Arg()

Exemples:

```
x <- 2+3i # définit nombre complexe
print(x)
print(y)
print(z)
print(Im(x))
print(Re(x))
print(Mod(x))
print(Arg(x))
```

1.2.4.6. Détails sur les valeurs de type "expression"

Une expression est généralement une formule définie à partir d'une combinaison d'autres variables. Exemple: soient les variables x, y et z définies comme suit:

```
x <- 3; y <- 2.5; z <- 1
```

On peut définir une expression comme suit:

```
myexp1<- expression(x /(y + exp(z)))
```

En imprimant myexp1, R affiche uniquement la formulation de l'expression

```
print(myexp1)
```

Si l'on souhaite voir la valeur contenue dans l'expression, on utilise la fonction eval() qui évalue les expression

```
eval(myexp1)
```

Différentes opérations mathématiques peuvent être effectuées sur les expressions définies. Par exemple, on peut calculer les dérivées partielles lorsqu'il s'agit d'une expression mathématique de type fonction. Pour cela, on utilise la fonction D().

Exemple: les dérivées de myexp1 par rapport à x, y et z sont respectivement:

```
deriveX<-D(myexp1, "x")
print(deriveX)
deriveY<-D(myexp1, "y")
print(deriveY)
deriveZ<-D(myexp1, "z")
print(deriveZ)
```

Evaluation des dérivées

```
eval(deriveX)
eval(deriveY)
eval(deriveZ)
```

1.2.5. Longueur d'une variable et nombre de caractères d'une variable: fonction length() et fonction nchar()

La longueur d'une variable ou d'un objet est égale au nombre d'éléments distincts que cet objet contient.

Attention le terme « éléments distincts » ne signifie pas « nombre de caractères qui constitue l'objet » mais plutôt le nombre d'éléments entier formant une séquence (ou en liste). Exemples :

```
x<-2
length(x) renvoie 1
y<-'Mon texte'
length(y) # renvoie 1
z<-c(2, 3, 7)
length(z) #. renvoie 3
```

Pour connaitre **le nombre de caractères** dans une chaîne on utilise la fonction nchar(). Exemple:

```
x<-'Mon texte'
nchar(x) # renvoie 9 (l'espace est aussi compté)
y<-1250.12
nchar(y) # renvoie 7 (applicable également pour les variables numériques)
```

1.2.6. Autres attributs d'une variable (objet)

Les attributs d'un objet sont des éléments d'information additionnels liés à cet objet. Les attributs les plus courants d'un objet sont:

- names : les étiquettes des éléments qui forment l'objet
- dim : dimensions d'un objet (matrices et tableaux)
- dimnames: étiquettes des dimensions des matrices et tableaux

- class : indique la classe d'un objet(généralisation de mode)

Pour afficher tous les attributs d'un objet, on utilise la fonction attributes()

```
x<-3
attributes(x)
```

Pour chaque attribut, il existe une fonction du même nom servant à extraire l'attribut correspondant d'un objet. La fonction attributes() permet d'extraire ou de modifier la liste des attributs d'un objet. On peut aussi travailler sur un seul attribut à la fois avec la fonction attr(). On peut ajouter d'autres attributs à la liste des attributs préxistants d'un objet. Par exemple, on pourrait vouloir attacher au résultat d'un calcul la méthode de calcul utilisée. Exemple :

```
x <- 3 # définit l'objet x
attr(x, "methode") <- "Valeur test"  # Associe un attribut à x
nommé méthode dont la valeur est "valeur test"
attributes(x) # Affiche les attributs de x
```

Extraire un attribut qui n'existe pas retourne NULL. Exemple :

```
dim(x) # renvoie NULL
```

À l'inverse, donner à un attribut la valeur NULL efface cet attribut. Exemple :

```
attr(x, "methode") <- NULL
attributes(x) # renvoie NULL
```

1.2.7 Lister les objets sur la mémoire: la fonction ls()

Les objets crées sont automatiquement stockés sur la mémoire. Pour les afficher, on utilise la fonction ls()

Exemples: soient les variables suivantes

```
monobjet<-2
montre<-4
commencer<-5
neutre_m<-10
testV<-50
ls() # affiche tous les objets sur la mémoire
ls(pattern="m") # affiche tous les objets (variables) dont le
nom contient m
ls(pattern="^m") # affiche tous les objets (variables) dont le
nom commence par m
```

1.2.8. Afficher les détails sur les objets: la fonction ls.str()

Exemples:

```
ls.str() # Affiche les détails sur tous les objets
```

```
ls.str(pattern="m") #  Affiche les détails sur tous les objets dont le nom contient m
```

1.2.9 Supprimer les objets: la fonction rm()

Exemples

```
rm(x) # Supprime la variable x
rm( x, y, z) # Les supprime les variables (objets) x, y et z
rm(list=ls(pattern= "m")) # Supprime tous les objets dont le nom contient m.
rm(list=ls()) # supprimer, sans exception, tous les objets crées par l'utilisateur
```

1.3. Les fonctions génératrices de séquences de valeurs: les fonctions seq() et rep()

La fonction seq() permet de générer des séquences de valeurs dans un intervalle préalablement défini.

La fonction rep() permet de répéter (un nombre de fois spécifié par l'utilisateur) une valeur ou une séquence de valeurs (le plus souvent généré par la fonction seq()).

Voir exemples ci-après

1.3.1. La fonction seq()

Exemples d'application:

- Générer une séquence de nombres entiers allant de 5 à 12 avec un pas de 1

```
seq(from=5, to=12, by=1) # renvoie 5  6  7  8  9 10 11 12
```

- Générer une séquence de nombres entiers allant de 5 à 12 avec un pas de 3

```
seq(from=5, to=12, by=3) # renvoie 5  8 11
```

- Générer une séquence de nombres réels entre 5,2 et 12,3 avec un pas 2,

```
seq(from=5.2, to=12.3, by=2) # renvoie 5.2  7.2  9.2 11.2
```

NB: il est possible d'ignorer les mots clés « from », « to » et « by » en indiquant simplement les valeurs. Dans ce cas, il faut respecter l'ordre de spécification des valeurs (pour éviter les confusions).

```
seq(5, 12, 3) # Equivalent à seq(from=5, to=12, by=3)
```

Il existe plusieurs autres variantes de la fonction seq(), il s'agit des fonctions seq.int(), seq_len(), seq_along(). Par exemple :

La fonction seq_len() permet de générer la suite des nombres de 1 à la valeur de l'argument. Ex:

```
x<- seq_len(10) # renvoie 1 2 3 4 5 6 7 8 9 10
```
La fonction seq.int permet de générer des séquences de nombre entiers

```
x<- seq.int(3,12, 2) # renvoie 3  5  7  9 11
```
Autres exemples :

```
seq(from = 1, to = 10) # équivalent à 1:10
seq_len(10) # plus rapide que 'seq'
seq(-10, 10, length = 50) # incrément automatique
seq(-2, by = 0.5, along = x) # même longueur que 'x'
seq_along(x) # plus rapide que 'seq'
```
La fonction sequence ()

La fonction sequence() est un cas particulier de la fonction seq(). Par exemple sequence(a) correspond à seq(1, a,1). Ex:

```
sequence(5)
seq(1,5)
```
NB: En écrivant sequence(a:b), R execute seq(1,r,1) pour tout nombre compris entre a et b. Exemple:

```
sequence(3:5)
```
Une telle spécification peut s'avérer utile dans certains contextes où par exemple on va utiliser des séquences incrémentales.

On peut aussi faire une autre combinaison telle que:

```
sequence( c(4, 5,6)) # Celle-ci exécute séquence(1,a,1) pour tous les nombres fournis.
```
1.3.2. La fonction rep()

Exemples d'application:

```
rep(2, 10) # répète 10 fois le nombre 2 et renvoie le résultat sous forme de séquence de valeurs.
rep(seq(5, 12, 2),3) #  répète 3 fois une séquence de valeurs générée par la fonction seq().
rep(1:3, 5) # génère une séquence de valeurs comprise entre 1 et 3 et répète 5 fois cette séquence
rep(1:2, 2, each = 3) # répète deux fois la séquence dans laquelle les éléments de la séquence 1:2 sont répétés trois fois chacun. Renvoie: 1 1 1 2 2 2 1 1 1 2 2 2 .
```

1.3.3. Création des séquences de valeurs catégorielle (factor): la fonction gl()

La fonction gl() permet de créer des séquences de valeurs facteurs (modalités d'une variable qualitatives). Elle requiert deux paramètres : n, pour indiquer le nombre de niveaux souhaité, et k pour indiquer le nombre de réplications voulu. Exemple :

```
gl(2, 4)   # renvoie  [1] 1 1 1 1 2 2 2 2 ; Repète 4 fois, chaque valeur entre 1 et 2 ## Levels: 1 2
gl(2, 4, length = 10)  ## Repète 4 fois, chaque valeur entre 1 et 2. Mais la répétition s'arrête lorsque le nombre d'éléments atteint 10. renvoie [1] 1 1 1 1 2 2 2 2 1 1 ;## Levels: 1 2
```

Il est possible de définir les étiquettes pour chacun des niveaux, en renseignant le paramètre labels ou encore de préciser si les niveaux des facteurs doivent être ordonnés, avec le paramètre logique ordered. Le paramètre length permet quant à lui de définir la longueur souhaitée du résultat.

Exemples:

```
gl(2, 4, labels = c("Oui", "Non")) ; # Attribue des values labels aux valeurs créees. [1] Oui Oui Oui Oui Non Non Non Non; ## Levels: Oui Non
```

NB: On peut aussi utiliser les fonction seq() et rep() pour génerer les valeurs, ensuite la fonction as.factor() pour convertir ces valeurs en facteurs.

Exemple:

```
x<-as.factor(rep(seq(5, 12, 2),3))
mode(x)
class(x)
```

1.3.4. Combinaison de séquences de valeurs : la fonction expand.grid()

La fonction expand.grid() est une fonction permettant de générer toutes les combinaisons possibles des vecteurs donnés en paramètre. Exemple:

```
mytable<-expand.grid(age = seq(18, 20), sexe = c("Femme", "Homme"), fumeur = c("Oui","Non")) # génère trois variables age, sexe, fumeur.
print(mytable)
```

1.4. Etude des principaux objets de programmation sous R

R étant un langage de programmation orienté « objet », la maitrise de ce langage passe par une bonne compréhension des principaux objets qui constituent son socle.

Pour rappel, les principaux objets de R sont:

- Les Vectors
- Les Matrices
- Les Arrays
- Les Lists
- Les Data frames
- Les Functions
- Les classes et les packages

1.5. Etude des objets de type vector (Vecteur)

Jusqu'à présent, nous avons basé nos discussions sur des variables à valeur unique. Mais la plupart du temps, les variables sont constituées de séquences de valeurs. L'une des formes de ces variables les plus couramment rencontrées sont les vecteurs. On dira que les vecteurs sont des généralisations des variables à valeur unique.

1.5.1. Définir un objet vector

Sous R, le vecteur est l'unité de base de programmation. Contrairement à certains autres langages, il n'y a pas de notion de scalaire en R ; ou du moins un scalaire est simplement un vecteur de longueur 1.

La manière la plus simple pour définir une variable « vecteur » est d'utiliser l'opérateur de concaténation c(),

Exemples:

```
x<-c(2 , 5,  8.5 , 12.3 , 14)  # Définit un vecteur constitué de valeurs numériques
y<-c("London" , "Paris" , "New-York" , "Paris" ) # Définit un vecteur constitué de valeurs caractères
z<-c(5, "London" , 8.5 , "Paris" , "New-York" , 12.3 , "Paris" ) # valeurs numériques et caractères.
```

NB: Une chaine de « caractères » n'est pas nécessairement une séquence de « lettres alphabétiques ». Une valeur formée de chiffres peut aussi être une chaine de caractères lorsqu'il est spécifié entre quotes doubles "" ou simples''.

Exemples:

```
x<-c('2','5', '8', '12', '14')
```

Ici x est bien un vecteur de chaines de caractères car les valeurs ne sont pas reconnues comme numériques à cause des quotes.

Par conséquent, on ne peut pas faire les opérations mathématiques simples. Pour convertir ces valeurs en valeurs numériques, il faut utiliser la fonction as.numeric()

```
x<-as.numeric(x)
print(x)
```

En plus de l'opérateur de concaténation c(), il existent d'autres fonctions prédéfinies permettant de générer des variables en vecteurs. Ci-dessous quelques exemples :

- **la fonction as.vector()** : convertit une séquence de valeurs en vecteur. Ex :

```
x<- seq(1 :20) # définit une séquence de valeur x
y<-as.vector(x) # convertit x en vecteur
```

- **la fonction numeric()** (crée un vecteur de mode numeric). Ex:

```
x<-numeric(length =5) # x a 5 éléments tous  par défaut à 0 (on peut par la suite modifier ces valeurs en utilisant les indices)
```

- **la fonction logical()** (fonction de création de vecteur de mode logical). Ex:

```
x<-logical(length =5) # x a 5 éléments égaux par défaut à False
```

- **la fonction character()** (vecteur de mode character) :

```
x<-character(length = 5) # x a 5 éléments égaux par défaut à espace vide " "
```

- **les fonction LETTERS ou letters** (vecteur constituer de lettre alphabétiques).

```
x<-LETTERS[1:7] # renvoie "A" "B" "C" "D" "E" "F" "G"
x<-letters[1:7] # renvoie des minuscules
```

Quelques remarques sur les vecteurs

Pour la définition d'un vecteur il est possible de donner une étiquette à chacun des éléments d'un vecteur.

```
x <- c(a = 1, b = 2, c = 5) # Attribue des noms aux éléments lors de la création de x
```

On peut aussi définir les noms des éléments après la création de x en utilisation l'attribut names(). Ex:

```
x <- c(1, 2, 5)
names(x) <- c("a", "b", "c")
```

Ces étiquettes ainsi définies font alors partie des attributs du vecteur.

Pour **tester si un objet est un vecteur**, on utilise la fonction is.vector(). Ex:

```
x<-c(2,5, 8, 12, 14)
is.vector(x) # renvoie TRUE
```

Et pour **compter le nombre d'éléments d'un vecteur** on utilise la fonction length(). Ex:

```
x<-c(2,5, 8, 12, 14)
length(x) # renvoie 5
```

1.5.2. Accéder aux éléments d'un vector: le slicing (ou l'indiçage)

1.5.2.1. Définition

La première étape de la manipulation d'un objet vector est le slicing. Le slicing (indiçage) consiste à accéder aux éléments d'un vecteur en se basant sur leur indice (ordre). Le slicing d'un vecteur se fait avec les crochets [ ].

Exemples:

```
x<-c( 2,5, 8, 12, 14, 6, 1, 21) # définit un vecteur
x[1] # renvoie le premier élément de x: 2
x[3] # renvoie le troisième élément de x: 8
```

Pour compter le nombre total d'éléments d'un vecteur, on utilise la fonction length()

```
length(x) # renvoie 8
```

1.5.2.2. Les règles générales du slicing d'un vecteur

On distingue trois manières pour l'indiçage d'un vecteur:

- l'indiçage positif,
- l'indiçage négatif
- l'indiçage par le nom

L'indiçage positif sélectionne les éléments dont les indices sont indiqués entre crochets avec des valeurs positives.

Exemples:

```
x[1] #Renvoie le premier élément
x[3] #Renvoie le troisième élément
x[c(1,3, 5)] #Renvoie le premier, le troisième et le cinquième
élément.
x[2:6] #Renvoie tous les éléments entre 2 et 6.
```

L'indiçage négatif par contre sélectionne tous les éléments à l'exception de ceux dont les indices sont spécifiés entre crochets avec des valeurs négatives.

Exemples:

```
x[-3]  #Renvoie tous les éléments de x sauf le troisième
x[c(-1,-3, -5)]   #Renvoie tous les éléments de x sauf le premier, le troisième et le cinquième
```

Quant à l'indiçage par le nom, il consiste à sélectionner les éléments dont les noms ont été spécifiés à l'intérieur des crochets.

Exemples:

```
x<-c( a=2, b=5, c=8, d=12, e=14, f=6, g=1, h=21) # Définition un vecteur dont les valeurs sont nommées
x['a'] # renvoie la valeur correspondant à l'élément a. ici:2
x['e'] # renvoie la valeur correspondant à l'élément e. ici:14
x[c('a', 'e', 'g')] # renvoie les valeurs correspondant aux éléments a, e et g.
```

1.5.2.3. L'indiçage conditionnel

L'indiçage conditionnel permet de sélectionner les éléments d'un vecteur remplissant une certaine condition mathématique ou logique.

Exemples:

```
x<-c( 2,5, 8, 12, 14, 6, 1, 21) # définition d'un vecteur
x[x<5] # Renvoie tous les éléments de x inférieur à 5.
x[x%%2!=0] # Renvoie tous les éléments impairs de x ( i.e éléments dont le reste de la division par 2 est différent de 0)
```

Utilisation de la fonction which() dans le slicing conditionnel

```
x[which.max(x)] # Renvoie le maximum de x:
x[which(x < 5)] # Renvoie tous les éléments de x inférieur à 5 équivalent à x[x<5]
```

1.5.3. Les opérations courantes sur les vecteurs

1.5.3.1. Les opérations arithmétiques et algébriques

D'une manière générale, lorsqu'on réalise une opération arithmétique ou algébrique sur un vecteur cette opération est effectuée élément par élément.

Voici ci-dessous quelques opérations arithmétiques et algébriques courantes sur les vecteurs:

Addition de deux vecteurs

Soient deux vecteurs x et y définis comme suit

```
x<-c(1, 2, 3)
y<-c(4, 5, 6)
z<-x+y # Fais la somme des éléments de même indice
print(z) # renvoie 5 7 9
```

Addition d'un scalaire à un vecteur

```
x<-c(4, 5, 6) # définit un vecteur x
y<-x+5 # Ajoute 5 à chacun des éléments de x
print(y) # renvoie 9 10 11
```

Produit de deux vecteurs

```
Soient deux vecteurs x et y définie comme suit
x<-c(1, 2, 3)
y<-c(4, 5, 6)
z<-x*y # Fais le produit des éléments de même indice
print(z) # renvoie 4 10 18
```

Division de deux vecteurs

```
Soient deux vecteurs x et y définis comme suit
x<-c(1, 2, 3)
y<-c(4, 5, 6)
z<-x/y # Fais le rapport des éléments de même indice
print(z) # renvoie 0.25 0.40 0.50
```

Produit d'un scalaire à un vecteur

Soient le vecteur x défini comme suit

```
x<-c(4, 5, 6)
y<-x*5 # multiplie par 5 à chacun des éléments de x
print(y) # renvoie 20 25 30
```

1.5.3.2. Les opérations de comparaisons (logiques) sur les éléments d'un vecteur

Tout comme pour les opérations arithmétiques, les opérations de comparaisons logiques sont effectuées élément par élément lorsque l'ensemble du vecteur est spécifié. Ex:

```
x <- seq_len(5)
x < 2 # test si chaque élément de x est Inférieur à 2. Renvoie
TRUE FALSE FALSE FALSE FALSE
x <= 2 # [1] TRUE TRUE FALSE FALSE FALSE
x > 2 # Supérieur à 2 ## [1] FALSE FALSE TRUE TRUE TRUE
x == 2 # Égal à  [1] FALSE TRUE FALSE FALSE FALSE
```

```
x != 2 # Différent de  [1] TRUE FALSE TRUE TRUE TRUE
```

Attention, à l'utilisation de la fonction d'égalité ==. En effet, deux objets qui nous semblent identiques peuvent ne pas l'être rigoueusement avec les fonctions R, à cause des approximations effectuées lors des calculs. Il convient alors dans certains cas d'utiliser la fonction all.equal() plutôt que l'opérateur logique == ou la fonction identical() . Ex:

```
0.9 == (1 - 0.1) ## [1] TRUE
0.9 == (1.1 - 0.2) ## [1] FALSE
identical(0.9, 1 - 0.1) ## [1] TRUE
identical(0.9, 1.1-0.2) ## [1] FALSE
all.equal(0.9, 1-0.1) ## [1] TRUE
all.equal(0.9, 1.1-0.2) ## [1]
```

En fait, la fonction all.equal() donne une égalité approximative, à l'aide d'un seuil de tolérance all.equal(0.9, 1.1-0.2, tolerance = 1e-16).

1.5.4. Ajouter un nouveau élément à un vecteur avec l'opérateur c()

La manière la plus simple pour ajouter un nouveau élément à un vecteur est d'utiliser l'opérateur c() sur l'ancien vecteur et le nouveau et stocker le tout sous un nom égal à celui de l'ancien vecteur.

Exemple 1 : soit le vecteur x définit comme suit :

```
x<- c('a','b','c','d','e','f','g')
```
Ajoutons l'élément "h" à ce vecteur . On a :

```
x<- c(x,'h')
```
Exemple 2

```
x<-c( 2,5, 8, 12, 14, 6, 1, 21) # définition d'un vecteur
x<-c( x, 24 ) #  ajoute 24 au vecteur x.
```

1.5.5. Modifier la valeur d'un élément dans un vecteur

Pour modifier la valeur d'un élément dans un vecteur on se base sur son indice. Exemples :

```
z=c(0, 0, 0, 2, 0)  # définit un vecteur z
z[3] <- 4 # assigne  4 à l'élément d'indice 3
z[c(1,5)] <- 1 # assigne 1 aux éléments d'indice 1 et 5
z[which.max(z)] <- 0 # remplace le maximum de z par 0
z[z==0] <- 8 # remplace les éléments de valeur 0 par 8
```

1.5.6. Exclure les valeurs manquantes d'un vecteur

Soit le vecteur x défini comme suit:

```
x<-c(1,  2,  3 , 4, NA,  6,  7,  8 , 9, NA, 11, 12, 13, 14,
NA, 16, 17, 18, 19, NA)
x<- x[!is.na(x)]
print(x) # renvoie 1  2  3  4  6  7  8  9 11 12 13 14 16 17 18
19
```

1.5.7. Les fonctions courantes de manipulation des vecteurs

Il existe sous R de nombreuses fonctions prédéfinies permettant de réaliser des opérations de traitement sur les vecteurs. Nous présentons ici quelques-unes.

1.5.7.1. Trier les éléments d'un vecteur: la fonction sort()

Exemples:

```
sort(c(4, -1, 2, 6)) # renvoie -1  2  4  6 (tri croissant)
sort(c(4, -1, 2, 6),decreasing = TRUE) # renvoie 6  4  2 -1
(tri décroissant)
```

1.5.7.2. Renverser les éléments d'un vecteur: la fonction rev()

La fonction rev() permet de renverser la disposition initiale des éléments dans un vecteur en plaçant le dernier élément comme le premier, et ainsi de suite. Ex:

```
rev(1:10) # renvoie 10 9 8 7 6 5 4 3 2 1
sort(c(4, -1, 2, 6),decreasing = TRUE) # renvoie 6  4  2 -1
(tri décroissant)
```

1.5.7.3. Supprimer les valeurs dupliquées dans un vecteur: la fonction unique ()

Exemple: Soit le vecteur x

```
x<-c(3,  7,  5, 3,0, 2,  6,7,7  8,  0,  4,  5,9, 10,2)
unique(x) # renvoie  3  7  5  2  6  8  0  4  9 10
```

1.5.7.4. Vérifier si une valeur existe dans un vecteur: l'opérateur %in%

Exemple: Soit le vecteur x définie comme suit:

```
x<-c(4, -1, 2, -3, 6)
5 %in% x # Teste si 5 existe dans x. Renvoie ici FALSE
! 5 %in% x # Teste si 5 n'existe pas dans x (not in). Renvoie
ici TRUE (attention l'opérateur de négation ! not )
```

1.5.7.5. Utilisation de la fonction outer()

La fonction outer() exécute l'opérateur indiqué entre chaque pair d'éléments de deux vecteurs (somme, produit, etc...) pour former une matrice.

Exemples:

```
outer(c(1, 2, 5), c(2, 3, 6), '+') # Calcule la somme entre chaque pair d'élément.
outer(c(1, 2, 5), c(2, 3, 6), '-') # Calcule la différence entre chaque pair d'élément.
outer(c(1, 2, 5), c(2, 3, 6), '*') # Calcule la somme entre chaque pair d'élément.
```

Lorsque la fonction outer() est appliquée sans symbole d'opérateur, elle effectue la multiplication.

On peut aussi appliquer avec outer() toutes les autres opérations (modulos, etc...)

1.5.8. Les opérations ensemblistes sur les vecteurs

Voici-ci dessous les détails sur les opérations ensemblistes sous R.

Appartenance : a appartient à A :

```
is.element(a,A) ou bien a %in%  Renvoie: True ou  False
```

Inclusion : A inclu dans B :

```
 all(A %in% B) Renvoie: True ou  False
```

Contenance : A contient B:

```
all(B %in% A)
```

Intersection : A inter B:

```
intersect(A,B)
```

union : A union B :

```
union(A,B)
```

Complémentaire : A complement B :

```
setdiff(A,B)
```

Difference symétrique : (A union B) complément (A inter B) :

```
setdiff(union(A,B),intersect(A,B))
```

1.5.9. Renvoyer les premières valeurs et les dernières valeurs d'un vecteur: fonctions head() et tail()

La fonction head() renvoie les premiers éléments d'un objet alors que la fonction tail() renvoie les derniers élements. Ex: soit le vecteur x défini comme suit:

```
x<-rep(seq(3,15,3),4)
head(x, n=3) # renvoie les 3 premiers éléments 3 6 9
```

```
head(x, n=3L) # équivalent au précédent renvoie les 3 premiers éléments 3 6 9
head(x, 3) # équivalent au précédent renvoie les 3 premiers éléments 3 6 9
print(tail(x, n=3)) # renvoie les 3 derniers éléments 9 12 15
print(tail(x, n=3L)) # équivalent au précédent renvoie les 3 derniers éléments 9 12 15
print(tail(x, 3)) # équivalent au précédent renvoie les 3 derniers éléments 9 12 15
```

1.5.10. Transformer un vecteur en variable factor: la fonction factor()

La fonction factor() permet de traiter les valeurs d'un vecteur comme des facteurs (modalités). Les variables factor représentent des variables qualitatives (numériques ou caractères) dont les valeurs doivent être traitées comme des modalités. Ex:

Soit le vecteur x défini par:

```
x <- c("France", "France", "Chine", "Espagne", "Chine")
pays <- factor(x)
class(pays) # renvoie factor
levels (pays) # renvoie "Chine"   "Espagne" "France"
```

L'emploi des variables factor est très fréquent dans les modélisations par régressions où on transforme les variables catégorielles en variables binaires pour les inclure dans les régressions. Pour ces tâches, on peut directement utiliser les variables factors (nous y reviendrons).

Pour définir la modalité de référence dans une variable factor, on utilise la fonction relevel() comme suit:

```
pays <- relevel(pays, ref = "Espagne") # Considère l'Espagne comme modalité de référence
```

Si on veut traiter la variable factor comme une variable à modalités ordonnées (variable ordinale), on utilise la fonction ordered(). Ex:

```
x<-c("<1500", ">2000", ">2000", "1500-2000", ">2000", "<1500")
# défini un vecteur constitué des tranches de de revenus
```

On traite cette variable comme une variable factor ordonnée

```
revenus <- ordered(x,levels = c("<1500", "1500-2000", ">2000"))
```

1.5.11. Tabulation d'un vecteur : tableau de fréquence et tableau de contingence: la fonction table()

Tableau de fréquence:

La fonction table() renvoie le tableau de fréquence des éléments d'un vecteur. Ex:

```
table(x) # effectue le tableau de fréquence de x
```

Pour présenter les fréquences relatives (pourcentages), on va procéder comme suit:

```
table(x)/length(x) # fournit proportions
100*table(x)/length(x) # les pourcentages
y<-100*table(x)/length(x) # stocke le tableau sous un nom
```

Tableau de contingence

On peut aussi appliquer la fonction table sur deux vecteurs pour produire un tableau de contingence. Ex:

```
table(x,y) # effectue le tableau de contingence entre x et y
```

Nous reviendrons sur l'utilisation plus poussée de la fonction table dans le chapitre consacré aux analyses statistiques.

1.5.12. Les fonctions statistiques courantes applicables aux vecteurs

La fonction sum(): calcule la somme des éléments d'un vecteur.

Ex:

```
x<-c(14, 17, 7, 9, 3, 4, 25, 21, 24,11)
sum(x) # renvoie 135
```

La fonction cumsum(): calcule la somme cumulée des éléments .

Ex:

```
x<-c(14, 17, 7, 9, 3, 4, 25, 21, 24,11)
cumsum(x) # renvoie   14  31  38  47  50  54  79 100 124 135
```

La fonction prod() : calcule le produit de tous les éléments d'un vecteur.

Ex:

```
x<-c(14, 17, 7, 9, 3, 4, 25, 21, 24,11)
prod(x) # renvoie 24938020800
```

Le tableau ci-dessous présente les fonctions statistiques courantes applicables à un vecteur.

Fonction statistique	Description
cumprod()	Produit cumulé des éléments
mean()	Moyenne des éléments
median()	Médiane des éléments
diff()	Différence entre les éléments successifs du vecteur
var()	variance des éléments du vecteur
sd()	écart-type des éléments du vecteur
min()	minimum des éléments du vecteur
max()	maximum des éléments du vecteur
cummin()	minimum cumulatif des éléments
cummax	maximum cumulatif des éléments
pmin()	minimum parallèle (élément par élément) sur deux ou plusieurs vecteurs
pmax()	maximum parallèle (élément par élément) sur deux ou plusieurs vecteurs
range()	Etendu des éléments du vecteur
quantile()	quantiles empiriques du vecteur(0% 25% 50% 75% 100%)
summary()	Statistiques descriptives

1.5.13. Les arrondis arrondis des valeurs sous R

Il existe différentes manières pour arrondir un nombre décimal sous R. Voici ci-dessous quelques fonctions utiles

La fonction round() : arrondi le nombre avec un nombre de décimal spécifié (par défaut c'est 0). C'est la méthode traditionnelle pour arrondir les nombres. Ex : soit le vecteur x défini comme suit:

```
x<- c(-3.6800000, -0.6666667, 3.1415927, 0.3333333,2.5200000)
round(x) # l'arrondis est effectué à la valeurs supérieur si le décimal est supérieur à 0.5. Renvoie -4 -1  3  0 3
```

```
round(x, 2) # arrondi à 2 chiffres après la virgules
```

La fonction floor() :renvoie la partie entière c'est à dire le plus grand entier inférieur ou égal à l'argument. Il correspond à la fonction round() avec 0 chiffre après la virgule à la seule différence qu'avec la fonction floor() le plus grand entier est pris q=quand la partie décimale est supérieur ou égale à 0.6. Exemple:

```
x<- c(-3.6800000, -0.6666667, 3.1415927, 0.3333333,2.5200000)
floor(x) #  Renvoie -4 -1  3  0  2
```

La fonction ceiling() : effectue un arrondissement un peu particulier qui consiste à prendre la plus petite valeur entière pour les nombres négatifs, la plus grande valeur entière pour les nombres postifs. Pour les nombres négatifs avec des valeurs comprises entre 0 et -1 (exclus), la fonction renvoie 0. Ex:

```
x<- c(-3.6800000, -0.6666667, 3.1415927, 0.3333333,2.5200000)
ceiling(x) #  Renvoie -3  0  4  1  3
La fonction trunc() : renvoie la partie entière d'un nombre
décimal. Ex:
x<- c(-3.6800000, -0.6666667, 3.1415927, 0.3333333,2.5200000)
trunc(x) #  Renvoie -3  0  3  0  2
```

1.6. Etude des objets de type matrix (matrice)

1.6.1. Définir une matrice

Conceptuellement, un objet matrice est jonction de plusieurs vecteurs. Sous R, une matrice se définit avec la fonction matrix().

Exemple:

```
x<-matrix(1:6, nrow = 2, ncol = 3)# Crée une matrice 2x3
rempli des valeurs allant  de 1 à 6 (remplissage par colonne
par défaut)
print(x)
y<-matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE) # ici le
remplissage se fait d'abord par lignes
print(y)
```

On peut aussi ignorer les mots clés ncol et nrow. Dans ce cas, on aura:

```
x<-matrix(1:6, 2, 3)
```

NB: Dans les matrices (tout comme pour les vecteurs), les éléments doivent tous être de même mode.

1.6.2. Déterminer les dimensions d'une matrice

La fonction dim() donne un vecteur contenant les dimensions de la matrice donnée en paramètre. Ex:

```
x<-matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
dim(x)# Renvoie 2 3
```

On peut stocker ces valeurs dans un vecteur

```
y<-as.vector(dim(x))
```

NB : Pour connaitre les noms de lignes et des colonnes on utilise respectivement les fonctions rownames() et colnames()

1.6.3. Indiçage d'une matrice

Le slicing d'une matrice consiste à accéder aux éléments en indiquant leurs indices de ligne et de colonne.

Tout comme pour les vecteurs, l'indiçage des matrices se fait avec les crochets [ ] à la différence qu'il faut indiquer l'indice des deux dimensions (ligne et colonne).

Ex:

```
x <- matrix(c(40, 80, 45, 21, 55, 32), nrow = 2, ncol = 3) # définit une matrice
x[1,3] # renvoie 55 (élément de la première ligne et de la troisième colonne)
x[2,] # renvoie tous les éléments de la 2 ième ligne
x[,2] # renvoie tous les éléments de la 2 ième colonne
x[,] # renvoie tous les éléments de la matrice équivalent à x ou x[]
```

On peut également effectuer un indiçage négatif en utilisant le symbole "-" comme pour les vecteurs.

Exemples:

```
x[-1,] # Renvoie x sans la ligne 1
x[,-3] # Renvoie x sans la colonne 3
x[, -c(1,3)] # Renvoie x sans les colonnes 1 et 3
x[, c(-1,-3)] # Renvoie x sans les colonnes 1 et 3 (équivalent à la formulation précédente)
x[ -c(1,3), ] # Renvoie x sans les lignes 1 et 3
```

Attention: avec les deux derniers exemples, R retourne un vecteur et non une matrice.

En effet quand, il ne reste qu'une seule ligne ou colonne, R renvoie un vecteur (ligne ou colonne). Mais, pour

obtenir une matrice à une seule colonne, il faut ajouter l'argument drop= FALSE

```
x[, -c(1,3), drop = FALSE]
```

1.6.4. Renvoyer les indices des éléments d'une matrice: la fonction wich()

Tout comme pour les vecteurs, on peut utiliser la fonction which() pour récupérer les indices des éléments répondant à une condition. Ex:

```
x <- matrix(1:9, ncol = 3, nrow = 3)
which(x > 5, arr.ind = TRUE) # Renvoie les indices de tous les éléments supérieurs à 5
```

Pour le cas d'une matrice, les indices renvoyés sont présentés sous formes de couple (ligne ou colonne).

1.6.5. Fusionner deux matrices de même dimension : les fonctions rbind() et cbin()

Les fonctions rbind() et cbind() permettent de fusionner des matrices. La première effectue une fusion par ligne (fusion verticale ou append) et la seconde une fusion par colonne (fusion horizontale ou merge). Exemples:

```
x <- matrix(c(40, 80, 45, 21, 55, 32), nrow = 2, ncol = 3)
y <- matrix
k <- matrix(1:4, nrow = 2)
```

1.6.5.1. Fusion verticale (append)

```
z1=rbind(x, y)
print(z1)
```

1.6.5.2. Fusion horizontale (merge)

```
z2=cbind(x, k)
print(z2)
```

NB : Pour le cas des bases de données, il est recommandé d'utiliser la fonction merge() pour la fusion horizontale.

1.6.6. Les opérations algébriques courantes sur une matrice

Soient les matrices A, B et C et un scalaire a définis comme suit:

```
A <- matrix(c(1, 3, 2, 2, 2, 1, 3, 1, 3), ncol = 3)
B <- matrix(c(4, 6, 4, 5, 5, 6, 6, 4, 5), ncol = 3)
C <- matrix(c(0, 3, 1), ncol = 1)
a<-2 # un scalaire
```

On peut effectuer les opérations algébriques suivantes:

```
A+a # ajout du scalaire à tous les éléments de A
A-a # soustraction de a de tous les éléments de A
a*A # Multiplication de a avec tous les éléments de A
A/a # Division de tous les éléments de A par a
A+B # Addition de A et B
A-B # Différence entre A et B
A%*%C # multiplication entre la matrice A et la matrice C ( ne pas confondre avec * multiplication terme à terme entre matrices)
```

Les opérations algébriques courantes sur les matrices

- Transposée d'une matrice: la fonction t()
- Déterminant d'une matrice: la fonction det()
- Inverse d'une matrice : la fonction solve()
- Renvoyer les éléments diagonaux d'une matrice: la fonction diag()
- La conjuguée d'une matrice: Les fonction Conj()
- Produit croisé entre matrices: Les fonction crossprod()

1.6.7. Autres opérations courantes sur les matrices

```
nrow(x) # renvoie le nombre de lignes de la matrice x
ncol(x) # renvoie le nombre de colonnes de la matrice x
rowSums(x) # renvoie  la somme par ligne
colSums(x) # renvoie la somme par colonne
rowMeans(x) # renvoie la moyenne par ligne
colMeans(x) # renvoie la moyenne par colonne
```

1.7. Etude des objets de type array(tableau)

1.7.1. Définition d'un array

Un array est une généralisation d'une matrice à plus de deux dimensions. La fonction de base pour créer des tableaux est array(). Exemple :

```
x<-array(1:24, dim = c(3, 4, 2))
```

Ici, on crée un array à deux dimensions dont chaque dimension est constituée par une matrice de dimension 3x4. Le remplissage des matrices est effectué à partir de la séquence de valeurs comprises entre 1 et 24 (remplissage par colonne ensuite par ligne)

Une matrice à plus de deux dimensions doit être d'abord vue comme une matrice dont les éléments sont eux-mêmes des vecteurs ou des matrices. Par exemple, un

array de dimension 3 × 4 × 5 est un tableau constitué de cinq matrices 3 × 4 (remplies par colonne) les unes à la suite des autres. Autrement dit, le tableau est un prisme rectangulaire de hauteur 3, largeur 4 et de profondeur de 5. Si l'on ajoute une quatrième dimension, cela revient à aligner des prismes les uns à la suite des autres, et ainsi de suite.

Notons aussi que, comme pour les vecteurs et les matrices, les éléments d'un array doivent toutes être du même type.

1.7.2. Dimension d'un array

La fonction dim() donne un vecteur contenant les dimensions de l'array donné en paramètre.

```
x<-array(1:24, dim = c(3, 4, 2))
dim(x)# Renvoie 3 4 2
```

1.7.3. Indiçage d'un array

Pour renvoyer un élément singulier d'un array, on indique l'index de ligne et l'index de colonne ainsi l'index de la dimension auquel il appartient. Ex:

```
x<-array(1:24, dim = c(3, 4, 2))
print(x)
x[1,1,1] # élément de la première ligne, première colonne, de la première dimension. Renvoie 1
x[1,1,2] # renvoie 13
x[2,3, 1] # renvoie 8
x[2,3, 2] # renvoie 20
x[,,1] # Renvoie tous les éléments de la dimension 1
x[,,2] # Renvoie tous les éléments de la dimension 2
x[,2,1] # Renvoie tous les éléments de la 2ième colonne de la dimension 1
x[2,,1] # Renvoie tous les éléments de la 2ième ligne de la dimension 1
x[2,3,] # Renvoie les éléments se trouvant sur la 2ième ligne et la troisième colonne dans chaque dimension.
```

Accéder à un élément singulier d'un array

On peut renvoyer un élément d'un array en spécifiant son rang dans la séquence sous-jacente. Ex:

```
x<-array(1:24, dim = c(3, 4, 2))
x[3] # renvoie le troisième élément de la séquence. ici 3
```

1.8. Etude des objets list (liste)

1.8.1. Définir une liste

L'objet de type list est l'objet le plus général du langage R. Les listes sont des collections d'objets (vecteur, matrice ,array) de type divers (numérique ou en caractères).

La fonction de base pour créer des listes est list(). Exemple :

```
x <- list(mesVal1 = c(1, 5, 2), mesVal2 = "mon texte", mesVal3
= 5)
y <- list(c(1, 5, 2), "mon texte",5)
```

Avec x, on définit une liste à trois éléments nommés respectivement mesVal1, mesVal2, mesval3.

Notons aussi qu'on peut définir une liste sans nommer les éléments. Mais le nommage est fortement recommandé. ydéfini ci-dessus est une liste définie sans nommer ses éléments

1.8.2. Attributs d'une liste

```
mode(x) # renvoie list
length(x) # renvoie 3
attributes(x) # Renvoie les attributs
```

1.8.3. Définition d'une liste à partir d'autres objets.

Il est possible de définir un objet liste à partir d'autres objets préalablement définis comme les vecteurs, les matrices, etc. Ex:

```
myVec1 <- c("Washington", "Dallas", "Chicago", "Los Angeles",
"St-Louis", "Détroit", "Montréal", "Boston")
myVec2 <- c(55, 56, 57, 58, 56, 57, 56, 57)
myVec3 <- c(36, 32, 30, 30, 25, 25, 22, 24)
myVec4 <- c(16, 19, 21, 22, 19, 21, 27, 31)
myVec5 <- c( 3, 5, 6, 6, 12, 11, 7, 2)
myVec6 <- c(75, 69, 66, 66, 62, 61, 51, 50)
myList<-list(myVec1,myVec2,myVec3,myVec4,myVec5,myVec6    )    #
associe les vecteur pour former une liste.
print(myList)
```

On pouvait aussi définir la liste tout en nommant les éléments.

```
myList<-list(nameVec1=myVec1,nameVec2=myVec2,nameVec3=myVec3,
nameVec4=myVec4,nameVec5=myVec5,nameVec6=myVec6 ) #
print(myList)
```

1.8.4. Indiçage des éléments d'une liste

Tout comme pour les autres objets, l'indiçage d'une liste se fait avec l'opérateur []. Mais il y a des spécificités propres aux listes que nous allons détailler ci-dessous à savoir l'utilisation de [[]] et de [][]

Soit la liste x suivante:

```
x <- list(c(12, 15, 25), "mon texte", 5)
```

Dans cette liste, on distingue les « éléments » et les « éléments des éléments ».

Voici ci-dessous quelques formes d'indiçage pour chaque niveau d'élements :

```
x[1] # renvoie l'élément d'indice 1:  12, 15, 25 (formulation non recommandée pour une liste)
x[[1]]  # renvoie l'élément d'indice 1:  12, 15, 25 (formulation recommandée)
x[[2]] # renvoie l'élément d'indice 2: "mon texte"
x[[1]][2] # renvoie l'élément d'indice 2 du premier éléments: 15
x[[c(1,2)]]  # équivalent à la formulation précédente. l'élément d'indice 2 du premier éléments Renvoie 15
```

NB: Avec les listes, on ne peut extraire qu'un seul élément à la fois avec les crochets doubles [[ ]]. Ce qui signifie qu'on ne peut effectuer d'indiçage multiple à moins d'utiliser une boucle « for …in »

Indiçage d'une liste avec les noms des éléments

Lorsque les éléments sont nommés, on peut directement utiliser ces noms pour accéder aux éléments en utilisant le symbole $ ou avec les guillemets dans les crochets. Ex:

```
x <- list(mesVal1 = c(1, 5, 2), mesVal2 = "mon texte", mesVal3 = 5)
 x$mesVal1 # renvoie les valeurs 1 5 2 (un vecteur numérique)
x[['mesVal1']]  # renvoie les valeurs 1 5 2
x[[1]]  # renvoie les valeurs 1 5 2
x$mesVal2 # renvoie "mon texte"
x[['mesVal2']]  # renvoie "mon texte"
x[[2]]  # renvoie "mon texte"
x$mesVal1[2] # renvoie 5
x[['mesVal1']][2]  # renvoie 5
x[[1]][2]  # renvoie 5
x[[c(1,2)]]  # renvoie 5
```

NB: L'utilisation du symbole dollar ($) est valable uniquement dans le cas des listes, arrays et data frames.

Il est possible d'examiner les attributs de chaque élément qui forme la liste. Ex:

```
mode(x$mesVal3)
mode(x$mesVal2)
mode(x$mesVal1)
```

On peut aussi tester la classe d'un élément d'une liste avec les fonctions is.list(), is.vector(), is.matrix(), is.array(). Exemples :

```
x <- list(mesVal1 = c(1, 5, 2), mesVal2 = "mon texte", mesVal3
= 5)
is.list(x) # renvoie TRUE
is.vector(x[1]) # TRUE
is.vector(x[[1]]) # TRUE
is.vector(x[[2]]) # TRUE
is.matrix(x[[2]]) # FALSE
is.matrix(x[[2]]) # FALSE
is.array(x[[1]]) # FALSE
is.array(x[[2]]) # FALSE
```

1.8.5 Convertir une liste en vecteur: fonction unlist()

D'une manière générale la fonction list() permet de convertir un vecteur en une liste. Mais il est possible de faire le chemin inverse en convertissant une liste en un vecteur. Pour cela, on utilise la fonction unlist(). Ex:

```
x <- list(mesVal1 = c(1, 5, 2), mesVal2 = "mon texte", mesVal3
= 5)
z=unlist(x) # Tous les éléments sont associés en un ensemble
de valeurs.
```

1.9. Etude des objets de type data frame (table de données)

Un data frame est un tableau de données à deux dimensions où les lignes représentent les observations et les colonnes les variables. Le data frame est une généralisation de la matrice dans la mesure où tous les éléments formant l'objet ne sont pas nécessairement de même type. En effet, dans une matrice, tous les objets doivent être de même type. Alors que dans un data frame, les éléments peuvent être de types différents (variables numériques et variables en caractères).

1.9.1. Création d'un data frame

On crée un data frame avec la fonction data.frame(). Exemple: Soient les vecteurs définis comme suit:

```
myVec1 <- c("Washington", "Dallas", "Chicago", "Los Angeles",
"St-Louis", "Détroit", "Montréal", "Boston")
myVec2 <- c(55, 56, 57, 58, 56, 57, 56, 57)
myVec3 <- c(36, 32, 30, 30, 25, 25, 22, 24)
```

```
myVec4 <- c(16, 19, 21, 22, 19, 21, 27, 31)
myVec5 <- c( 3, 5, 6, 6, 12, 11, 7, 2)
myVec6 <- c(75, 69, 66, 66, 62, 61, 51, 50)
```

Rassemblons ces 6 vecteurs en un data frame tout en les attribuant des noms. On a:

```
myData<-
data.frame(myVar1=myVec1,myVar2=myVec2,myVar3=myVec3,myVar4=
myVec4,myVar5=myVec5,myVar6=myVec6 ) #
print(myData)
```

NB: Si les noms ne sont pas indiqués, les colonnes prennent les noms des vecteurs qui servent à les définir :

```
myData<-data.frame(myVec1,myVec2,myVec3,myVec4,myVec5,myVec6 )
print(myData)
```

On peut aussi définir les noms des colonnes d'un data frame après sa création lorsque les colonnes n'ont pas de noms. Ex:

```
myData<-data.frame(c("Washington",  "Dallas",  "Chicago",  "Los
Angeles", "St-Louis", "Détroit", "Montréal", "Boston"),
                   c(55, 56, 57, 58, 56, 57, 56, 57),
                   c(36, 32, 30, 30, 25, 25, 22, 24),
                   c(16, 19, 21, 22, 19, 21, 27, 31),
                   c( 3, 5, 6, 6, 12, 11, 7, 2),
                   c(75, 69, 66, 66, 62, 61, 51, 50)
                   )
print(myData)
myColNames<-
c("myVar1","myVar2","myVar3","myVar4","myVar5","myVar6")      #
préparation des noms pour les colonnes
names(myData)<-myColNames # Attribution des noms aux colonnes
print(myData)
```

1.9.2. Dimensions d'un data frame

Comme pour les matrices, on peut obtenir les dimensions d'un data frame avec la fonction dim(), le nombre de lignes avec nrow() et le nombre de colonnes avec ncol().

```
dim(myData) # renvoie 8 6
nrow(myData) # renvoie 8
ncol(myData) # renvoie  6
```

1.9.3. Conversion d'un objet en data frame

Pour convertir un autre type d'objet en data frame on utilise la fonction as.data.frame().

Exemple1: Conversion d'un vecteur en data.frame:

```
myVector <- c(55, 56, 57, 58, 56, 57, 56, 57) # définit un vecteur
myData<-as.data.frame(myVector)
print(myData)
```

Exemple2: Conversion d'une matrice en data frame

```
myMatrix <- matrix(c(40, 80, 45, 21, 55, 32), nrow = 2, ncol = 3)
myData<-as.data.frame(myMatrix)
print(myData)
```

Exemple3: Conversion d'un array en data frame

```
myArray<-array(1:24, dim = c(3, 4, 2))
myData<-as.data.frame(myArray)
print(myArray)
print(myData)
```

1.9.4. Indiçage d'un data frame

Comme un data frame se présente comme une matrice à 2 dimensions (ligne x colonne), l'indiçage se fait alors comme pour une matrice. Par ailleurs, puisque les colonnes sont nommées, on peut effectuer l'indiçage par nom sur ces colonnes pour renvoyer ces colonnes. On peut également utilier l'indiçage par le symbole $. C'est d'ailleurs la méthode la plus couramment utilisée pour les data frames :

Exemple: soit le data frame myData défini précédemment avec myVec1 jusqu'à myVec6 :

On peut effectuer les indiçage suivants

```
myData[1] # renvoie la première colonne nommée myVar1
myData$myVar1 # renvoie la première colonne en se basant sur le nom myVar1.
myData['myVar1'] # équivalent aux précédent méthode.
myData[-1] # Renvoie toutes les colonnes sauf la première
myData[c(1,3,5)] # Renvoie les colonnes d'indices 1, 3 et 5
myData[c('myVar1','myVar2','myVar3')] # sélectionne les trois colonnes
```

1.9.5. Fusion de data frame : ajout de lignes ou de colonnes à un data frame

Tout comme pour les matrices, les fonctions rbind() et cbind() peuvent être utilisées pour ajouter des lignes ou des colonnes à un data frame (à condition qu'il y ait une correspondance parfaite entre les deux data frames notamment dans la disposition des observations pour le cas cbind(). Exemple : soit deux vecteurs définis comme suit:

```
myNewVar<-c(21, 30 , 14 ,10 , 60, 42, 50, 12)
myNewObs<-list(c('San Francisco', 63, 25 , 35 ,7, 60)
```

Ajoutons ces deux vecteurs au data frame myData définie plus haut. On a:

```
myData2=cbind(myData, as.data.frame(myNewVar)) # ajoute une nouvelle colonne
print(myData2)
myData3=rbind(myData, as.data.frame(myNewObs)) # ajoute une nouvelle observation
print(myData3)
```

NB : Pour ajouter des observations ou des variables à une data frame, il est conseillé d'utiliser la fonction merge() qui offrent plus d'options que les fonction rbind() et cbind() (Nous reviendrons sr l'utlisation de cette fonction dans le chapitre consacré au data management).

1.9.6. Rendre disponible un data frame à un environnement de travail

Pour pouvoir effectuer les analyses sur le data frame sans être obligé à chaque fois de spécifier le nom de la table de donnée, on peut indiquer une fois pour toute le nom de la table en utilisant la fonction attach(). Cette fonction rend disponible les colonnes d'un data frame dans l'espace de travail.On utilise la fonction attach(). Pour les détacher, on utiliser detach().

Exemple :

```
attach(myData)
detach(myData)
```

1.9.7. Création de data frame à partir de données externes (importations)

1.9.7.1. Lecture des données à partir d'un fichier texte (avec séparateur)

Lecture des fichiers de données texte ASCII.

Pour lire un fichier texte sous R, on utilise généralement les fonctions de la famille read.table() ou la fonction scan().

La fonction read.table() possède plusieurs variantes dont les principales sont read.csv, read.csv2, read.delim, read.delim2 (voir description plus bas).

On utilise read.cvs ou read.cvs2 pour les données séparées avec virgules ou point-virgule. Et on utilise read.delim ou read.delim lorsqu'on a des données avec un séparateur quelconque qui doit être spécifié par l'utilisateur.

Exemples : importation de fichier texte avec séparateur tabulation \t.

```
mydata3<-
read.table("mytabdelimdata.txt",header=T,dec=".",sep="\t")
head(mydata3, n=5L)
```

Il est conseillé d'exploiter les nombreuses options disponibles pour la fonction read.table().

Voici ci-dessous la description de la fonction read.table()

- file: Le nom du fichier. Il peut être précédé du chemin relatif ou absolu. Attention. Pour les utilisateurs de Windows,le caractère "\" doit être remplacé par "/" ou bien "\\". À A noter qu'il est possible de saisir une adresse web (URL) en guise de nom de fichier (nous y reviendrons).
- header: Valeur logique (header=FALSE par défaut) indiquant si la première ligne contient les noms de variables.
- sep: Le séparateur de champ dans le fichier (chaîne vide par défaut, ce qui est au final traduit par une espace comme séparation). Par exmple, utiliser sep=";" si les champs sont séparés par un point-virgule, ou encore sep="\t" s'ils sont séparés par une tabulation.
- dec: Le caractère employé pour les décimales (par défaut, dec=".").
- row.names: un vecteur contenant le nom des lignes (de type caractère), ou bien le numéro ou le nom d'une variable du fichier. En omettant ce paramètre, les lignes sont numérotées de 1 à n.
- na.strings: Une chaîne de caractère (ou un vecteur de chaînes de caractères) indiquant la valeur des données manquantes (par défaut, na.strings="NA"). Ces données manquantes seront converties en NA.
- colClasses: Un vecteur de caractères indiquant les modes des colonnes.

Importation du fichier texte avec séparateur csv

Importons le fichier mycsvdata (fichier avec séparateur csv) et le stocker dans un objet de donnée nommé mydata1.

```
mydata1<-read.table("mycsvdata.csv",header=T,dec=".", sep=";")
head(mydata1, n=5L) # Affiche les 5 premières lignes
```

Importation d'un fichier de données ne contenant pas les noms des variables

```
mydata<-
read.table("mycsvdata.csv",header=FALSE,dec=".",sep=";")
```

On définit dans un second temps les noms des variables.

```
colnames(mydata) <- c("annee","mois", "ihpc", "datapubli")
```

A la place de la fonction read.table(), on peut aussi utiliser read.csv ou read.delim pour lire ce fichier. En effet, il existe des variantes de read.table() qui s'appuient sur cette fonction pour proposer à l'utilisateur des fonctions directement capables de lire leur fichier de données, sans avoir à changer les paramètres sep et decim. Ci-dessous la description de ces fonctions :

- read.csv() :séparateur "," ; symbole décimal "."
- read.csv2() :séparateur ";" ; symbole décimal ","
- read.delim() :séparateur "\t" ; symbole décimal "."
- read.delim2() :séparateur "\t" ; symbole décimal ","

Par ailleurs, lors de l'importation, il est possible d'employer l'instruction file.choose() pour ouvrir une boîte de dialogue au lieu d'écrire le chemin complet vers le fichier de données.

```
mydata <- read.table(file.choose())
```

Données importées à partir du presse-papier (copier-coller)

Pour importer les données à partir du presse-papier, on sélectionne par CTRL+C ou par le copier classique avec clique droit de la souris dans le tableur la plage de données. Une fois que les données ont été copiées (utiliser COMMAND+C sous Mac), il suffit ensuite de taper l'instruction suivante dans la console de R pour que les données y soient transférées depuis le presse-papiers.

```
x <- read.table(file("clipboard"),sep="nt",header=TRUE,dec=
",")
```

1.9.7.2. Lecture de données à partir du tableau Microsofst Excel.

Il existe plusieurs packages fournissant des fonctions pour importer les données Excel dans R. Les fonctions les plus connues sont read.xls(), read_excel() et read.xlsx() ou read.xls2() qui sont des fonctions appartenant respectivement aux packages gdata, readxl et xlsx. Ici, nous présentons uniquement l'utilisation des fonctions read.xlsx() et read.xlsx2() du package xlsx qui sont des fonctions plus récentes.

Utilisation du packahe xlsx

Le package doit d'abord être installé avec :

```
install.packages("xlsx")
```

Exemples:

```
library(xlsx)
```

```
mydata<- read.xlsx("dataWorkbook.xlsx",sheetName="patientdata"
,startRow=1,endRow=355,colIndex=c(1:22), header=TRUE,
as.data.frame= TRUE)
```
La différence entre les fonctions read.xlsx() et read.xlsx2() est que:

read.xlsx() préserve le type des données en essayant de deviner le type de classe (nombre, caractère, logique,) de chaque variable correspondant à chaque colonne dans le classeur. Notons toutefois que la fonction read.xlsx() est un peu plus lent pour lire des gros fichiers de données (c'est à dire une feuille Excel avec plus de 100 000 cellules). Quant à la fonction read.xlsx2(), elle est beaucoup plus rapide pour la lecture des gros fichiers Excel.

1.9.8. Exporter les data frames vers des formats externes

A l'inverse d'une importation de data frame à partir de fichiers externes, on peut aussi exporter un data frame vers des fichiers externes (fichiers textes avec séparateur, fichiers excels). La fonction de base pour exporter les données est la write.table(). Il existe aussi divers packages offrant des fonctions d'exportation. Nous allons présenter le cas de la fonction write.xlsx() et write.xlsx2() du package xlsx.

1.9.8.1. Exportation du data frame vers des formats texte avec séparateur : la fonction write.table()

Pour enregistrer des données depuis un data.frame, un vecteur ou une matrice, la fonction write.table() peut être utilisée. Par exemple, si le data.frame se nomme mydata, la syntaxe de base se présente comme suit :

```
write.table(donnees, file = "nom_fichier.txt", sep = ";")
#Fichier txt séparé avec ";"
```
Exemple:

```
write.table(mydata , file = "mydata_exp.csv", append = FALSE,
sep = ";", col.names = T, dec = "." ,qmethod = "double" ) #
Exportation vers un fichier csv
```
Attention: Dans la fonction write.table(), le nom de l'objet à exporter est écrit sans guillemets.

Il existe plusieurs variantes de write.table notamment write.csv()

1.9.8.2. Exportation vers un fichier Microsoft Excel : fonctions write.xlsx() et write.xlsx2() du package xlsx

Les fonctions write.xlsx et write.xlsx2 peuvent être utilisées pour exporter des données de R vers Excel. Noter que la fonction write.xlsx2 est plus performante que write.xlsx pour les grosses tables de données (avec plus de 100 000 cases).

La syntaxe simplifiée de l'utilisation de ces fonctions est :

```
write.xlsx(x, file, sheetName="Sheet1", col.names=TRUE,
row.names=TRUE, append=FALSE)
write.xlsx2(x, file, sheetName="Sheet1",col.names=TRUE,
row.names=TRUE, append=FALSE)
```

- x : le data.frame à écrire dans le fichier Excel
- file : chemin du fichier résultat
- sheetName : texte indiquant le nom de la feuille Excel
- col.names, row.names : une valeur logique indiquant si le nom des colonnes/lignes doit être écrit dans le fichier
- append : une valeur logique indiquant si les données doivent être ajoutées dans un classeur déjà existant (dans une nouvelle feuille).

Exemples :

```
library(xlsx)
write.xlsx(mydata, file="myExcelData.xlsx",sheetName="data")
```

Pour ajouter plusieurs data frames dans le même classeur Excel, on utilise l'argument append = TRUE.

```
write.xlsx(mydata1,
file="myworkbook.xlsx",sheetName="mydata1", append=FALSE) #
Ecrire la première table dans un nouveau classeur
write.xlsx(mydata2, file="myworkbook.xlsx",
sheetName="mydata2",append=TRUE) # Ajouter une deuxième table
```

1.10. Les structures de contrôle dans un programme R

1.10.1. Les principales structures de contrôle

On distingue deux principales structures de contrôle dans le langage R: les instructions conditionnelles et les instructions en boucle.

- Les « **instructions conditionnelles** » sont des instructions exécutées lorsqu'une condition préalablement définie par l'utilisateur se vérifie. Sous R, les instructions conditionnelles sont formulées dans des clauses « if… else », des clauses « ifelse » ou des clauses « switch ».

- Les « **instructions en boucle** » sont des instructions qui continuent de s'exécuter tant qu'une condition d'arrêt n'est pas vérifiée, On distingue deux principales catégories d'instruction en boucles: les boucles de type « while » et les boucles « for…in ». Les instructions « while » s'exécutent tant qu'une condition reste vérifiée et les instructions « for…in » s'exécutent pour chaque élément appartenant à la séquence de valeurs définie par la clause « in ».

Les deux principales structures de contrôles sont très souvent accompagnées par des clauses complémentaires notamment les clauses: repeat, break et next.

1.10.2. La clause « if...else »

La structure générale d'une clause « if... else » sous R se présente comme suit:

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

Les instructions sont spécifiées entre accolades surtout s'il en y a plusieurs pour une même branche de la clause « if....else ».

Exemple:

```
x<-123
if (x %% 5==0 ) {
    print("Oui x est divisible par 5")
} else{
  print("Non, x n'est pas divisible par 5")
}
```

1.10.3. La clause « ifelse »

La clause « ifelse » est le condensé de la clause « if... else » permettant de spécifier les deux branches de la clause dans un même bloc d'instructions. La structure générale d'une clause ifelse est la suivante:

```
ifelse condition, {instruction si condition vraie}, { instruction si condition fausse}
```

Les trois membres de la clause sont séparés par des virgules et les instructions se rattachant à chaque branche de la clause peuvent être spécifiées entre accolades si elles sont nombreuses. Exemple:

```
experience<-c(2, 12, 6,15,8,1,3,4,11,5) # Définit un vecteur représentant l'expérience de 10 employés.
ifelse(experience > 3, "Sénior", "Junior" ) # Définit une ifelse qui renvoie « Sénior« lorsque experience >3 et « junior« sinon.
```

NB: Les valeurs renvoyées peuvent être assignées à un nom. Ex:

```
statut<-ifelse(experience > 3, "Sénior", "Junior" )
statut
```

Noter aussi que la clause « ifelse » peut se présenter sous forme de clause « ifelse » imbriqué. C'est à dire qu'au niveau de la branche où l'instruction est fausse, on peut définir une nouvelle clause ifelse. Exemple:

```
ifelse( mydata$age > 75, c("Elder"), ifelse(mydata$age > 45 &
mydata$age <= 75 , c("Middle Aged"), c("younger")) )
```
Attention à l'emplacement des ifelse et le nombre de parenthèses. A chaque ifelse correspond une nouvelle parenthèse.

Le code "autre" doit toujours être spécifié dans le dernier ifelse

1.10.4. La clause switch()

La clause switch permet de définir les instructions conditionnelles pour chaque valeur d'une variable test. Sa structure générale est la suivante:

```
switch(valeur_test,
       cas_1 = {
         instruction_cas_1
       },
       cas_2 = {
         instruction_cas_2
       },
       ...
       cas_n = {
         instruction_cas_n
       }
)
```

Avec valeur_test un nombre ou une chaîne de caractères. Si valeur_test vaut cas_1, alors uniquement instruction_cas_1 sera évaluée, si valeur_test vaut cas_2, alors ce sera instruction_cas_2 qui le sera, et ainsi de suite.

Exemple :

```
mystats <- function(x, type) { # type prendra 2 valeurs
possible: mean ou mediane
  switch(type,
         mean = mean(x),
         median = median(x))
}
x<- rcauchy(10)
mystats(x, "mean") # renvoie la moyenne de x
mystats(x, "median") # renvoie la médianne de x
```
Dans la syntaxe générale, on peut rajouter une valeur par défaut en utilisant la syntaxe suivante :

```
switch(valeur_test,
       cas_1 = {
         instruction_cas_1
```

```
        },
        cas_2 = {
          instruction_cas_2
        },
        {
          instruction_defaut
        }
)
```

1.10.5. Les boucles « for… in »

La structure générale d'une boucle for se présente comme suit:

for élément in sequence {instructions}

La boucle « for… in » exécute toutes les instructions définies entre accolades pour chaque élément de la séquence. La séquence est généralement un vecteur.

Exemple1:

```
for (i in c(2,5,8, 24, 36,6)) {
    print(i) # Affiche la valeur du compteur.
}
```

Exemple 2:

```
for (i in 1:50) {
  if (i %% 5==0 )  {
    print(i)  #  Affiche  la  valeur  du  compteur  lorsqu'il  est
divisible par 5.
  }
}
```

Exemple 3: On va élaborer une boucle for dans laquelle on crée un vecteur x rempli de valeurs non divisibles par 5 comprises entre 1 et 100. Et lorsqu'une valeur est multiple de 5, celle-ci sera simplement affichée à l'écran. La démarche sera la suivante:

```
x <- c() # vecteur vide
for (v in 1:100) { # définition des valeurs de la séquence
  if (v %% 5!=0 ) # si v n'est pas un multiple de 5
  {
    x <- c(x,v) # Créer un nouveau vecteur en combinant x et v
  }
  else  { # sinon
    print(v) # afficher la valeur à l'écran
  }
```

```
}
print(x) # Affichage de x (x ne contient plus les NA)
```

1.10.6. Les boucles « while »

La structure générale d'une boucle « while » est la suivante:

```
while (condition) {instructions}
```

Cette boucle exécute les instructions définies entre accolades tant que la condition "condition" est vérifiée.

Exemple: On va élaborer une boucle « while » définie à partir d'un compteur allant de 1 à 100. Lorsque la valeur du compteur n'est pas multiple de 5, elle est stockée dans un vecteur x. Lorsque la valeur du compteur est multiple de 5, celle-ci sera affichée à l'écran. La démarche sera la suivante:

```
x <- c() # vecteur vide
v<-1 # Initialisation du compteur à 1
while (v <= 100) { # définition de la condition d'arrêt
  if (v %% 5!=0 ) # si v n'est pas un multiple de 5
  {
    x <- c(x,v) # ajoute v à x (Créer un nouveau vecteur en
combinant x et v)
  }
  else { # sinon
  print(v) # afficher la valeur à l'écran
  }
  v<-v+1 # Incrémentation du compteur
}
print(x) # Affichage de x
```

1.10.7. L'instruction repeat

L'instruction repeat est une forme particulière de la boucle « while » qui répète l'exécution des mêmes instructions tant qu'une condition secondaire est vérifiée.

Exemple 1 : Incrémenter la valeur d'une variable jusqu'à une valeur donnée

```
i <- 1
repeat {
  i <- i + 1
  if (i == 3) break
}
```

Exemple 2: On va élaborer une boucle repeat définie à partir d'un compteur commençant à 1 et qu'on incrémente. Lorsque la valeur du compteur n'est pas

multiple de 5, elle est stockée dans un vecteur x. Lorsque la valeur du compteur est multiple de 5, celle-ci sera affichée à l'écran. La boucle s'arrête lorsque la valeur du compteur dépasse 100. La démarche sera la suivante:

```
x <- c() # vecteur vide
v<-1 # Initialisation du compteur à 1
repeat   {
  if (v %% 5!=0 ) # si v n'est pas un multiple de 5
  {
    x <- c(x,v) # ajoute v à x (Créer un nouveau vecteur en
combinant x et v)
  }
  else # sinon
  {
    print(v) # afficher la valeur à l'écran
  }
  v<-v+1 # Incrémentation du compteur
  if (v>100) #critère d'arrêt
    break
}
print(x) # Affichage de x
```

La différence entre la boucle « while » et l'instruction « repeat » est l'emplacement de la condition d'arrêt. Pour la boucle « while », elle est placée à l'entrée de la boucle et pour la boucle « repeat », elle est placée à l'intérieur de la boucle à la suite de l'instruction d'incrémentation et spécifiée avec une condition if.

1.10.8. L'instruction break

La clause break arrête l'exécution d'une boucle lorsqu'une condition secondaire définie à l'intérieur d'une boucle se vérifie. Cette clause est utilisée généralement dans le cas des boucles « while » et « repeat ».

Exemple: On va élaborer une boucle « repeat » définie à partir d'un compteur commençant à 1 et qu'on incrémente.

Lorsque la valeur du compteur n'est pas multiple de 5, elle est stockée dans un vecteur x. Lorsque la valeur du compteur est multiple de 5, elle-ci sera affichée à l'écran. La boucle s'arrête lorsque la valeur du compteur dépasse 100

La démarche sera la suivante:

```
x <- c() # vecteur vide
v<-1 # Initialisation du compteur à 1
repeat  {
  if (v %% 5!=0 ) # si v n'est pas un multiple de 5
  {
```

```
    x <- c(x,v) # ajoute v à x (Créer un nouveau vecteur en
combinant x et v)
  }
  else { # sinon
      print(v) # afficher la valeur à l'écran
  }
  v<-v+1 # Incrémentation du compteur
  if (v>100) #critère d'arrêt
    break
}
print(x) # Affichage de x
```

La boucle continue d'exécuter les instructions jusqu'à ce que la condition d'arrêt soit vérifiée. break est généralement placée en fin de boucle après la ligne d'incrémentation.

1.10.9. L'instruction next

L'instruction « next » suspend l'exécution d'une boucle lorsqu'une condition secondaire définie à l'intérieur d'une boucle se vérifie et reprend l'exécution pour l'itération suivante. Cette clause est utilisée généralement dans le cas des boucles « for…in », « while » et l'instruction « repeat ».

Exemple: On va élaborer une boucle « for…in » sur les éléments d'une séquence allant de 1 à 100. Lorsque la valeur du compteur n'est pas multiple de 5, elle est stockée dans un vecteur x. Lorsque la valeur du compteur est multiple de 5 on saute cet élément et on passe à l'élément suivant. La démarche sera la suivante:

```
x <- c() # vecteur vide
for (v in 1:100) { # définition des valeurs de la séquence
  if (v %% 5==0 ) # Condition de suspension de la boucle pour
l'élément
  {
    next # passe à l'élément suivant
  }
  x <- c(x,v) # ajoute v à x (Créer un nouveau vecteur en
combinant x et v)
}
print(x) # Affichage de x
```

La boucle vérifie d'abord si la condition est vérifiée avant d'exécuter les instructions après next.

L'instruction next est généralement placée en début de boucle.

Exemple 2:

```
for(i in 1:10) {
```

```
  if(i == 5) next
  resul[i] <- i
}
resul ## [1] 1 2 3 4 NA 6 7 8 9 10 # Le 5e élément de resul
est resté non-disponible
```

1.11. Etudes des objets fonctions

1.11.1. Définir une fonction

Une fonction est un ensemble d'instructions séquentielles conçu pour exécuter une tâche bien précise. C'est généralement un mini-programme élaboré à partir d'un ensemble d'objets sur lesquels on exécute un ensemble d'instructions notamment des structures de contrôle.

La syntaxe générale de définition d'une fonction sous R est la suivante :

```
myFunction <- function(arguments){
    instructions
}
```

- myFunction est le nom qu'on attribue à la fonction (les règles pour les noms de fonctions sont les mêmes que pour tout autre objet) ;
- arguments est la liste des arguments (paramètres et variables servant à définir les instructions). Les arguments sont séparés par des virgules ;
- instruction constitue le corps de la fonction. Elles se présentent sous forme d'expression ou de groupes d'expressions réunies par des accolades.

Voici ci-dessous quelques exemples de fonctions:

Exemples1:

Ecrivons une fonction affichant les n premiers termes de la table de multiplication d'un nombre quelconque x indiqué par l'utilisateur.

```
myMultipTable <- function(x, n) { # Déclaration de la fonction
nommée myMultipTable dont les arguments sont x et n
print(sprintf("Les %i premiers éléments de la table de
multiplication par %i",n, x))
  for (i in 1:n){
    val<-x*i
    print(sprintf("%i  x  %i = %i", x ,i, val))
  }
}
myMultipTable (x=5,n=10) # appel de la fonction avec les
valeurs x=5 et n=10
# On peut aussi appeler la fonction sans les noms des
arguments. Ex:
```

```
myMultipTable(5,10)
```

Exemple 2 : Fonction calculant les n premiers termes de la suite de FIBONACCI:

```
fib <- function(n) # Déclaration de la fonction nommée fib
dont l'argument est n
{
  res <- numeric(n) # Vecteur devant contenir les n valeur (
mais initialisée d'abord à 0)
  res[2] <- 1 # Le remplissage comme à l'indice 2 car res[1]
vaut déjà 0
  for (i in 3:n) # Calcul pour les autres éléments
    res[i] <- res[i - 1] + res[i - 2] # Somme entre l'élément
i et son précédent
  res # renvoie la valeur de res
}
fib(5) # appel de la fonction avec n=5
fib(20) # appel de la fonction avec n=20
```

1.11.2. Définition de fonctions sans arguments (paramètres)

On est parfois amené à créer des fonctions qui ne prennent pas de paramètres. Il suffit alors de laisser la liste de paramètres formels vide.

Exemple:

```
f <- function() {sample(letters, size = 10, replace = TRUE)} #
Tirage aléatoire de 10 lettre parmi les 26 lettres minuscules
de l'alphabet
f() # appel de la fonction f.
```

1.11.3. Portée d'une variable dans une fonction : variables locales et variables globales

Comme la majorité des langages de programmation, R comporte des concepts de variable locale et de variable globale. Une variable locale est une variable dont la valeur est accessible uniquement à l'intérieur de la fonction alors qu'une variable globale est une variable qui peut être accessible partout dans le programme principal et par toutes les autres fonctions qui le constituent.

Au-delà de cette première distinction, les variables locales et globales se distinguent aussi sur plusieurs autres aspects :

- toute variable définie comme locale dans une fonction ne modifie pas une variable portant le même nom dans le programme principale.

- la variable locale prend la valeur qui lui a été assignée l'intérieur de la fonction. La valeur qui lui a été assignée à l'extérieur de la fonction reste intacte.

Ajoutons à la fonction mymultipTable une variable supplémentaire qui stocke les valeurs calculées,

```
mymultipTable <- function(x, n) {
  print(sprintf("Les %i premiers éléments de la table de multiplication par %i",n, x))
  listVal<-c() # Définir une liste vide à laquelle on va joindre les valeurs calculées
  for (i in 1:n){
    val<-x*i
    print(sprintf("%i  x  %i = %i", x ,i, val))
    listVal<-c(listVal, val)   }}
```

Exécutons cette fonction modifiée avec les valeurs x=5 et n=10 et tentons ensuite de calculer la somme des valeurs obtenues. On a:

```
myMultipTable(x=5,n=10)  # appel de la fonction avec les valeurs x=5 et n=10
sommeVal<-sum(listVal)  # Ceci renvoie un message "Error: object 'listVal' not found"
```

On obtient ce message car la variable listVal est déclarée comme local lors de la définition de la fonction myMultipTable.

Pour rendre possible l'opération, il faut déclarer la variable comme global.

Pour ce faire on va définir la variable listVal en utilisant la fonction assign() avec l'option envir = .GlobalEnv

La définition de la fonction se fait alors comme suit:

```
myMultipTable <- function(x, n){
  print(sprintf("Les %i premiers éléments de la table de multiplication par %i",n, x))
  assign('listVal', c(), envir = .GlobalEnv)
  for (i in 1:n){
    val<-x*i
    print(sprintf("%i  x  %i = %i", x ,i, val))
    assign("listVal", c(listVal, val), envir = .GlobalEnv)
     }
}
```

Exécutons la fonction avec les valeurs x=5 et n=10 et tentons maintenant de calculer la somme des valeurs obtenues. On a:

```
myMultipTable(x=5,n=10)
```

```
sommeVal<-sum(listVal)
print(sommeVal) # Renvoie maintenant la somme demandée.
```
Au final, une variable locale globale est une variable dont la valeur est accessible à l'extérieur de la fonction.

NB : Signalons aussi qu'il existe la notion de fonction locale. En effet, fonction définie à l'intérieur d'une autre fonction. Cette fonction sera locale à la fonction dans laquelle elle est définie.

1.11.4. Disposition des paramètres lors de la définition et de l'appel d'une fonction

Lors de l'appel de la fonction, R cherche d'abord s'il y a des paramètres nommés afin de leur associer des valeurs. S'il n'y a pas de noms définis, il se basera alors sur la position donnée aux paramètres. Exemple :

Considérons par exemple le problème suivant. Nous disposons d'une fonction de production Y (L,K,M), qui dépend du nombre de travailleurs L et de la quantité de capital K, et du matériel M, telle que Y (L,K,M) = (L^0.3)(K^0.5)*(M^0.2). Cette fonction pourra s'écrire, en R de la manière suivante :

```
production <- function(l, k, m) { l^(0.3) * k^(0.5) * m^(0.2)}
```
Si on nous donne L = 60 et K = 42 et M = 40, on peut appeler la fonction comme suit :

```
production(60, 42, 40) # renvoie 46.28945
```
Dans l'exemple ci-dessus, l'appel est effectué sans spécifier les noms des paramètres et les égalités avec les valeurs indiquées. Alors 60 est attribué au paramètre situé première position, 42 au deuxième paramètre et 40 au troisième paramètre.

Effectuons un second appel avec:

```
production(k = 42, m = 40, l = 60)  # renvoie 46.28945
```
Dans cet appel, on indique tous les noms de paramètres

```
production(k = 42, 60, 40) # On indique seulement le nom du paramètre k
```
Dans chacun de ces appels, R affecte les valeurs selon leu position après avoir attribué les valeurs aux paramètres dont les noms sont biens spécifiés. Mais ceux-ci doivent être placés avant les autres.

1.11.5. Paramètres obligatoires et paramètres avec les valeurs par défaut

Essayons par exemple de définir la fonction de production, en attribuant une valeur par défaut à k telle que:

```
production2 <- function(l, m, k = 42) l^(0.3) * k^(0.5) *
m^(0.2) # on fixe à 42 la valeur par défaut de k.
```

La valeur par défaut est la valeur que la fonction va utiliser lorsque le paramètre en question est omis lors de l'appel de la fonction. Dans ce cas, le paramètre devient un paramètre optionnel alors que les autres restent des paramètres obligatoires. Par exemple, faisons quelques appels de la fonction définie:

```
production2(l = 42, m = 40) # le paramètre k est omis alors,
la fonction prend la valeur par défaut 42
production2(l = 42, m = 40, k = 2) # le paramètre k prend la
valeur 2
```

Dans l'exemple, le paramètre à qui nous avons donné une valeur est placé en dernier. Ce n'est pas obligatoire, mais c'est plus pratique, si le but recherché est de ne pas avoir à saisir le paramètre effectif lors de l'appel de la fonction. De plus, si l'utilisateur ne nomme pas les paramètres lors de l'appel, des problèmes liés à l'ordre peuvent apparaître. Il faut donc déclarer les paramètres optionnels après les paramètres obligatoires. Pour le voir prenons l'exemple ci-dessous:

```
production3 <- function(l, k = 42, m) l^(0.3) * k^(0.5) *
m^(0.2)
production3(l = 42, m = 40) # Ne renvoie pas d'erreur car les
paramètres obligatoires sont explicitement spécifiés avec leur
nom.
production3(42, 40) # renvoie une erreur
```

Ce dernier appel renvoie une erreur car la fonction attribue 42 à l, 40 à k et il ne reste plus de valeur correspondante pour m (alors qu'en réalité la valeur 40 devait plutôt être attribué à m puisque k a une valeur par défaut). R envoie le message suivant: ## Error in production_3(42, 40): l'argument "m" est manquant, avec aucune valeur par défaut

1.11.6. Accéder aux composantes d'une fonction: les fonctions formals(), body() et environment()

A l'exception des fonctions primitives du package base, toutes les fonctions définies par l'utilisateur sont composées de trois parties : une liste de paramètres, un corps d'instructions, contenant du code exécuté lors de l'appel à la fonction ; et un environnement, qui définit l'endroit où sont stockées les variables. On peut accéder à ces trois composantes (et les modifier) avec les fonctions formals() pour les paramètres, body() pour le corps et environment() pour l'environnement.

Exemple : soit la fonction fib définie précédemment, on peut accéder aux éléments de cette fonction en faisant

```
body(fib) # Renvoie le code entier
```

```
formals(fib) # renvoie la liste des paramètres
environment(fib) # Renvoie l'environnement de d'élaboration du programme.
```

1.11.7. Récupérer les valeurs renvoyées par une fonction: l'instruction return

Par défaut, une fonction retourne toujours le résultat de la dernière instruction du corps de la fonction. Il faut donc éviter que la dernière expression soit une assignation, car la fonction ne retournera alors aucune valeur et on ne pourra utiliser une construction de la forme x <- f() pour affecter le résultat de la fonction à une variable. En revanche, si le résultat à récupérer n'est pas à la dernière ligne de la fonction (situé par exemple à l'intérieur d'un bloc conditionnel), il faut utiliser la fonction return.

Exemple 1a: Renvoyer la valeur sans utilisation de la fonction return

```
f <- function(x) {
  y<-x^2 # Renvoie le carré de x dans y
  y
}
f(2) # appel de la fonction avec le paramètre 2
z1<-f(2) # Assignation de f(2) à z1
```

Exemple 1b: Renvoyer la valeur avec l'utilisation de la fonction return

```
f <- function(x) {
  return(x^2) # Renvoie le carré de x
}
f(2) # appel de la fonction avec le paramètre 2
z2<-f(2) # Assignation de f(2) à z
```

NB : L'utilisation de return à la toute fin d'une fonction est tout à fait inutile et considérée comme du mauvais style en R.

Lorsqu'on doit récupérer plusieurs résultats à partir d'une fonction, il est en général préférable de les stocker d'abord dans une liste à l'intérieur de la fonction et de renvoyée cette liste.

Exemple : utilisation d'une liste pour stocker les résultats

```
mystats <- function(x) {
  list(moyenne = mean(x), ecart_type = sd(x)) # Calculer la moyenne et l'écart-type pour un vecteur
}
x<- runif(10) # génère une séquence de valeurs.
mystats (x)
```

1.11.8. Exécuter une fonction sans afficher les valeurs : utilisation de la fonction invisible()

Il est possible de ne pas afficher le résultat renvoyé suite à l'appel à une fonction. Pour cela, on utilise la fonction invisible().

Exemple:

```
mystats2 <- function(x) {
  invisible(list(moyenne = mean(x), ecart_type = sd(x)))
}
# appel de la fonction
x <- runif(10)
mystats2(x) # n'affiche pas les résultats
str(mystats2(x)) # Affiche les résultats
print(mystats2(x)) # affiche les résultats
mystats2(x) # affiche les résultats
```

Lorsque la dernière instruction est une assignation, nous sommes dans le cas d'un résultat invisible. Exemple :

```
f <- function(x){
  res <- x^2
}
f(2)
```

1.11.9. Exécuter une fonction sur chaque élément d'une séquence de valeurs: la fonction do.call()

La fonction do.call() permet d'exécuter une fonction sur chaque valeur d'une séquence. La fonction prend deux paramètres : le premier est le nom d'une fonction et le second celui d'une liste.

1.11.10. Sélectionner les éléments à afficher parmi les résultats renvoyés par une fonction :

Tout comme les éléments d'un array, une liste, ou une data frame, on peut sélectionner parmi les résultats renvoyés par une fonction stockés dans une liste) ceux que nous souhaitons afficher. Pour cela, on utilise l'opérateur $. Ex:

```
mystats <- function(x) {
  invisible(list(moyenne = mean(x), ecart_type = sd(x)))
}
# appel de la fonction
x <- runif(10)
mystats # tous les résultats
mystats(x)$moyenne # affiche la moyenne
```

```
mystats (x)$ecart_type # affiche ll'écart-type
```

1.11.11. Quelques fonctions avancées de R : les fonctions de la famille « apply »

Les fonctions de la famille « apply » sont des fonctions avancées de R qui permettent de vectoriser les instructions R. En d'autres termes, elles permettent d'«appliquer» une fonction sur les éléments d'un vecteur afin de renvoyer les résultats. Les fonctions apply se présentent souvent comme des alternatives à l'utilisation des boucle « while » ou « for…in ».

Ci-après quelques fonctions de la famille apply et leur principal rôle.

1.11.11.1. La fonction apply()

La fonction apply() sert à exécuter une fonction quelconque sur une matrice ou un array et renvoie les résultats sous forme de vecteur.

Exemple: soit la matrice définie comme suit:

```
x <- matrix(sample(1:100, 20, rep = TRUE), 5, 4)
```

On a les apply suivantes:

```
apply(x, 1, min) # Renvoie le minimum pour chaque ligne de la matrice x (dimension 1)
apply(x, 2, min) # minimum pour chaque colonne de la matrice x (dimension 2)
apply(x, 1, mean) # moyenne pour chaque ligne de la matrice x (dimension 1)
apply(x, 2, mean) # moyenne pour chaque colonne de la matrice x (dimension 2)
apply(x, 1, sum) # somme pour chaque ligne de la matrice x (dimension 1)
apply(x, 2, sum) # somme pour chaque colonne de la matrice x (dimension 2)
```

Les fonctions utilisées ici sont les fonctions statistiques usuelles (cf. section opérations sur les vecteurs).

1.11.11.2. Les autres fonctions apply()

La fonction apply() se décline en plusieurs autres variantes dont les principales sont:

- **lapply():** applique une fonction à tous les éléments d'un vecteur ou d'une liste et retourne le résultat sous forme de liste.
- **sapply ()** : similaire à lapply (s'applique aux éléments d'un vecteur ou d'une liste), sauf que le résultat est retourné sous forme de vecteur, d'une matrice

ou un array au lieu d'une liste dans le cas de lapply. Le résultat est donc simplifié par rapport à celui de lapply, d'où le nom de la fonction.

- **mapply()** : une version de sapply applicable à plusieurs listes ou vecteurs simulatanément.
- **vapply():** similaire à sapply(), mais elle possède un type de valeurs spécifié,
- **tapply():** s'applique à chaque cellule d'un tableau, sur des regroupements définis par les variables catégorielles fournies.

Conseils pratiques: Avant d'utiliser une boucle, vérifier d'abord si les tâches ne sont pas réalisables avec l'une des fonctions de la famille « apply ».

1.12. Tirage aléatoire et générateurs de nombres aléatoires

1.12.1. Fixation du seed : la fonction seed()

Avant de faire un tirage aléatoire ou générer des nombres aléatoires, il est conseillé de fixer le seed. Cela permet de reproduire à l'identique les résultats du tirage lorsque celui-ci est re-exécuté une nouvelle fois. En effet à chaque exécution de la fonction génératrice de nombres aléatoires, de nouvelles valeurs sont renvoyées. Il faut alors fixer le seed afin de pouvoir reproduire les résultats. Pour fixer le seed, on utilise la fonction set.seed comme suit:

```
set.seed(2016) # fixe le seed à 2016
```

Pour annuler le seed, on spécifie la fonction :

```
set.seed(Sys.time()) # FIxe le seed à la valeur par défaut qui est le temps sur le système.
```

1.12.2. Le tirage aléatoire simple dans un échantillon : la fonction sample()

Le tirage aléatoire simple consiste à tirer un échantillon à partir d'une base de tirage (base de sondage ou population). On distingue le tirage aléatoire simple avec remise et le tirage aléatoire simple sans remise. Dans chacun de ces deux méthodes de tirage, il peut s'agir d'un tirage à probabilités égales ou un tirage à probabilités inégales.

Pour effectuer un tirage aléatoire simple, on utilise la fonction sample(). Sa syntaxe générale est la suivante:

```
sample(x, size, replace = FALSE, prob = NULL)
```

- x : représente la base de tirage c'est à dire le vecteur d'éléments à partir duquel le tirage doit être effectué.

- size: représente la taille de l'échantillon à c'est à dire le nombre d'observations à tirer
- replace : permet d'indiquer s'il s'agit d'un tirage avec remise (TRUE) ou sans remise (FALSE)
- prob, elle indique s'il s'agit d'un tirage à probabilités égales (valeur NULL) ou probabilité inégale. Dans ce dernier cas, il faut alors spécifier le nom du vecteur contenant les probabilités des éléments du vecteur x.

Exemples d'applications

```
sample(1:49, 7) # Tire sans remise 7 éléments sur les 49
sample(1:10, 10) # tire 10 éléments sur 10 (sans remise).
```

NB: Dans le deuxième exemple comme, il s'agit d'un tirage sans remise où tous les éléments sont tirés. Ce qui d'ailleurs constitue une bonne astuce pour mélanger l'ordre initial des éléments et mettre un ordre aléatoire. Cela peut s'avérer nécessaire pour faire d'autres types de tirages comme les tirages systématiques.

```
sample(1:10, 10, replace = TRUE) # échantillonnage avec remise.
```

On peut aussi spécifier une distribution de probabilités non uniforme.

```
x <- sample(c(0, 2, 5), 1000, replace = TRUE,prob = c(0.2, 0.5, 0.3))
table(x) # tableau de fréquences
```

1.12.3. Fonctions génératrices de nombres aléatoires suivant une loi donnée

Comme de nombreux logiciels de programmation et de statistiques, R offre plusieurs possibilités pour générer des nombres aléatoires suivant une loi donnée. Par exemple, pour générer un nombre aléatoire suivant une loi uniforme, on utilise la fonction runif(). Dans cette fonction, on spécifie la borne inférieure et la borne supérieure ainsi que le nombre de valeurs à tirer.

Exemple:

```
x<-runif(5,2,10) # Tirer 5 nombres aléatoires suivant une loi uniforme dans l'intervalle de 2 et 10
print(x)
set.seed(2016) # fixe le seed à 2016
x<-runif(5,2,10) # Tirer 5 nombres aléatoire suivant une loi uniforme dans l'intervalle de 2 et 10
set.seed(2017)
set.seed(Sys.time()) #Réinitialise le seed
print(x)
print(y)
```

En fixant le seed, à chaque exécution du programme, x envoie les mêmes valeurs et y envoie les mêmes valeurs. A chaque tirage, il faut fixer son propre seed si nécessaire.

La liste ci-dessous présente les principales fonctions génératrices de nombres aléatoires suivant une certaine loi.

- Binomiale : rbinom(n, size, prob) ;(la loi de bernoulli est un cas particulier de la loi binomiale avec size=1)
- Poisson : rpois(n, lambda) ;
- Géométrique : rgeom(n, prob) ;
- Hyper-géométrique : rhyper(nn, m, n, k) ;
- Binomiale négative : rnbinom(n, size, prob, mu).
- Normale : rnorm(n, mean = 0, sd = 1) ;
- Student : rt(n, df, ncp) ;
- Khi-deux : rchisq(n, df, ncp = 0) ;
- Fisher : rf(n, df1, df2, ncp) ;
- Exponentielle : rexp(n, rate = 1) ;
- Uniforme : runif(n, min = 0, max = 1) ;
- Beta : rbeta(n, shape1, shape2, ncp = 0) ;
- Logistique : rlogis(n, location = 0, scale = 1) ;
- Log-Normale : rlnorm(n, meanlog = 0, sdlog = 1) ;
- Gamma : rgamma(n, shape, rate = 1, scale = 1/rate) ;
- Weibull : rweibull(n, shape, scale = 1) ;
- Loi Wilcoxon:fonction--> wilcox():Arguments:--> m, n
- Loi Cauchy:fonction--> cauchy():Arguments:--> location, scale

Exemples d'applications des fonctions

```
runif(10) # tirage de 10 valeurs suivant une loi uniforme sur (0, 1) par défaut
runif(10, 2, 5) # Tirage sur un autre intervalle
2 + 3 * runif(10) # équivalent à la précédente spécification
rbinom(10, 5, 0.3) # 10 tirage: Binomiale(5, 0,3)
rbinom(10, 1, 0.3) # 10 tirage: Bernoulli(0,3). la loi de bernoulli est un cas particulier de la loi binomiale avec size=1
rnorm(10) # 10 tirage selon une loi Normale(0, 1)
rnorm(10, 2, 5) # 10 tirage: Normale(2, 25)
rpois(10, c(2, 5)) # 10 tirage: P(2), P(5), P(2), ..., P(5)
rgamma(10, 3, 2:11) # 10 tirage:  G(3, 2), G(3, 3), ..., G(3, 11)
rgamma(10, 11:2, 2:11) #10 tirage:  G(11, 2), G(10, 3), ..., G(2, 11)
```

Il faut aussi noter que pour chaque racine loi (r), il existe 3 fonctions associées que sont :

1. rloi : qui génère des observations de cette loi (racine de la loi).

2. dloi : renvoie la fonction de densité de probabilité (loi continue) ou la fonction de masse de probabilité (loi discrète) ;

3. ploi : renvoie la fonction de répartition de loi ;

4. qloi : renvoie la fonction de quantile de loi ;

Par exemple, pour la loi uniforme, on a les fonctions suivantes :

```
runif(n, min, max) # Fonction génératrice de base. Génère des nombres aléatoires suivant la loi uniforme.
dunif(x, min, max) # Renvoie la densité de probabilité des éléments figurant dans le vecteur de quantile x. (min et max représente les bornes théoriques de la distribution)
punif(q, min, max) # Renvoie la fonction de répartition des éléments figurant dans le vecteur de quantile q.
qunif(p, min, max) # Représente la réciproque de punif, c'est à dire renvoie les quantiles pour un vecteur de fonction de répartition donné p
```

Cette règle s'applique également à toutes les lois spécifiées ci-dessus.

Par ailleurs, il faut noter que toutes les fonctions ci-dessus présentées sont vectorielles c'est-à-dire qu'elles acceptent en argument un vecteur de points où la fonction (de densité, de répartition ou de quantile) doit être évaluée et même un vecteur de paramètres.

Les exemples ci-dessous illustrent quelques utilisations des fonctions génératrices de nombre aléatoire.

```
dpois(c(3, 0, 8), lambda = c(1, 4, 10)) # retourne la probabilité que des lois de Poisson de paramètre 1, 4 et 10 prennent les valeurs 3, 0 et 8, respectivement.
```

Le premier argument de toutes les fonctions de simulation est la quantité de nombres aléatoires désirée. Ainsi, dans le cas suivant, on a:

```
rpois(3, lambda = c(1, 4, 10)) # retourne trois nombres aléatoires issus de distributions de Poisson de paramètre 1, 4 et 10, respectivement.
```

1.13. Traitement des variables en chaîne de caractères

1.13.1. Définition une variable en chaine de caractères

Une variable en chaine de caractères se définit comme une variable classique par assignation directe avec l'opérateur <- ou en utilisant la fonction assign(). Toutefois la valeur à assigner doit être indiquée entre quotes (simples ' ' ou double " "). **Exemples :**

```
chaine1<-"Ceci est une valeur en chaîne de caractères" # définit une chaine à valeur unique
print(chaine1)
chaine2 <- c("un","deux","trois") # Un vecteur contenant plusieurs chaines
print(chaine2)
chaine3<-as.character(1:3) # Chaine définie à partir des valeurs numériques
print(chaine3)
```

1.13.2. Générer une séquence lettres alphabétiques à partir des fonctions letters et LETTERS

Les fonctions letters et LETTERS renvoient les vingt-six lettres de l'alphabet en minuscules et en majuscules.

```
x<-letters
print(x)
y<-LETTERS
print(y)
```

1.13.3. Convertir une variable numérique en chaine de caractères: la fonction toString()

```
x<-2
y<-toString(x)
print(y)
```

On pouvait aussi utiliser la fonction as.character() comme suit:

```
z<-as.character(x)
print(z)
```

1.13.4. Convertir une variable en caractères en valeurs numériques: la fonction as.numeric()

Pour convertir en format numérique un chiffre stocké sous forme de caractères, on utilise la fonction as.numéric(). Exemple :

```
x<-"2" # définit une variable caractère à partir des chiffres.
x<-as.numeric(x) # Convertit la variable x en numérique.
print(y)
```

1.13.5. Affichage des chaines de caractères sans les guillemets : la fonction noquote()

Par défaut en faisant print() sur une variable en caractères, R affiche les valeurs entre guillemets. On peut alors utiliser la fonction noquote() supprimer l'affichage des guillemets dans les sorties.

```
chaine1<-"Ceci est une valeur en chaîne de caractères"
print(chaine1)
noquote(chaine1) # fait un print() sans les guillemets
```

1.13.6. Concaténation de chaine de caractères: les fonctions cat(), paste(), paste0() et str_c()

1.13.6.1. La fonction cat()

La fonction cat() permet concatener et afficher un ensembles de chaines de caractères défini par des valeurs libres ou par des variables en chaines de caractères. **Exemple:**

```
cat("Ceci", "est un", "test") #  concatène les trois chaînes
avec le séparateur par défaut (espace). Renvoie Ceci est un
test
cat("Ceci", "est un", "test", sep = "--") # concaténation avec
séparateur par "--"
cat("Ceci", "est un", "test", sep = "\n") # Le séparateur ici
\n qui est l'opérateur de passage à la ligne.. Chaque élément
est donc affiché sur une ligne spécifique.
```

NB : La fonction cat() peut prendre en argument tout objet R (vecteur, matrice, array, etc…). Mais la fonction convertit d'abord ces objets en vecteurs de chaînes de caractères, qui sont ensuite concaténés en un seul vecteur de caractères. Les éléments de ce vecteur sont ensuite joints entre eux, et éventuellement séparés par un caractère différent de l'espace, si le paramètre sep n'est pas explicitement indiqué. Exemple.

```
x<-matrix(1:6) #  définit une matrice
cat(x) # Concatène les éléments de x. Renvoie 1 2 3 4 5 6.
```

1.13.6.2. Les fonctions paste() et paste0()

La fonction paste() permet de concaténer les chaines de caractères en spécifiant un séparateur comme pour la fonction cat(). Quant à la fonction paste0(), elle concatène les caractères sans séparateur.

```
texte1<-"Ceci est une chaine de caractères"
texte2<-"Celle-là aussi est une chaine de caractères"
paste(texte1, texte2, sep="--") # Concatène texte1 et texte2
avec comme séparateur le symbole "--".
paste0(texte1, texte2) # renvoie "Ceci est une chaine de
caractèresCelle-là aussi est une chaine de caractères"
```

1.13.6.3. La fonction str_c() du package stringr

La fonction str_c() du package stringr permet de concaténer les chaines de caractères passées en argument. Cette fonction offre beaucoup plus d'options que les fonctions classiques cat(), paste() et paste0(). En effet, contrairement à la fonction cat() qui se limite à l'affichage, la fonction str_c() permet de stocker le résultat dans un nouvel objet.

Une autre différence de la fonction str_c avec les fonctions de base est qu'il n'y a pas de séparateur par défaut. Celui-ci doit être explicitement indiqué.

Voici ci-dessous quelques exemples d'utilisation de str_c()

```
install.packages("stringr") # installation du package stringr
library(stringr) # chargement de la librairie.
x <- str_c("Hello", "World", "!", sep = " ") # Séparateur avec
espace indiqué. Renvoie "Hello World !"
print(x)

x <- str_c("Hello", "World", "!", sep = "--") # Renvoie
"Hello--World--!"
print(x)
```

Utilisation de l'option collapse

La fonction str_c() dispose aussi de l'options collapse, qui permet de joindre les éléments d'un vecteur dans une même chaîne de caractères. Exemple :

Soit le vecteur x définit comme suit :

```
x<- LETTERS[1:10] # définit une vecteur contenant "A" "B" "C"
"D" "E" "F" "G" "H" "I" "J".
```

```
str_c(x, sep = ", ") # Renvoie le même vecteur (sans modification) car il n'y a qu'un seul argument (le séparateur est alors sans effet). Il faut utiliser collapse.
str_c(x, collapse = ", ") # renvoie : "A, B, C, D, E, F, G, H, I, J"
```

1.13.7. Formatage de valeurs : placer une valeur au milieu d'une chaine de caractères: la fonction sprintf()

La fonction sprintf() permet de concaténer un ensemble de chaines de caractères tout en autorisant l'insertion d'une valeur numérique ou d'une valeur en caractères provenant d'une autre variable. Exemples:

```
x<-2
y<-3
z1<-x*y
z2<-x/y
mytext<- "Le résultat"
sprintf("%s de la multiplication de %i par %i est égal à %i" , mytext, x,y, z1) # renvoie "Le résultat de la multiplication de 2 par 3 est égal à 6"
sprintf("%s de la multiplication de %i par %i est égal à %f" , mytext, x,y, z2) # renvoie "Le résultat de la multiplication de 2 par 3 est égal à 0.666667"
```

Les format %s, %i et %f signifient respectivement : string, integer et float.

Le format %s doit être utilisé pour les valeurs en chaine de caractères.

Le format %i pour les entiers et le format %f pour les nombres réels. Pour fixer le nombre de virgules pour un float, on fait par exemple %.2f qui fixe le nombre de décimal à deux chiffres.

Pour placer le text à l'intérieur d'une chaîne, on place la sprintf() à l'intérieur de la chaines avec les valeurs comme indiqué plus haut. Ex :

```
myVal=sprintf("La valeur est %.2f",val)
```

Il existe d'autres formats (consulter la documentation R)

1.13.8. Convertir une chaine de caractères en minuscule ou majuscule: fonctions tolower(), toupper() et casefold()

Pour convertir une chaine de caractères en minuscule ou majuscule, on utilise les fonction tolower(), toupper ou casefold(). Ci-dessous les exemples d'applications

```
x <- "Bonjour !"
toupper(x) # Conversion en majuscule "BONJOUR !"
tolower(x)# Conversion en minuscule "bonjour !"
```

```
casefold(x) # Conversion en minuscule, valeur par défaut de casefold()
casefold(x, upper = TRUE) # Convertit en majuscule "BONJOUR !"
```

1.13.9. Compter le nombre de caractères dans une chaine

Pour compter le nombre de caractères dans une chaines, on peut utiliser la fonction nchar() du module base de R ou la fonction str_length() du package stringr. Ces fonctions indiquent le nombre de caractères contenus dans une chaîne (espace compris). Exemple :

```
nchar("Bonjour Monsieur")
library(stringr)
str_length("Bonjour Monsieur") # Renvoie 16
```

Les fonctions nchar() et str_length() peuvent aussi prendre des objets comme argument (vecteur, liste, etc). Dans ce cas, c'est le nombre de caractères de chaque élément qui est renvoyé. Exemple:

```
mytexte <- c("Ceci", "est un", "test sur les caractères", NA)
nchar(mytexte) # renvoie 4  6 23 NA
str_length(mytexte) # renvoie 4  6 23 NA
```

On peut noter que la longueur de NA vaut NA avec les deux fonctions.

1.13.10. Indiçage dans une chaine de caractères

Pour indicer ou extraire une sous-chaîne, on peut utiliser la fonction substr() ou substring() du package base ou utiliser la fonction str_sub() du package stringr. Ces fonctions prennent en paramètres une chaîne de caractères, la position du début et celle de la fin de l'élément à extraire.

NB : On peut aussi utiliser la fonction word() du package stringr pour renvoyer un mot complet en spécifiant son indice. Des exemples d'application sont fournit un peu plus bas.

Exemple: utilisation de la fonction substr() et str_sub()

```
x <- "Ceci est une chaine de caractères pour faire le test"
```

1.13.10.1. Utilisation de la fonction substr()

```
substr(x, 1, 4) # Renvoie les chaines entre l'indice 1 et 4. Ici "Ceci"
substr(x, 1, 1) # renvoie le caractère d'indice 1 . Ici "C"
substr(x, 6,nchar(x)) # renvoie tous les caractères à partir du 6ième (inclu).
```

1.13.10.2. Utilisation de la fonction str_sub() du package stringr

La fonction str_sub() s'appuie sur la fonction substr() du package base en proposant quelques améliorations. Exemples :

```
library(stringr)
str_sub(x, 1, 4) # Renvoie les chaines entre l'indice 1 et 4. Ici "Ceci"
str_sub(x, 1, 1) # renvoie le caractère d'indice 1 . Ici "C"
str_sub(x, 6,nchar(x)) # renvoie tous les caractères à partir du 6ième (inclu).
str_sub(x, 6,) # renvoie tous les caractères à partir du 6ième (formulation non possible avec substr)
str_sub(x, ,10) # renvoie tous les caractères avant le 10ième (formulation non possible avec substr)
```

Nb: La manière la plus correcte pour faire l'indiçage avec la fonction str_sub() est de spécifier les mots clés string, start et end comme suit:

```
str_sub(string = x, start = 4, end = 8) # start est le début de l'indiçage et end la fin.
```

NB: lorsque l'objet spécifiée comme argument de la fonction srt_sub() est un vecteur, alors l'indiçage (ou l'extraction) est effectuée sur chaque élément. Exemple:

```
x<-c("Rouge", "Vert", "Bleu")
str_sub(string = x, start =2,  end =3) # renvoie ## [1] "ou" "er" "le"
```

Indiçage négatif

En fournissant aux paramètres start et end des valeurs négatives, on indique à R de lire la chaîne à l'envers :

```
str_sub(string = x, start = 4, end = 8) # "i est"
str_sub(string = x, start = -13, end = -1) # Compter à partir du dernier caractère (-1)
str_sub(string = x, start = -13) # équivalent à la précédente formulation.
```

1.13.10.3. Utilisation de la fonction word() du package stringr

La fonction word() extrait les mots en fonction de leur indice dans la chaine. Ex:

```
x <- "Ceci est une chaine de caractères pour faire le test"
word(x, 1) # extrait le premier mot
word(x, 3) # extrait le troisième mot
```

```
word(x, 1:3, -1) # Du premier au dernier mot, du second au dernier, et du troisième au dernier  pour le premier élément de phrase
word(x, 1, 1:3) # Premier mot, Premier et second mot, Premier et troisième mot # pour le second élément de phrase
```

1.13.11. Modifier les éléments d'une chaine de caractères en se basant sur leur indice

Pour modifier un ou plusieurs éléments dans une chaine de caractères on utilise l'opérateur d'assignation tout en spécifiant la tranche de la chaine à modifier. Exemple:

```
x <- "Debt is one person's liability, but another person's asset."
str_sub(x, 1, 4) <- "Credit" # Remplace les caractères compris entre 1 et 4 par Credit
```

Lorsque x est une liste, l'assignation est faite sur chaque élément. Ex:

```
x<-c("Rouge", "Vert", "Bleu")
str_sub(x, 1, 2) <- "color" # renvoie "coloruge" "colorrt" "coloreu"
```

Attention : Lors du remplacement, R peut avoir recours au recyclage lorsque par exemples la valeur à assigner est aussi un vector dont la dimension n'est pas la même que la dimension de x. ex:

```
x <- c("Rouge", "Vert", "Bleu")
str_sub(x, 2, 3) <- c("!!", "@@")   ## [1] "R!!ge" "V@@t" "B!!u" avec un message d'erreur.
```

Remarque:

Dans le cas du remplacement d'une chaîne extraire par une autre, on peut distinguer trois cas :

Lorsque la chaîne de remplacement est de même longueur que celle extraite alors les fonctions str_sub() et substr() donnent le même résultat ;

Lorsque la chaîne de remplacement est plus courte que celle extraite : avec substr(), la chaîne de remplacement est complétée par la fin de celle extraite, tandis qu'avec str_sub(), la chaîne extraite est intégralement remplacée par celle de remplacement ;

Lorsque la chaîne de remplacement est plus longue que celle extraite : avec substr(), la chaîne de remplacement est tronquée, tandis qu'elle ne l'est pas avec str_sub().

Au final, on retient que la fonction substr() effectue un tronquage alors que la fonction str_sub effectue un remplacement simple. Exemples :

```
texte1 <- "le train de tes injures roule sur le rail de mon indifférence"
texte2 <- texte1 # On copie le contenu de texte dans une nouvelle variable
str_sub(string = texte1, start = 17, end = 23) # Remplacement plus court que la chaîne extraite "injures"
str_sub(string = texte1, start = 17, end = 23) <- "jurons"
substr(x = texte2, start = 17, stop = 23) <- "jurons"
texte1 ## [1] "le train de tes jurons roule sur le rail de mon indifférence"
texte2 ## [1] "le train de tes jurons roule sur le rail de mon indifférence"
str_sub(string = texte, start = 1, end = 8) # Remplacement plus long que la chaîne extraite ## [1] "le train"
str_sub(string = texte, start = 1, end = 8) <- "la locomotive"
substr(x = texte_2, start = 1, stop = 8) <- "la locomotive"
texte ## [1] "la locomotive de tes jurons roule sur le rail de mon indifférence"
texte_2 ## [1] "la locom de tes juronss roule sur le rail de mon indifférence"
```

1.13.12. Vérifier si un caractère (ou un motif) existe dans une chaine de caractères

Pour rechercher un motif (un pattern) dans une chaine de caractères, on utilise les fonction grepl() du package base. On peut aussi utiliser la fonction str_detect() package stringr. Les exemples ci-dessous illustrent l'utilisation de ces fonctions :

Exemple 1: soit la chaine suivante:

```
x<-"Bonjour Monsieur"
```

On va vérifier si cette chaine contient les motifs o, k et z. On a:

1.13.12.1. Utilisation de la fonction grepl()

```
grepl( pattern = "o", x=x,ignore.case = FALSE) # Renvoie True
grepl( pattern = "k", x=x,ignore.case = FALSE) # Renvoie FALSE
grepl( pattern = "z", x=x,ignore.case = FALSE) # Renvoie FALSE
```

1.13.12.2. Utilisation de la fonction stringr()

```
library(stringr)
str_detect(string = x, pattern = "o",ignore.case = FALSE) # Renvoie True
Exemple 2: Cas où la chaine est  un vecteur :
x<-c("Pomme", "Poire", "Ananas")
```

```
grepl( pattern = "o", x=x,ignore.case = FALSE) # Renvoie  TRUE
TRUE FALSE
library(stringr)
str_detect(string = x, pattern = "o",ignore.case = FALSE) #
Renvoie  TRUE  TRUE FALSE
```

Dans le cas où la chaine est un vecteur, la vérification est effectuée élément par élément.

1.13.13. Rechercher et remplacer un caractère (un motif) dans une chaine de caractères

Pour rechercher et remplacer un motif dans une chaine de caractères, on utilise la fonction sub() ou gsub() du package de base. On peut aussi utiliser la fonction str_replace() ou str_replace_all() du package stringr. Chacune de ces fonctions effectue un type de remplacement particulier. Par exemple, les fonctions sub() et str_replace() remplacent la première occurrence du motif dans un intervalle donné alors que les fonctions gsub() et str_replace_all() remplacent toutes les occurrences du motifs dans cet intervalle. Ci-dessous les détails sur chaque fonction.

1.13.13.1. Remplacement de la première occurrence du motif

Utilisation de la fonction sub ()

Exemple:

```
x<-c("Pomme", "Poire", "Ananas")
sub(pattern = "a", replacement = "@@", x, ignore.case = FALSE)
# Recherche tous les a (minuscule) et remplace la première
occurrence par @@ ## [1] "Pomme" "Poire" "An@@nas"
sub(pattern = "a", replacement = "@@", x, ignore.case = TRUE)
# Recherche tous les a (minuscule et majuscule) et remplace la
première occurrence par @@   "Pomme"   "Poire"   "@@nanas"
```

Utilisation de la fonction str_replace()

```
x<-c("Pomme", "Poire", "Ananas")
library(stringr)
str_replace(string =x , pattern = "a", replacement = "@@") #
Recherche tous les a (minuscule) et remplace la première
occurrence par @@ ## [1] "Pomme" "Poire" "An@@nas"
str_replace(string   =x   ,   pattern   =   ignore.case("a"),
replacement = "@@") # Avec la fonction ignore.case().
```

1.13.13.2. Remplacement de toutes les occurrences du motif

Utilisation de la fonction gsub ()

Exemple :

```
x<-c("Pomme", "Poire", "Ananas")
gsub(pattern = "a", replacement = "@@", x, ignore.case = FALSE) # # "Pomme"    "Poire"    "An@@n@@s"
gsub(pattern = "a", replacement = "@@", x, ignore.case = TRUE) #  "Pomme"     "Poire"     "@@n@@n@@s"
```

Utilisation de la fonction str_replace_all ()

```
library(stringr)
x<-c("Pomme", "Poire", "Ananas")
str_replace_all(string = x, pattern = "a", replacement = "@@") #  renvoie "Pomme" "Poire" "An@@n@@s"
str_replace_all(string = x, pattern = ignore.case("a"), replacement = "@@") # renvoie "Pomme" "Poire" "@@n@@n@@s"
```

NB: Dans le cas de la recherche de motif, il est souvent de bonne pratique de convertir d'abord l'ensemble de la chaine en minuscule ou majuscule avant de faire les recherches.

1.13.14. Découper une chaine de caractères en des éléments distincts en fonction d'un séparateur

Pour découper une chaine de caractères en des mots séparés, on utilise les fonctions strsplit() du package base. On peut aussi utiliser les fonctions str_split() du package stringr. Exemple : Soit la chaine suivante:

x = "Ceci est une chaine de caractère conçue pour les tests"

1.13.14.1. Utilisation de la fonction strsplit()

```
strsplit(x ," ") # fait du split en fonction de l'espace.
```

1.13.14.2. Utilisation de la fonction str_split() du package stringr

```
library(stringr)
str_split(string = x, pattern = " ") # split en fonction de l'espace
```

1.13.15. Suppression des espaces et des caractères spéciaux dans une chaine: la fonction str_trim() du package stringr

Pour retirer des caractères vides comme les espaces, les sauts de ligne, ou les retours à la ligne, on peut utiliser la fonction str_trim() du package stringr. Cette fonction permet d'éliminer tous les caractères blancs à gauche et à droite d'une chaîne de caractères. Exemple :

```
texte <- c("\n\nPardon, du sucre ?", "Oui, seize \n ",
"...\t\t...\t")
library(stringr)
str_trim(texte, side = "left")
str_trim(texte, side = "right")
str_trim(texte, side = "both")
```

1.13.16. Créer des objets indicés, préfixés ou suffixés : combinaison de la fonction get() avec la fonction str_c()

La fonction get() permet d'accéder à la valeur d'une variable en renseignant son nom en utilisant la fonction paste() ou la fonction str_c().

Exemple 1: Utilisation de la fonction str_c() du packake stringr

On sait que str_c("variable_", 1) renvoie "variable_1".

Définissons par exemple une variable nommée "variable_1" telle que:

```
variable_1 <- 5
```

Pour afficher la valeur de la variable en utilisant la fonction str_c() et la fonction get(), on fait:

```
get(str_c("variable_", 1)) # ce qui renvoie 5
```

NB: La combinaison de la fonction str_c() et de la fonction get() peut s'avérer utile dans de nombreuses situations notamment les opérations en boucles sur les variables indicées. Par exemple, on souhaite créer 5 variables prenant chacune un nombre aléatoire uniforme tiré entre 50 et 200. Par la suite, on souhaite afficher la valeur de chacune en utilisant la fonction print(). On va alors suivre la démarche suivante:

```
for (i in 1:5) {
    assign(str_c("myVar",   i),runif(1,50,200))   #   première
méthode
    #assign(paste("myVar", i, sep=""),runif(50,200)) # méthode
alternative: utilisation de la fonction paste
}
```

Cette première partie de l'exercice utilise la fonction str_c() entre le suffixe et le compteur pour former une chaine. Cette chaine est utilisée comme nom de variable en lui assignant avec la fonction assign() l'expression runif(1,50,200) qui tire 1 nombre aléatoire entre 50 et 200.

La méthode alternative à str_c() est la fonction paste(). On peut aussi utiliser paste0()

La seconde partie de l'exercice va consister à afficher les valeurs des 5 variables créées en utilisant une boucle. Pour cela, on combine la fonction get() et str_c()

```
for (i in 1:5) {
    print(get(str_c("myVar", i))) # première méthode
    #print(get(paste("myVar",    i,    sep=""))))    #    méthode
alternative: utilisation de la fonction paste
}
```
La combinaison de la fonction get() et de la fonction str_c() permet donc d'accéder aux valeurs des variables myVar1-myVar5.

L'exemple ci-dessus montre également que la fonction paste() est une alternative à la fonction str_c().

N'oublier pas également le rôle de la fonction assign() qui est ici un substitut à l'opérateur <- dans des cas où celui-ci n'est pas utilisable.

NB : Noter aussi qu'on peut aussi utiliser la fonction eval(parse()) pour faire des assignations ou des formules.

Exemples:

```
x <- array(seq(1,18,by=1),dim=c(3,2,3))
for (i in 1:length(x[1,1,])) {
  eval(parse(paste(letters[i],"<-mean(x[,,",i,"])",sep="")
}
```
Exemple 2: **Utilisation de la fonction paste() et get() du package base**

Soit p et q les ordres maximums de retard des termes AR et MA d'un modèle ARMA. On souhaite faire des estimations de modèle ARMA en combinant toutes les valeurs possibles de p (allant de 0 à p) et de q (allant de 0 à q) avec p=5 et q=3. Chaque modèle estimé porte un nom commposé comme suit : estimation_i_j où i fait référence à la valeur prise par p et j la valeur prise par q. On va alors adopter une boucle comme suit :

```
for (i in 0:p) {
  for (j in 0:q) {
    assign(paste("estimation",i,          j,          sep="_"),
arima(dpbrent,order=c(i, 0, j))) # estimater un modèle arima
et stocker ses résultats dans une variable nommée
estimation_p_q
    results<-(get(paste("estimation",i,    j,    sep="_")))    #
récupération des valeurs stockées dans la variable
    aic<-results$aic
    print(paste("p=",i,  "  ",  "q=",  j,"  ",  "aic=",aic  ,
sep=""))
  }
}
```
Pour supprimer une variable créée par concaténation on utilise rm(list=) sur le nom composé comme lors de la création. Exemples:

```
i<-4
rm(list=(paste("estimation",i, sep="_"))) # supprimer l'objet estimation nommé « estimation_4 ».
```
On peut mettre cet exemple à l'intérieur d'une boucle.

1.13.17. L'utilisation de l'opérateur antislash \

Le caractère \ (barre oblique inversée, ou backslash) est un opérateur très important dans le traitement des variables caractères. D'abord, il joue le rôle de caractère d'échappement c'est-à-dire qu'il permet d'afficher certains caractères comme les guillemets, qui autrement, sont ignorés par R. Il permet aussi jouer le rôle de caractère de contrôle pour les tabulations, les sauts de ligne, etc. Ci-dessous quelques détails sur le rôle de l'antislash \ :

\n insère une nouvelle ligne

\r insère un retour à la ligne

\t insère une tabulation

\b Effectue Retour arrière

\\ Echappe une barre oblique inversée

\' Echappe une Apostrophe ou un guillemet simple

\" Echappe une Apostrophe double ou un guillemet double

\' Echappe Accent grave

1.13.18. Quelques fonctions utiles du package stringr pour le traitement des chaines de caractères.

Le tableau ci-dessous présentent quelques fonctions utiles du package stringr pour le traitement des chaines de caractères

Fonction	Rôle
str_detect()	Détecter la présence d'un motif dans une chaîne
str_extract()	Extraire la première occurrence d'une chaîne contenant un motif
str_extract_all()	Extraire toutes les occurrences d'une chaîne contenant un motif
str_replace()	Remplacer la première occurrence d'un motif
str_replace_all()	Remplacer toutes les occurrences d'un motif
str_match()	Renvoyer la première chaine qui matche un motif
str_match_all()	Renvoyer toutes les chaines qui matchent un motif
str_locate()	Localiser la position de la première occurrence d'un motif dans une chaîne
str_locate_all()	Localiser les positions de toutes les occurrences d'un motif dans

	une chaîne
str_split()	Découper une chaîne de caractères en plusieurs morceaux
str_string_fixed()	Découper une chaîne de caractères en un nombre fixe de morceaux fen fonction d'un motif

1.14. Traitement des variables date sous R

1.14.1. Distinction des classes de date sous R :

On distingue deux principales classes pour représenter les variables de type date sous R : la classe Date et la classe POSIX.

- la classe Date est utilisée pour les dates formées de trois composantes Année-Mois-Jour.

Exemples: 25-09-2016, 16 Octobre 2015, 12/07/95,….

Cette classe Date stocke les valeurs des dates en nombres de jours depuis 1970-01-01, avec des valeurs négatives pour des dates antérieures à 1970-01-01.

- La classe POSIX est utilisée lorsque la date comporte des composantes supplémentaires que sont : l'heure, la minute et la seconde.

Exemples: 25-09-2016 15:12:00, 16 Octobre 2015 23:59:45, 12/07/95 09:30:15

La classe POSIX se distingue en deux sous-classes que sont POSIXct et POSIXlt

Avec la classe POSIXct, les dates sont stockées en secondes depuis 1970-01-01 01:00:00

Avec POSIXlt, les dates sont stockées sous forme de listes dont chaque élément correspond à une composante spécifique: année, mois, jour, heure, minute, seconde.

1.14.2. Conversion d'une chaîne de caractères en format date de classe Date

Pour convertir une chaîne de caractères en format date de classe Date, on utilise la fonction as.Date().

Exemples:

```
dateText1 <- c("2016-09-25", "2016-09-26", "2016-09-27", "2016-09-28") # Vecteur de chaînes de caractères
dateVal1 <- as.Date(dateText1, format="%Y-%m-%d") # Convertit dateText1 en format Date
```

```
dateText2 <- c("2016 09 25", "2016 09 26", "2016 09 27", "2016
09 28") # Vecteur de chaînes de caractères
dateVal2<- as.Date(dateText2, format = "%Y %m %d") # Convertit
dateText2 en format Date
dateText3 <- c("25 Septembre 2016", "26 Septembre 2016", "27
Septembre 2016")
dateVal3<- as.Date(dateText3, format = "%d %B, %Y") #
Convertit dateText3 en format Date
```

NB: Si le format est de type %Y-%m-%d ou %Y/%m/%d, il n'est pas nécessaire de renseigner le paramètre format de la fonction as.Date() car ces formats sont reconnus par défaut.

1.14.3. Conversion d'une chaîne de caractères en format date de classe POSIX

Pour convertir une chaîne de caractères en format date de classe POSIX, on utilise soit la fonction as.POSIXct(), soit la fonction POSIXlt().

Exemples:

```
dateText1 <- c("2016-09-25 11:30:00", "2016-09-25 11:35:00",
"2016-09-25 11:40:00") # un vecteur de chaînes de caractères
dateValp1 <- as.POSIXct(dateText1, format = "%Y-%m-%d
%H:%M:%S") # date de classe POSIXct
dateValp2 <- as.POSIXlt(dateText1, format = "%Y-%m-%d
%H:%M:%S")# date de classe POSIXlt
```

1.14.4. Les principaux formats de conversion des dates

L'option format des fonction as.Date() ou as.POSIX permet d'indiquer le format sous lequel se présente la date en tant que chaînes de caractères. Par exemple "%Y-%m-%d " signifie la date en texte se présente en format "Année-mois-jour". Ex: 2016-09-12.

Mais la date peut aussi se présenter sous divers autres formats qu'il va falloir indiquer lors de la conversion.

Le tableau suivant illustre les composantes les plus couramment rencontrées pour indiquer le format d'une date:

Fonctions	**Description**
Composantes: année	
%Y	Année; Ex: 2015
%y	Année sans le "siècle" (00-99) ; Ex: 15

%C	La composante siècle de l'année (partie entière de la division de l'année par 100). Ex: 20 pour 2000
Composantes: mois	
%b	Abréviation du nom du mois ; Ex: Oct pour Octobre
%h	De même chose que %b ; Ex: Oct
%B	Nom complet du mois. Ex: Octobre
%m	Mois (01-12) ; Ex: 10
Composantes: semaine	
%U	Semaine de l'année (00-53), dimanche comme début de semaine; Ex:42
%V	Semaine de l'année (00-53).lundi comme début de semaine;
%W	Semaine de l'année (00-53), lundi comme début de semaine; le premier lundi de l'année définit la semaine1 ; Ex:42
Composantes: jours	
%a	Abréviation du nom du jour. Ex: Mer pour Mercredi
%A	Nom complet du jour. Ex: Mercredi
%d	Jour du mois (01-31) ; Ex: 21
%e	Jour du mois en nombre décimal (1-31) ; Ex: 21
%j	Jour de l'année (001-366) ; Ex: 294
%u	Jour de la semaine (1-7), commence le lundi ; Ex: 3
%w	Jour de la semaine (0-6), dimanche étant 0 ; Ex: 3
Composantes: date complète	
%c	Date et heure au format %a %e %b %H:%M:%S %Y ; Ex: Mer 21 oct 13:55:44 2015
%D	Date au format %m/%d/%y ; Ex: 10/21/15
%F	Date au format %Y-%m-%d ; Ex: 2015-10-21
%x	Date (dépend du lieu) ; Ex: 21.10.2015
Composantes: heure	
%H	Heure (00-24) ; Ex: 13
%I	Heure (01-12) ; Ex: 01
%r	Heure au format 12 AM/PM ; Ex: 01:55:44 PM

%T	Heure au format 24 %H:%M:%S ; Ex: 13:55:44
%R	Même chose que %H:%M ; Ex: 55
%X	Heure ; Ex: 13:55:44
%p	AM/PM ; Ex: PM
%z	offset en heures et minutes par rapport au temps UTC ; Ex: +0200
%Z	Abréviation du fuseau horaire (en output seulement) ; Ex: CEST
Composantes: minutes et secondes	
%M	Minute (00-59) ; Ex: 55
%S	Seconde (00-61) ; Ex: 44
Autres composantes: caractères spéciaux	
%n	Retour à la ligne en output, caractère blanc en input
%t	Tabulation en output, caractère blanc en input

1.14.5. Extraction des éléments d'une date (année, mois, jour, etc..)

Extraction des éléments d'une date de format Date

Pour extraire les éléments d'une date (année, mois, jours), on utilise la fonction format() en indiquant l'élément à extraire. Exemple :

```
format(dateVar,'%Y')# extrait la partie année
format(dateVar,'%m') # Extrait la partie mois
format(dateVar,'%d') # Extrait le jour
```

Extraction des éléments d'une date de format POSIXt

Lorsqu'une date est stockée sous le format POSIXlt, on peut extraire les éléments années, mois, jour, heure, minute, seconde, etc… en utilisant un mot clé. Ci-dessous les différents mots clés permettant d'accéder aux éléments d'une date POSIXlt.

- sec : secondes (0-61)
- min : minutes (0-59)
- hour : heures (0-23)
- mday : jour du mois (1-31)
- mon : mois après le premier de l'année (0-11)
- year : années depuis 1900
- wday : jour de la semaine (0-6), début de semaine le dimanche
- yday : jour de l'année (0-365)

- isdst : indicateur d'heure d'été (positif si appliquable, zéro sinon ; négatif si inconnu)
- zone : Abréviation du fuseau horaire local ("" si inconnu, ou UTC)
- gmtoff : décalage en minutes par rapport au temps GMT (NA ou 0 si inconnu)

1.14.6. Opérations arithmétiques sur les dates

NB : Pour ajouter ou soustraire des durées de temps, il faut se rappeler comment sont stockées les dates. Avec les dates de classe Date, il s'agit d'un nombre de jours. Aussi, additionner un nombre n à un objet de type date retourne la date n jours plus tard. Pour les objets de classe POSIXct ou POSIXlt, comme R stocke la date en secondes, l'ajout d'un nombre n retourne la date augmentée de n secondes.

Exemples : Créons d'abord quelques variables de dates de classe Date ou POSIX.

```
d_date_1 <- as.Date("2014-09-01")
d_date_2 <- as.Date("2015-10-21")
d_posix_ct_1 <- as.POSIXct("2014-09-01 18:32:28")
d_posix_ct_2 <- as.POSIXct("2015-10-21 13:55:44")
d_posix_lt_1 <- as.POSIXlt("2014-09-01 18:32:28")
d_posix_lt_2 <- as.POSIXlt("2015-10-21 13:55:44")
```

1.14.6.1. Addition ou soustraction d'une valeur

```
d_date_2 + 10 ## [1] "2015-10-31"
d_date_2 - 10 ## [1] "2015-10-11"
d_posix_ct_2 ## [1] "2015-10-21 13:55:44 CEST"
d_posix_ct_2 + 10 ## [1] "2015-10-21 13:55:54 CEST"
d_posix_ct_2 - 10 ## [1] "2015-10-21 13:55:34 CEST"
```

Toujours dû à ce stockage interne en jours (Date) ou secondes (POSIX), il est possible de comparer facilement deux dates entre-elles, avec les opérateurs classiques.

1.14.6.2. Opérations logiques

```
d_date_1 > d_date_2 ## [1] FALSE
d_posix_ct_2 > d_posix_ct_1 ## [1] TRUE
d_posix_lt_2 == d_posix_lt_1 ## [1] FALSE
```

1.14.6.3. Soustraction de deux dates : l'opérateur «-» ou la fonction difftime()

Pour obtenir la différence entre deux dates, on peut utiliser l'opérateur " -" . Dans les cas des POSIXlt et POSIXlt, il est également possible d'utiliser la fonction difftime(), en indiquant les deux dates en paramètres, et éventuellement en renseignant le

paramètre units pour indiquer l'unité désirée pour la différence. Voici ci-après répertorié les valeurs possibles pour le paramètre units.

- auto: le choix de l'unité pour que celle-ci soit la plus large possible (à l'exception de weeks)
- secs: secondes
- mins: minutes
- hours: heures
- days: jours (périodes de 24 heures)
- weeks: semaines

Exemples avec *des dates de classe Date*

```
d_date_2 - d_date_1 ## Time difference of 415 days
d_posix_ct_2 - d_posix_ct_1   ## Time difference of 414.8078 days
```

Exemples avec *des dates de classe* POSIXct

```
difftime(d_posix_ct_2, d_posix_ct_1, units = "hours") ## Time difference of 9955.388 hours
round(difftime(d_posix_ct_2, d_posix_ct_1, units = "hours")) ## Time difference of 9955 hours
```

Exemples avec *des dates de classe* POSIXlt

```
d_posix_lt_2 - d_posix_lt_1 ## Time difference of 414.8078 days
difftime(d_posix_lt_2, d_posix_lt_1, units = "week") ## Time difference of 59.25826 weeks
```

Utilisation de la fonction difftime ()

Cette fonction permet d'effectuer la différence entre deux dates. Exemple : soit le vecteur de dates définit comme suit:

```
datesstring <- c("17/02/92", "27/02/92")
mydates <- as.Date(datesstring, "%d/%m/%y")
difftime(mydates[1], mydates[2]) # Calcul la différence entre l'élément d'indice 1 et 2 et renvoie en nombre de jours
difftime(dates[1], dates[2], units = "s") # Renvoie la différence en nombre de secondes.
```

1.14.6.4. Générer des séquences de date: la fonction seq()

Cette fonction s'utilise avec les arguments from, to et by.

Exemple:

```
d1 <- seq(from = as.Date("01/11/08", "%d/%m/%y"), to = as.Date("15/11/08","%d/%m/%y"), by = "day")
```

Les arguments from et by sont spécifiés sous forme de date avec leur format. Et l'argument by indique le pas en jour (day), mois (month), année(year).

1.14.7. Traitement des variables date avec le package lubricate

lubridate est un module très puissant de traitement de date sous R. Il permet d'effectuer diverses opérations de traitement des dates (extraction de partie d'une date, conversion de format, etc...). Consulter la documentation R sur le package.

Chapitre 2 : Traitement et organisation des données

Ce chapitre présente les différentes opérations de traitement et d'organisation couramment effectées sur les données avant les analyses.

Les bases de données étant généralement présentées sous forme de data frames sous R, de nombreuses opérations de traitement ont été déjà abordées au chapitre 1 dans la section consacrée à l'étude de l'objet data frame. Il est donc fortement conseillé de lire cette section avant d'entamer ce présent chapitre.

Ce chapitre va se consacrer essentiellement aux traitements des variables ainsi pour la constitution d'une base de données prête pour les analyses.

2.1. Initialisation de la phase de traitement

2.1.1. Fixation du répertoire du travail

Le répertoire de travail pour la lecture et l'écriture éventuelle des tables de donnés se fait avec la fonction setwd(). Exemple :

```
setwd("C:/R-applications/data") # Oriente R vers le dossier data situé dans le dossier R-application situé à son tour sur le disque C.
```

2.1.2. Rappel sur la lecture et l'écriture de fichiers externes de données

Ci-dessous quelques exemples de lecture et d'écriture de fichiers externes :

Importation de fichier texte avec séparateur tabulation \t

```
mydata<-
read.table("mytabdelimdata.txt",header=T,dec=".",sep="\t")    #
importation de fichier texte avec séparateur tabulation \t
```

Importation du fichier texte avec séparateur csv

```
mydata<-read.table("mycsvdata.csv",header=T,dec=".", sep=";")
head(mydata1, n=5L) # Affiche les 5 premières lignes
```

Importation d'un fichier de données ne contenant pas les noms des variables

```
mydata<-
read.table("mycsvdata.csv",header=FALSE,dec=".",sep=";")
colnames(mydata) <- c("annee","mois", "ihpc", "datapubli")
```

Lecture de données à partir du tableau Microsofst Excel : package xlsx.

```
library(xlsx)
```

```
mydata<-
read.xlsx("dataWorkbook.xlsx",sheetName="patientdata",startRow
=1,endRow=355,colIndex=c(1:22),    header=TRUE,  as.data.frame=
TRUE)
```

Exportation du data frame vers des formats texte avec séparateur : la fonction write.table()

```
write.table(mydata  , file = "mydata_exp.csv", append = FALSE,
sep = ";", col.names = T, dec = "." ,qmethod = "double" ) #
Exportation vers un fichier csv
```

1.9.8.2. Exportation vers un fichier Microsoft Excel : fonctions write.xlsx()du package xlsx

```
library(xlsx)
write.xlsx(mydata,    file="myExcelData.xlsx",sheetName="data",
append=FALSE)
```

2.1.3. Description du contenu de la base de données

2.1.3.1. Lister toutes les variables la table

```
names(mydata)
```

Pour voir la liste des variables et leur type on utilise la fonction str().

```
str(mydata)
```

2.1.3.2 Visualiser le contenu de la table de données

```
view(mydata) # Aperçu des données
fix(mydata)      #   Visualiser   et   éditer   la   table   avec
enregistrement automatique des modifications
edit(mydata) # Editer la table sans enregistrement automatique
des modifications
```

2.1.3.3. Faire un codebook des variables

```
install.packages("Hmisc")
library("Hmisc")
describe(mydata)
```

2.2. Déclarer et Structurer les variables selon leur type

2.2.1. Déclarer et structurer les variables qualitatives nominales

Exemples :

```
sexe <- as.factor(sexe) # Traite la variable sexe comme une
variable catégorielle (initialement codée 1 et 2)
```

```
levels(sexe) <- c("Homme","Femme") # Attribue des values labels aux variables
sexe <- relevel(sexe, ref = 1 ) # Considère l'homme comme modalité de référence
```

2.2.2. Déclarer et structurer les variables qualitatives nominales

```
satisfaction <- as.ordered(satisfaction) # Valeurs initiale sont "Pas satisfait", "Satisfait" et "Très satisfait"
levels(satisfaction) <- c("Pas satisfait", "Satisfait", "Très satisfait"))
```

2.2.3. Déclarer et structurer les variables quantitatives discrètes

```
nbre_enfants <- as.integer(nbre_enfants)
```

2.2.4. Déclarer et Structurer les variables quantitatives continues

```
poids <- as.double(poids)
```

2.3. Création et modification de variable

2.3.1. Créer une nouvelle variable

Exemples : Considérons la table mydata1

Créons une variable égale au carré de la variable duresurfa. Nous allons nommer cette variable duresurfa2 Pour cela, nous avons trois méthodes:

Methode1:

```
attach(mydata1)
mydata1$duresurfa2 <- duresurfa^2 # (peu explicite) # nécessite par conséquent l'utilisation de la fonction attach().
detach(mydata1)
```

Methode2:

```
mydata1$duresurfa2 <- mydata1$duresurfa^2 # (très explicite). Ne nécessite donc pas l'utilisation de la fonction attach.
```

Methode3: utilisation de la fonction transform().

```
mydata1 <- transform( mydata1,duresurfa2 = duresurfa^2 ) # Méthode recommandée
```

Comme on peut le constater, la méthode 3 utilise la fonction transform() qui est beaucoup plus générale. En effet, dans la fonction transform(), pour créer ou modifier plusieurs variables, on sépare leur expression par des virgules. Par exemple:

```
mydata1 <- transform( mydata1,duresurfa2 = duresurfa^2, duresurfa3 = duresurfa^3,duresurfa4 = duresurfa^4)
```

2.3.2. Modifier une variable existante

On peut également utiliser la fonction transform() pour modifier une variable existante. Exemple :

```
mydata1 <- transform( mydata1,duresurfa2 = duresurfa2*60) # multiplie duresurfa2 par 60.
```

2.3.3. Modifier la valeur d'une variable selon une condition

Pour modifier la valeur d'une variable on utilise l'opérateur de subsetting [...] en y incluant la condition.

Exemple : On va modifier la variable IMC selon les catégories de valeurs. La règle de modification est la suivante: Si l'âge est inférieur à 30, on multiplie l'IMC arbitrairement par 2. Et si l'âge est entre 30 et 40, on mutiplie ICM par 3. Et pour les autres valeurs de l'âge, on multiplie IMC par 4.

Ces modifications seront faites comme suit :

```
mydata$IMC[age<30]<-mydata$IMC[age<30]*2
mydata$IMC[age>=30 & age<40]<-mydata$IMC[age>=30 & age<40]*3
mydata$IMC[age>=40]<-mydata$IMC[age>=40]*4
```

Il faut noter que la condition doit accompagner toutes les variables apparaissant dans la ligne de commande mais aussi dans l'expression de la formule. Pour l'égalité, il faut utiliser == plutôt que = et pour l'opérateur ou, utiliser le symbole |.

2.3.4. Création de variables indicées

Pour avoir plus de détails sur la création de variables indicées, nous vous recommandons de lire d'abord le chapitre 1 dans la section consacrée à la création d'objets indicés dans la section dédiée aux traitements des variables en chaînes de caractères. Nous présentons ici un résumé de ce qui a été présenté dans cette section.

Pour la création et la gestion des variables indicées, nous allons nous baser essentiellement sur les fonction paste(), paste0(), get() du package base mais aussi la fonction str_c() du package stringr.

La fonction get() permet d'accéder à la valeur d'une variable en renseignant son nom en utilisant la fonction paste() ou la fonction str_c().

Exemple: Définissons une variable nommée "variable_1" qui prend 5. On peut utiliser les méthodes suivantes :

Méthode 1 : Fonction paste()

```
assign(paste("variable_", 1,sep=""),5)
print(variable_1) # affiche la valeur de variable_1
get(paste("variable_", 1,sep="")) # Récupère la valeur de la variable variable_1
```

Méthode 2 : Fonction paste0()

```
assign(paste0("variable_", 1),5)
print(variable_1) # affiche la valeur de variable_1
get(paste(paste0("variable_", 1)) # Récupère la valeur de la variable variable_1
```

Méthode 1 : Fonction str_c() de stringr

```
library(stringr)
assign(str_c("variable_", 1),5)
print(variable_1) # affiche la valeur de variable_1
get(paste(str_c("variable_", 1)) # Récupère la valeur de la variable variable_1
```

2.4. Recodage des variables

Pour recoder une variable en une autre variable, on utilise les opérateurs de subsetting à l'intérieur desquels on indique les conditions et auquel on assigne les codes. On peut aussi utiliser la clause « ifelse ». Par exemple, on veut recode l'âge en classe d'âge dans la table mydata.

2.4.1. Utilisation de l'opérateur [ ]

2.4.1.1. Recodage en variable catégorielle en caractères

Exemple : Recodage de la variable age

```
attach(mydata)
mydata$class_age[age > 75] <- "Elder"
mydata$class_age[age > 45 & age <= 75] <- "Middle Aged"
mydata$class_age[age <= 45] <- "Young"
mydata$class_age <- as.factor(mydata$class_age) # convertit la variable class_age  en variable factor (variable catégorielle)
detach(mydata)
```

Pour l'égalité, il faut utiliser == plutôt que =et pour l'opérateur ou, utilier |

2.4.1.2. Recodage en variable catégorielle numérique

Exemple : Calculons la variable plus10ans à partir des deux variables Q17.3 et Q17.4

```
attach(mydata)
mydata$plus10ans[Q17.3 == 1 | Q17.4 == 1] <- 1
```

```
mydata$plus10ans[Q17.3 == 0 & Q17.4 == 0] <- 0
mydata$plus10ans <- as.factor(mydata$plus10ans) # convertit la variable class_age  en variable factor (variable catégorielle)
detach(mydata)
```

2.4.2. Utilisation de la clause « ifelse »

On peut aussi utiliser la fonction ifelse() pour faire le recodage des variables. Exemple :

```
mydata$class_age <- ifelse( mydata$age > 75, "Elder", ifelse(mydata$age > 45 & mydata$age <= 75 , "Middle Aged", "younger") )
```

Attention à l'emplacement des « ifelse » et le nombre de parenthèses. A chaque ifelse correspond une nouvelle parenthèse. Le code "autre" doit toujours être spécifié dans le dernier ifelse. C'est le cas ici de "younger". Sinon on pouvait aussi modifier le recodage comme suit:

```
mydata$class_age <- ifelse( mydata$age > 75, "Elder", ifelse(mydata$age > 45 & mydata$age <= 75 , "Elder", ifelse(mydata$age <= 45 , "younger", "NA")) )
```

En règle générale, le nombre de parenthèses à fermer est égal au nombre de ifelse déclaré (hormis la cause "autre").

2.5. Renommer les variables dans un data frame

2.5.1. Méthode 1 : Nommage par l'intitulé ou par l'indice

```
names(mydata)[names(mydata)=="myOldVar"] <- "myNewvar" # permet de renommer en spécifiant l'ancien et le nouveau nom
names(mydata)[3] <- "myNewVar" # change le nom de la troisième colonne en myNewVar
```

Ces méthodes sont applicables sur tous les objets nommables sous R comme les vecteurs ou les matrices

2.5.2. Méthode 2 : Renommage par l'utilisation de la fonction rename.vars() du package gdata

On peut aussi la fonction rename.vars() pour renommer une variable. Exemple : renommer la variable dursurf2 en dursurf_squared dans la table mydata1.

```
library(gdata)
mydata <- rename.vars(mydata, "dursurf2", "dursurf_squared" )
```

Pour renommer plusieurs variables, on crée un vecteur pour les variables anciens nom et un vecteur pour les nouveaux noms. Exemple:

```
mydata <- rename.vars(mydata,
c("dursurf2","dursurf3","dursurf4"),
c("dursurf_power2","dursurf_power3","dursurf_power4"))
```

2.6. Supprimer une variable dans une table

Il existe plusieurs façons de supprimer une variable dans un data frame. On peut noter en particulier la méthode de suppression par sélection (subsetting) ou par l'utilisation de la fonction remove.var().

2.6.1. Utilisation de l'opérateur de subsetting [ ]

La première fonction pour supprimer des variables dans une table de données est d'utiliser l'opérateur **[]**

Exemple : Créons une table contenant les variables Jour, duretot, taille et duresurfa à partir de la table mydata1. Les étapes sont les suivantes:

Etape 1: Créons un vecteur qui contient la liste des variables à garder

```
myvars<-c("Jour", "duretot", "taille", "duresurfa")
```

Etape 2: Création de la base contenant les variables sélectionnées

```
mydata<-mydata[myvars]
```

Lorsque la liste des variables est constituée des noms successives tels que: var1, var2,..., varn, alors on peut utiliser la fonction paste() pour constituer la sélection.

Par exemple pour 10 variables nommer var1 à var10, on fait:

```
myvars <- paste("var", 1:10, sep="")
mydata<- mydata[myvars]
```

Lorsque les variables ne sont pas numérotées, mais si l'on connait leur ordre, on peut procéder la sélection en se référant à ces rangs. Par exemple pour sélectionner les 3ième, 5ième et 10ième variables, on fait la liste:

```
mydata <- mydata[c(3,5,10)]
```

On peut aussi incrémenter la sélection lorsque les variables sont successives. Par exemple, pour sélectionner la de la première à la dixième variable, on a:

```
mydata <- mydata[c(1:10)]
```

On peut aussi combiner les deux modes. Par exemple:

```
mydata <- mydata[c(1,4:10)]
```

Au lieu de garder les variables de la liste constituée, on peut aussi les exclure pour garder les variables restantes. Pour cela, on utilise l'indiçage négatif. Exemple:

```
mydata <- mydata [-1] #Exclut la première variable (indice 1).
```

2.6.2. Utilisation de la fonction remove.var() du package gdata

On peut aussi utiliser la fonction remove.var() pour supprimer une variable dans un data frame.

Exemple:

```
library(gdata)
mydata <- remove.vars(mydata, c("duresurfa_power2", "duresurfa_power3"))
```

Supprime les deux variables duresurfa_power2 et duresurfa_power3.

2.7. Labéliser les noms des variables (variables labels)

La labélisation des noms des variables sous R n'est pas très directe. On se sert le plus souvent des packages additionnels. Par exemple le package Hmisc offre une possibilité pour labéliser les variables. Attention, toutefois, les labels crées ne prennent effet qu'avec les fonctions appartenant au package Hmisc.

```
install.packages("Hmisc")
library("Hmisc")
label(mydata$age) <- "Age de l'individu"
label(mydata$poids) <- "Poids de l'individu"
label(mydata$taille) <- "Taille de l'individu"
```

On peut ensuite utiliser la fonction describe() de Hmisc pour voir les détails des variables:

```
describe(mydata)
```

2.8. Labéliser les valeurs codées des variables (values labels)

Pour définir des values labels, on utilise la fonction as.factor() avec les options levels et labels. Exemple : Attribuer des labels aux codes de la variable IMC_cat

```
mydata$IMC_cat <- factor(mydata$IMC_cat, levels = c(1,2,3,4),labels = c("Sous-poids", "Normal", "Sur-poids", "Obésité"))
Description des variables
str(mydata)
library("Hmisc")
describe(mydata)
```

2.9. Sélection des observations: utilisation l'opérateur de subsetting [], de la fonction which() ou de la fonction subset()

Il existe plusieurs manières pour sélectionner les observations dans un data frame. On dénote en particulier l'utilisation de l'opérateur [], l'utilisation de la which() ou la fonction subset().

2.9.1. Sélection d'observations basée sur l'indice des observations : l'opérateur []

Pour ce type de sélection, on utilise l'opérateur [ ] en indiquant l'indice des observations à sélectionner ou à exclure. Attention, contrairement à la sélection des variables où [1] signifie sélection de la première variable, pour la sélection des observations, pour sélectionner la première observation on ajoute une virgule [1,]

Exemples:

```
mydata2 <- mydata1[1:10,] # Sélection des dix premières observations :
mydata2 <- mydata1[10:20,] # Sélectionner entre la 10ième et la 20 ième observation
```

2.9.2. Sélection d'observations basée sur les valeurs des variables (sélection conditionnelle).

Pour ce type de sélection, on peut utiliser la fonction which() avec l'opérateur de subsetting [...]. Exemple:

```
attach(newdata)
newdata <- mydata[ which(gender=='F' & age > 65),]
detach(newdata)
```

2.9.3. Sélection d'observations avec l'utilisation de la fonction subset()

La fonction subset() est beaucoup générale et beaucoup plus souple que les méthodes précédentes. Cette fonction permet de sélectionner en même temps les variables et les observations.

Exemple:

```
newdata <- subset(mydata, age >= 20 | age < 10, select=c(ID, Weight))
```

Ici, on sélectionne deux variables ID et Weight lorsque l'âge est inférieur à 10 ou supérieur à 20.

Exemple 2:

```
newdata <- subset(mydata, sex=="m" & age > 25,select=weight:income)
```
Ici on sélectionne les hommes de plus de 20 ans et on garde les variables consécutives allant de weight à income.

2.10. Trier les observations

Pour trier les observations, on utilise la commande order() en indiquant les variables de tri.

Exemple:

```
mydata <- mydata[order(mydata$country,decreasing = FALSE),] # Tri croissant sur une seule variable
mydata <- mydata[order(mydata$country, mydata$year,decreasing = FALSE),] # Tri croissant sur deux variables
mydata <- mydata[order(mydata$country, mydata$year,decreasing = TRUE),] # Tri décroissant
mydata <- mydata[order(mydata$annee,-mydata$mois, decreasing = FALSE),] # croissant sur annee et decroissant sur mois (utilisation de -)
mydata <- mydata[order(-mydata$annee,mydata$mois, decreasing = FALSE),] # décroissant sur annee et croissant sur mois
mydata <- mydata[order(-mydata$annee,-mydata$mois, decreasing = FALSE),] # décroissant sur les deux variable (équivalent à l'option decreasing=TRUE en enlevant les -).
```

2.11. Fusion de table de données (merge et append)

Les fonctions les plus basiques pour faire la fusion de data frame sont les fonctions rbind() et cbind(). Consulter la section sur l'étude de l'objet data frame dans le chapitre 1. Mais ces fonctions peuvent être limitées dans certaines situations en particulier la fonction rbind(). C'est pourquoi, il faut privilégier la fonction merge().

2.11.1 Fusion de tables ayant les mêmes observations et des variables différentes

Pour fusionner deux tables ayant des variables différentes mais ayant les mêmes clés d'identification, on utilise la fonction merge(). Exemple:

```
myddata <- merge(mydata1, mydata2, by=c("country","year"), all=TRUE)
```
Cette fonction est valable que ça soit la fusion 1 à 1 ou une fusion 1 à n.

Toutefois, il faut noter que R exclut les observations qui ne se trouvent pas dans les deux tables. C'est pourquoi il faut utiliser des options supplémentaires. Par exemples:

```
mydata <- merge(mydata1, mydata2, by=c("country","year"), all=TRUE)
```
avec l'option all=TRUE, R garde toutes les observations dans la table finale.

Si l'on souhaite garder les observations venant (uniquement) de la table mydata1, on fait:

```
mydata <- merge(mydata1, mydata2, by=c("country","year"), all.mydata1=TRUE)
```
De même pour la table mydata2

mydata <- merge(mydata1, mydata2, by=c("country","year"), all.mydata2=TRUE)

NB: On peut aussi utiliser la fonction cbind() à la place du merge() dans ce cas, il y doit y avoir une correspondance exact entre les deux tables.

```
myddata <- cbind(mydata1, mydata2)
```

2.11.2. Fusionner ayant les mêmes variables mais les observations différentes

Pour fusionner deux tables ayant des observations différentes mais les mêmes variables, on utilise la fonction rbind(). Exemple :

```
myddata <- rbind(mydata1, mydata2)
```

2.12. Reformater une table de données : reshape wide et long

Pour faire du reshape wide et long sur une table, on peut utiliser la package reshape2

```
install.packages("reshape2")
library(reshape2)
```
Exemple d'application:

Soit la table suivante:

```
pop <- data.frame(ville = c("Paris", "Paris", "Lyon", "Lyon"),
                  arrondissement = c(1, 2, 1, 2),
                  pop_municipale = c(17443, 22927, 28932, 30575),
                  pop_totale = c(17620, 23102, 29874, 31131))
```
Cette table se présente sous format wide.

2.12.1. Reshape long : la fonction melt()

Pour passer d'un tableau large à un tableau long, on peut utiliser la fonction melt() en indiquant à travers le paramètre id.vars le vecteur contenant le nom des variables à conserver. Le paramètre value.name permet de changer le nom de la colonne du data frame retourné dans laquelle les valeurs sont empilées. Le paramètre variable.name permet de renommer la colonne du data frame retourné contenant les modalités.

```
pop_long <- melt(pop, id.vars=c("ville", "arrondissement"),
                 value.name = "population",
                 variable.name="type_population"
```

2.12.2. Reshape wide : la fonction dcast()

On va convertir le tableau pop_long précédent au format wide

Pour effectuer l'opération inverse, on peut utiliser la fonction dcast() du package reshape2. Le paramètre formula doit contenir un objet de classe formula (de type x ~ y), qui précise les colonnes que l'on souhaite conserver en fonction de celles qui contiennent les noms des nouvelles colonnes dans lesquelles nous voulons placer les mesures. Le paramètre value.var permet justement d'indiquer le nom de la colonne dans laquelle se trouvent les valeurs.

```
dcast(pop_long, formula = ville + arrondissement ~
type_population,
      value.var="population")
```

2.13. Scinder un data frame selon les modalités d'une variable

Pour scinder une table de données selon les modalité d'une variable (généralement catégorielle), on utilise la fonction split(). Exemple :

```
decoupeData=split(mydata,mydata$sexe) #découpe mydata selon
les valeurs de la variable sexe.
hommeData=decoupeData$H # data-frame pour les hommes
femmeData=decoupeData$F # data-frame pour les femmes
```

2.14. Calculer la somme ou la moyenne sur une ligne ou une colonne: les fonctions colSums(), rowSums(), colMeans() et rowMeans()

Pour calculer la somme par ligne ou par colonne ou la moyenne, on utilise les fonctions suivantes: colSums, rowSums, colMeans rowMeans.

Exemple: soit la table de données définie comme suit:

```
region = rep(c(rep("Bretagne", 4), rep("Corse", 2)),2)
departement = rep(c("Cotes-d'Armor", "Finistere","Ille-et-Vilaine", "Morbihan","Corse-du-Sud", "Haute-Corse"),2)
annee = rep(c(2011, 2010), each = 6)
ouvriers = c(8738, 12701, 11390, 10228, 975, 1297, 8113, 12258, 10897, 9617, 936, 1220)
ingenieurs = c(1420, 2530, 3986, 2025, 259, 254, 1334, 2401, 3776, 1979, 253, 241)
myData <- data.frame(region =region ,departement=departement,annee=annee,
ouvriers=ouvriers,ingenieurs=ingenieurs)
```

Cette table donne le nombre d'ouvriers et d'ingénieurs par année et par département.

On veut calculer la somme de chaque variable (ouvrier et ingénieurs). On souhaite également calculer la moyenne sur ouvriers et ingénieurs (individuellement) et par lignes. Alors on fait:

```
colSums(myData[, c("ouvriers", "ingenieurs")], na.rm = TRUE, dims = 1) # Calcule la somme par colonne pour chaque variable
rowSums(myData[, c("ouvriers", "ingenieurs")], na.rm = TRUE, dim=1) # Calcule la somme par ligne (ouvrier+ingénieur)
colMeans(myData[, c("ouvriers", "ingenieurs")], na.rm = TRUE, dims = 1) # Calcule la moyenne par colone pour chaque variable
rowMeans(myData[, c("ouvriers", "ingenieurs")], na.rm = TRUE, dim=1) # Calcule la moyenne par ligne ( moyenne ouvrier et ingénieur)
On pouvait aussi utiliser la fonction apply() en spécifiant les dimensions (1: pour ligne et 2 pour colonne). Exemple :
apply(myData[, c("ouvriers", "ingenieurs")],2,mean) # Calcule la moyenne par colonne pour chaque variable
apply(myData[, c("ouvriers", "ingenieurs")],1,mean) ## Calcule la moyenne par ligne ( moyenne ouvrier et ingénieur)
```

NB: Il arrive souvent qu'on ait besoin de joindre les moyennes et sommes calculées à la table initiale. Il faut alors penser à convertir ces résultats en data frame et les joindre en utilisant merge() ou rbind() ou cbind().

Mais très généralement, les moyennes et les sommes sont calculées par sous-groupes. D'abord, pour effectuer ces calculs par sous-groupes, il faut utiliser la fonction aggregate() présentée plus bas. Avec cette fonction, les statistiques se présentent directement sous forme de data frame. Le merging devient donc très simple.

2.15. Calcul de valeurs par sous-groupe et agrégation de valeurs: la fonction aggregate()

La fonction aggregate() permet d'élaborer des statistiques en les présentant par groupe ou par sous-groupes..

Ex: soit la table de données définie comme suit:

```
region = rep(c(rep("Bretagne", 4), rep("Corse", 2)),2)
departement = rep(c("Cotes-d'Armor", "Finistere","Ille-et-Vilaine", "Morbihan","Corse-du-Sud", "Haute-Corse"),2)
annee = rep(c(2011, 2010), each = 6)
ouvriers = c(8738, 12701, 11390, 10228, 975, 1297, 8113, 12258, 10897, 9617, 936, 1220)
ingenieurs = c(1420, 2530, 3986, 2025, 259, 254, 1334, 2401, 3776, 1979, 253, 241)
myData <- data.frame(region =region ,departement=departement,annee=annee, ouvriers=ouvriers, ingenieurs=ingenieurs)
```

Cette table donne le nombre d'ouvriers et d'ingénieurs par année et par département. On souhaite faire plusieurs agrégations comme par exemple: -calculer le nombre total d'ingénieurs par région, etc... Voici-ci dessous quelques applications.

1- Calcul de la somme totale des ouvriers et des ingénieurs par année:

```
aggregate(myData[, c("ouvriers", "ingenieurs")],by = list(annee = myData$annee), FUN = sum)
```

Nb: la statistique produite est un data frame. On peut le joindre à la table initiale en faisant merge(). Exemple:

```
x<-aggregate(myData[, c("ouvriers", "ingenieurs")],by = list(annee = myData$annee), FUN = sum)
testdata <- merge(myData, x, by=c("annee"))
```

2- Somme totale d'ouvriers et d'ingénieurs par année et par région

```
aggregate(myData[, c("ouvriers", "ingenieurs")],by = list(region = myData$region, annee = myData$annee),FUN = sum)
```

Astuce: Pour calculer la moyenne générale ou la somme sur tout l'échantillon avec la fonction aggregate(), on peut créer une variable qui prend une seule valeur,ensuite spécifier cette valeur dans l'argument by=.

2.16. Centrer et réduire les variables d'une table: la fonction scale()

Le centrage consiste à retrancher à chacune des colonnes d'une table la moyenne de cette colonne. La réduction consiste à diviser chacune des colonnes par son écart type. La fonction scale() permet de centrer et/ou de réduire une matrice.

Exemple:

```
scale(mydata,scale=FALSE) # Centre uniquement les colonne de mydata
scale(mydata,center=FALSE,scale=apply(mydata,2,sd)) # Centre et réduit les colonnes de mydata
```

Dans cet exemple, on suppose que toutes les variables de mydata sont quantitatives. Si ce n'est pas le cas, il faut sélectionner les variables à traiter par subsetting().

Chapitre 3 : Les analyses statistiques classiques sous R

Ce chapitre présente la mise en application des analyses statistiques classiques classiques sous R. Nous présentons notamment les analyses statistiques univariées, les statistiques descriptives bivariées, les analyses d'association entre variables, les tests de comparaison de moyennes. Au-delà des analyses descriptives, nous présentons également les régressions linéaires, les régressions logistiques mais aussi les analyses multi-dimensionnelles (analyses en composantes principales et analyses factorielles).

3.1. Statistiques descriptives

3.1.1. Statistiques descriptives sur variables qualitatives nominales et ordinales

3.1.1.1. Tableaux de fréquences absolues et relatives (tri à plat)

Exemple: Répartition des classes d'âge des patients

```
attach(mydata)
table(classe_age) # Fréquences absolues
table(classe_age)/length(classe_age)  # Fréquences relatives (proportions)
100*table(classe_age)/length(classe_age)     # Fréquences relatives (pourcentages)
nlevels(classe_age)     # renvoie le nombre de modalités(classe_age déclarée comme factor)
detach(mydata)
```

Pour fixer le nombre de décimal, utiliser la fonction round(). Exemple:

```
round(table(classe_age)/length(classe_age),3) # fixe le nombre de décimales à 3
```

3.1.1.2. Tableaux de contingence (tri croisé)

Exemple: Répartition des classes d'âge par sexe

```
attach(mydata)
table(classe_age,sexe) # Fréquences absolues
100*table(classe_age,sexe)/sum(table(classe_age,sexe))        # répartition en % (sur l'échantillon total)
detach(mydata)
```

Ajout des margins au tableau de contingence (somme par ligne et par colonne)

```
mytable<-table(classe_age,sexe)  # Construit le tableau de contingence
```

```
mytable2<-addmargins(mytable,FUN=sum) # Ajoute les margins au tableau de contingence
mytable2
tableaufreq <- 100*table(classe_age,sexe)/sum(table(classe_age,sexe)) # Calcule les pourcentages(par rapport à l'échantillon total)
tableaufreq2 <-addmargins(tableaufreq,FUN=sum)# calcule les margins du tableau de fréquence relative
tableaufreq2
```

Accéder aux éléments d'un tableau de contingence

Le tableau de contingence a les mêmes propriétés qu'une matrice, on peut donc accéder aux éléments en indiquant leurs indices

Exemples:

```
mytable[1,1] # élément de la première ligne et première colonne
mytable[3, 2] # élément de la troisième ligne et deuxième colonne
mytable[1,] # Tous les éléments de la première ligne
mytable[, 2 ] # Tous les éléments de la deuxième colonne
```

3.1.2. Statistiques descriptives sur variables quantitatives : caractéristiques de tendance centrale et de dispersion

Les caractéristiques de tendance centrale incluent la moyenne, le mode ou la médiane, etc…

Les caractéristiques de dispersion incluent la variance (l'écart-type), le min, le max, l'étendue, l'écart interquartile, etc…

Ces caractéristiques sont calculables uniquement sur les variables quantitatives (continues ou discrètes)

Sélection des variables quantitatives de la table patientdata

```
quantivars<-c("age", "poids", "taille", "imc", "glucose","diastolic", "serum", "douleurm0","douleurm1")
```

Si toutes les variables numériques de la table étaient des quantitatives:

```
quantivar<-which(sapply(mydata, is.numeric))
```

Attention: Toute variable quantitative est numérique mais une variable numérique n'est pas nécessairement quantitative. Exemple une variable qualitative codée par des valeurs numériques.

Utilisation de la fonction summary()

```
quantivars<-c("age", "poids", "taille", "imc", "glucose","diastolic", "serum", "douleurm0","douleurm1")
summary(mydata[quantivars]) # sommaire sur les vars quantitatives
```

Utilisation de la fonction stat.desc() du package pastecs

```
install.packages("pastecs")
library(pastecs)
stat.desc(mydata[quantivars], basic = FALSE)
```

NB : Utiliser la fonction round() pour contrôler le nombre de décimales.

3.2. Matrice de corrélation: la fonction cor()

```
quantivars<-c("age", "poids", "taille", "imc", "glucose","diastolic", "serum", "douleurm0","douleurm1"))
```

Matrice de corrélation complète

```
cor(mydata[quantivars])
round(cor(mydata[quantivars]),3)
```

Afficher la matrice sous forme triangulaire

```
corm<-round(cor(mydata[quantivars]),3) # produit la matrice complète
corm[upper.tri(corm)]<-"" # Supprime les valeurs de la partie supérieure de la matrice complète
corm<-as.data.frame(corm)
print(corm)
```

Tester la significativité des corrélations*: la fonction rcorr() du package Hmisc*

```
library(Hmisc)
rcorr(as.matrix(mydata[quantivars]), type="pearson") # Matrice de corrélation avec les p-values.
```

3.3. Calculer un quantile d'une variable : la fonction quantile()

Exemple

Soit la variable age de la table de données mydata. On peut calculer quelques quantiles comme suit :

```
attach(mydata)
quantile(AGE,probs=c(0.1,0.9)) # Fractiles d'ordre 10 % et 90 %
quantile(AGE,probs=c(0.25,0.5,0.75)) # Calcul les quartiles
quantile(AGE,probs=c(0.25,0.5,0.75)) # Calcul les quartiles
```

```
quantile(x,probs=1:10/10)  # Calcul les déciles
detach(mydata)
```

3.4. Tests statistiques usuels

3.4.1. Test d'indépendance du Khi-deux : la fonction chisq.test()

Le test d'indépendance du khi-2 se réalise à partir d'un tableau de contingence.

Exemple: Test entre les variables « imc_cat » et « maladie_chronique » de la table patientdata

```
chisq.test(table(mydata$imc_cat, mydata$maladie_chronique))
```

Test de khi-2 avec correction de Yates: à appliquer dans le cas d'un tableau 2X2

Exemple: Test entre les variables « diabete » et « sport » de la table patientdata

```
chisq.test(table(mydata$diabete, mydata$sport), correct=TRUE)
```

Récupération des statistiques post-test

```
testtab<-chisq.test(table(mydata$imc_cat,
mydata$maladie_chronique)))
testtab$expected # Renvoie le tableau de fréquence théorique
testtab$residuals^2 # renvoie les résidus au carré ( les
écarts au carré)
testtab$statistic # renvoie les statistiques du test
```

Test exact de Fisher: à appliquer lorsque les fréquences absolues dans certaines cellules du tableau de contingence sont inférieures à 5.

Exemple: Test entre les variables « imc_cat » et « maladie_chronique » de la table patientdata

```
fisher.test(table(mydata$imc_cat,    mydata$maladie_chronique),
alternative = "greater")
```

3.4.2. Test de Student

3.4.2.1. Test d'égalité de la moyenne à une valeur de référence

Il s'agit ici de tester si la moyenne observée sur l'échantillon est bien égale à une valeur de référence. Ce test est également appelé test de conformité.

Exemple: Testons si la pression diastolique des patients de l'échantillon est en moyenne égale à 80 (théorique normale)

```
t.test(mydata$diastolic,mu=80,alternative = "two.sided")# Test
bilatéral
t.test(mydata$diastolic,mu=80,alternative="less")#         Test
unilatéral à gauche
```

```
t.test(mydata$diastolic,mu=80,alternative="greater")#      Test
unilatéral à droit
```

3.4.2.2. Test comparaison de moyennes entre deux échantillons indépendants

Il s'agit de tester la différence de moyennes entre deux groupes distincts

Exemple: Testons la différence de moyenne de la pression diastolique entre les patients diabétiques et les patients non-diabétiques

```
#Présentation des moyennes par groupe
aggregate(mydata[,   c("diastolic")],by   =   list(diabeteG   =
mydata$diabete), FUN = mean)
#Création des variables de test
diastolicG1 <- subset(mydata, diabete==1, select=c(diastolic))
# Les diabétiques
diastolicG2 <- subset(mydata, diabete==0, select=c(diastolic))
# Les non-diabétiques
#Réalisation du test
t.test(diastolicG1,diastolicG2,var.equal=TRUE)
```

3.4.2.3. Test comparaison de moyennes de deux échantillons appariés

Il s'agit de tester la différence de moyennes sur le même groupe mais dans deux situations différentes (ou à deux dates différentes)

Exemple: Tester le niveau de douleur ressenti par les patients avant et après la prise d'un traitement.

Prenons ici: les variables douleurm0 et douleurm1 de la table de données patientdata

```
#Présentation des moyennes
mean(mydata$douleurm0)
mean(mydata$douleurm1)
Réalisation du test
t.test(mydata$douleurm0,       mydata$douleurm1,var.equal=TRUE,
PAIRED=TRUE)
```

3.4.3. Test de comparaison de moyennes sur plusieurs échantillons : ANOVA

C'est une généralisation du test de Student à plusieurs groupes. Il s'agit de tester la différence de moyennes sur plusieurs groupes (fondée sur l'hypothèse de normalité de distribution)

Exemple: Tester la moyenne de pression diastolique selon les différentes catégories d'imc (définies par la variable « imc_cat » du patiendata)

```
#Présentation des moyennes par groupe
```

```
aggregate(mydata[, c("diastolic")],by = list(imcGroup =
mydata$imc_cat), FUN = mean)
Réalisation du test
stat_anova <- aov(diastolic~ imc_cat, data=mydata)
summary(stat_anova)
```

3.4.4. Test de comparaison de proportions entre deux échantillons

Exemple: Test d'égalité de la proportion de diabétiques chez les hommes comparés aux femmes

```
#Réalisation du test
prop.test(table(mydata$sexe,mydata$diabete))
```

3.5. Modèles de régressions linéaires : les moindres carrés ordinaires MCO (OLS)

3.5.1. But des modèles de régressions linéaires

Evaluer l'influence d'une ou de plusieurs variables indépendantes (quantitatives ou qualitatives) sur une variable dépendante (quantitative continue ou discrète).

Les variables « indépendantes » sont aussi appelées « variables explicatives »

La variable « dépendante » est aussi appelée « variable expliquée »

Exemple: Mesurer l'influence des caractéristiques biologiques et les antécédents cliniques des patients sur le niveau de la pression diastolique.

Pour élaborer un tel modèle, distinguons deux cas: l'un où toutes les variables explicatives sont des variables quantitatives (ex: âge, poids, taille, etc..) et l'autre où les variables explicatives incluent à la fois des variables quantitatives et des variables qualitatives (comme le sexe, pratique du sport, fumeur, etc…).

3.5.2. Estimation du modèle dans le cas où toutes les variables explicatives sont quantitatives

```
myreg1 <- lm(diastolic ~  age + poids +glucose +taille , data
= mydata)
summary(myreg1)  # Table de régression
confint(myreg1) # Intervalle de confiance des paramètres
Pour avoir la liste de tous attributs de cette régression, on
fait:
attributes(myreg1)
```

3.5.3. Estimation du modèle dans le cas où les variables explicatives contiennent des variables qualitatives

Lorsque les variables explicatives contiennent des variables catégorielles (ex: sexe, classe_age, imc_cat, tabac, sport), celles-ci doivent être d'abord transformées en variables factors avant d'être intégrées au modèle.

Pour transformer les variables qualitatives en factors on utilise la fonction as.factor() comme suit:

```
mydata$sexe<-as.factor(mydata$sexe)
mydata$classe_age<-as.factor(mydata$classe_age)
mydata$imc_cat<-as.factor(mydata$imc_cat)
mydata$tabac<-as.factor(mydata$tabac)
mydata$sport<-as.factor(mydata$sport)
#Estimation après transformation
myreg2 <- lm(diastolic ~  sexe+ classe_age+ poids +glucose
+taille+imc_cat+tabac+sport , data = mydata)
summary(myreg2)
```

NB: Il faut noter que R choisit par défaut la première modalité de la variable catégorielle comme modalité de référence. On peut fixer la modalité de référence en utilisant la fonction relevel(). Ex:

```
mydata$classe_age <- relevel(mydata$classe_age, ref = "35-45
ans") # les 35-45 ans pris comme reférence.
On peut alors estimer de nouveau le modèle après cette
redéfinition.
myreg2 <- lm(diastolic ~  sexe+ classe_age+ poids +glucose
+taille +imc_cat + tabac+sport , data = mydata)
summary(myreg2)
```

3.5.4. Insérer un terme quadratique dans le modèle: la fonction I()

On soupçonne par exemple une relation non linéaire entre l'âge et la pression diastolique. Pour cela, on intègre au modèle un terme quadratique pour capter cette non linéarité. Le terme quadratique correspond à l'âge au carré. Ainsi, on a:

```
myreg3 <- lm(diastolic ~  sexe+ age+I(age^2)+ poids +glucose
+taille+ imc_cat+tabac+sport , data = mydata)
summary(myreg3)
```

Il est également possible de prendre en compte les effets d'interaction entre les variables. Par exemple l'effet d'interaction entre le niveau de glucose et le tabac. On a:

```
myreg3b <- lm(diastolic ~  sexe+ age+I(age^2)+ poids +glucose+
tabac+ I(glucose*tabac)+taille+imc_cat +sport , data = mydata)
summary(myreg3b)
```

3.5.5. Sélection automatique des variables explicatives: la fonction step()

La fonction step() permet de sélectionner les variables explicatives pertinentes (sur la base de leur significativité) pour estimer le modèle. L'utilisation de cette fonction est surtout utile lorsque le nombre de variables explicatives est très élevé.

On distingue deux modes de sélection des variables: la « forward selection » et la « backward selection ».

3.5.5.1. Forward selection

```
step(lm(diastolic ~1 , data = mydata),diastolic ~  sexe+ age+ poids +glucose +taille+ mal_respire+tabac +sport+emploi+depression,direction="forward")
```

Le modèle optimal est :lm(formula = diastolic ~ glucose + taille + age, data = mydata)

```
selectedmodel1<-lm(formula = diastolic ~ glucose + taille + age, data = mydata)
summary(selectedmodel1)
```

3.5.5.2. Backward selection

```
step(lm(diastolic ~  sexe+ age+ poids +glucose +taille+ mal_respir+tabac+sport+emploi+depression , data = mydata),direction="backward")
```

Le modèle optimal est :lm(formula = diastolic ~ glucose + taille + age, data = mydata)

```
selectedmodel2<-lm(formula = diastolic ~ glucose + taille + age, data = mydata)
summary(selectedmodel2)
```

3.5.6. Test de contraintes linéaires

Soit le modèle linéaire suivant:

```
myreg4 <- lm(diastolic ~  sexe+ age+ poids +glucose +taille+ tabac+sport , data = mydata)
summary(myreg4)
```

On va tester quelques hypothèses linéaires (formulées sous forme de propositions. Pour cela, on va utiliser la fonction linearHypothesis() du package car

Mise en œuvre des tests

```
install.packages ("car")
library(car)
```

P1: L'âge n'a aucune influence sur la pression diastolique.

```
linearHypothesis(myreg4, "age=0") # hypothèse rejetée
```

P2: L'effet total du poids et de la taille sur la pression diastolique vaut 1.5

```
linearHypothesis(myreg4, "poids+taille=1.5")
```

P3: Est-ce l'âge et le poids ont le même effet sur la pression diastolique ?

```
linearHypothesis(myreg4, "age=poids")
```

P4: Est-ce le tabac et le sport ont des effets strictement contraires sur la pression diastolique ?

```
linearHypothesis(myreg4, "tabac1= -sport1")
```

3.5.7. Diagnostic des résidus

3.5.7.1. Rappel des hypothèses de base du modèle linéaire

Spécification du modèle

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + \varepsilon$$

Les hypothèses qui conditionnent la validité de l'estimation par les MCO sont:

H1: Les erreurs sont de moyenne nulle

H2: La variance des erreurs constante

H3: La covariance des erreurs nulle (absence de corrélation des erreurs)

H4: Absence de corrélation entre les erreurs et les variables explicatives

H5: Les variables explicatives non aléatoires (mesurées sans erreur)

Lorsque l'une de ces hypothèses n'est pas vérifiée, il faut prendre des dispositions méthodologiques pour apporter des corrections.

3.5.7.2. Diagnostic sur la normalité des résidus

Soit le modèle:

```
myreg5 <- lm(diastolic ~  sexe+ age+ poids +glucose +taille+
tabac+sport , data = mydata)
summary(myreg5)
Graphiques post-régression:
graphics.off()
x.11()
par(mfrow = c(2,2))
plot(myreg5)  # Trace 4 graphiques permettant d'analyser le
résultat de la régression
```

3.5.7.3. Histogramme des résidus

```
resid<-myreg5$residuals #  Récupère les résidus de régression
m<-mean(resid, na.rm = TRUE) # Calcule la moyenne des résidus
std<-sqrt(var(resid, na.rm = TRUE)) # Calcule l'écart-type des
résidus
graphics.off();x.11()
```

```
hist(resid,breaks=seq(min(resid),max(resid),by=(
range(resid)[2]-range(resid)[1])/50), freq = FALSE) # trace
l'histogramme
curve(dnorm(x, mean=m, sd=std), col="darkblue", lwd=2,
add=TRUE, yaxt="n") # ajoute le courbe de la loi normale.
```

3.5.7.4. Test de normalité

```
install.packages("tseries")
library("tseries")
# Test de Jarque-Bera
jarque.bera.test(resid) # Test de non-normalité de Jareque et
Bera.
# Test de Kolmogorov-Smirnov)
ks.test(resid, y="pnorm",alternative = "two.sided", exact =
NULL)
#Test de shapiro-wilk
shapiro.test(resid)# Shapiro-Wilk Normality Test (stats)
```

3.5.7.5. Test d'autocorrélation de Durbin-Watson

```
install.packages("lmtest")
library(lmtest)
dwtest(myreg5)
```

3.5.7.6. Non constance de la variance des erreurs (hétérocédasticité)

Test d'hétéroscédasticité de Breush Pagan: le test classique

```
library(lmtest)
bptest(myreg5)
```

Test d'hétéroscédasticité de Breush Pagan: Non-constant Variance Score Test

```
library(car)
ncvTest(myreg5)
```

3.5.7.7. Correction de la corrélation des erreurs et de l'hétérocédasticité par les moindres carrés généralisés:

```
#install.packages("nlme")
library(nlme)
myglsreg <- gls(diastolic ~ age+ poids +glucose +taille ,
data = mydata, correlation = corAR1(value=0.3, form = ~ 1 ),
weights = varPower() )
summary(myglsreg)
```

3.5.7.8. Test de multicolinéarité entre les variables explicatives : les Variance Inflation Factors(VIF)

```
library(car)
vif(myreg5) # affiche les VIFs
```

3.5.8. Prévision à partir du modèle estimé

On peut utiliser les coefficients estimés pour faire une prédiction de la variable dépendante sur un nouvel échantillon. Par exemple, pour prédire la pression diastolique de nouveaux patients dont on connait déjà les caractéristiques biologiques et antécédents cliniques mais pour lesquels la pression diastolique n'a pas encore relevée.

Pour cela, on se sert la fonction predict() et du modèle final estimé.

Exemple:

Soit le modèle final suivant:

```
finalreg <- lm(diastolic ~  sexe+ age+ poids +glucose +taille+
tabac+sport , data = mydata)
summary(finalreg)
```

Soit un échantillon fictif constitué des dix dernières observations de l'échantillon patientdata:

```
newsample <- mydata[1:10,]
```

Prévision sur le nouvel échantillon

```
predval<-predict(finalreg, newdata = newsample) # Renvoie les
valeurs prédites pour les 10 observations
predval # vecteur des pressions diastoliques prévues
```

3.6. Moindres carrés non-linéaires(NLS)

3.6.1. But des moindres carrés non-linéaires

Les estimations NLS sont utilisées lorsque la variable dépendante ne peut pas être exprimée comme une combinaison linéaire (par rapport aux coefficients) des variables explicatives.

En guise de rappel le modèle linéaire se présente de manière générale comme suit:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + \varepsilon$$

Les variables explicatives sont liées linéairement à la variable dépendante.

Mais dans le cas des modèles non-linéaires, cette condition n'est pas vérifiée pour au moins un paramètre.

Ci-dessous deux exemples de modèles non-linéaires par rapport aux coefficients:

$$y = \beta_0 e^{\beta_1 x_1 + \beta_2 x_2}$$

$$y = \beta_0 {x_1}^{\beta_1} + \log(\beta_2 x_2)$$

3.6.2. Estimation des modèles non-linéaires

Pour l'estimation d'un modèle non-linéaire, deux cas se présentent:

- Soit le modèle est « linéarisable » auquel cas on utilise un changement de variable et estimer ensuite un modèle linéaire par MCO. Le plus souvent, on procède par transformation logarithmique.
- Soit le modèle est « non-linéarisable », auquel cas on utilise les moindres carrés non-linéaires.

Dans les deux exemples précédents, le premier modèle est linéarisable par transformation logarithmique telle que:

$$\log(y) = \widetilde{\beta_1} + \beta_1 x_1 + \beta_2 x_2$$

Avec $\widetilde{\beta_1} = \log(\beta_0)$.

Par contre le second modèle n'est pas linéarisable par transformation logarithmique.

3.6.2.1. Cas pratique de modèles non-linéaires: estimation du modèle de BASS

Conçu par Frank Bass en 1969 , le modèle de BASS est un modèle servant à étudier la diffusion d'une innovation technologique et à prédire son taux d'adoption de l'innovation par la population au cours du temps. Ex : adoption de la technologie de la fibre optique, la 4G, une nouvelle marque de smart-phone, etc....

Exemple

Soient les données suivantes représentant le nombre d'adopteurs d'une technologie au cours du temps après son premier lancement :

```
nadopt <- c(840, 1470, 2110, 4000, 7590, 10950, 10530, 9470,
7790, 5890,1470, 4000, 7590, 10532, 9475, 5890, 8540)
```

On veut élaborer un modèle capable de prédire le nombre d'adoptions de cette technologie dans le futur en partant du modèle de BASS qui se présente comme suit:

$$n(t) = m * \left[\frac{(p+q)^2 e^{-(p+q)t}}{p\left[1 + \frac{q}{p} e^{-(p+q)t}\right]^2} \right]$$

Où m, p et q sont les paramètres à estimer; m est la taille du marché et n(t) le nombre d'adopteurs à la date t.

Les étapes de l'estimation sont présentées comme suit:

```
# Préparation des données
```

```
nadopt <- c(840, 1470, 2110, 4000, 7590, 10950, 10530, 9470, 7790, 5890,1470, 4000, 7590, 10532, 9475, 5890, 8540) # Vecteur des données
t <- seq(1:length(nadopt)) # définit la variable temps
# Estimation du modèle
BassModel <- nls(nadopt ~ M * (((P + Q)^2/P) * exp(-(P + Q) * t))/(1 + (Q/P) *exp(-(P + Q) * t))^2, start = list(M = sum(nadopt)*2, P = 0.05, Q = 0.5))
summary(BassModel)
# Récupération des coefficients estimés:
Bcoef <- coef(BassModel)
m <- Bcoef[1]
p <- Bcoef[2]
q <- Bcoef[3]
Représentation de la fonction de densité et de la fonction de répartition du nombre d'adopteurs:
Tdelt <- seq(from=1, to= length(nadopt), by=0.1) # on génère des points de temps entre 1 et 16 (la dimension de t)
# Fonction de densité
Bpdf <- m * ((p + q)^2/p) * exp(-(p + q) * Tdelt)/(1 + (q/p) * exp(-(p + q) * Tdelt))^2
# Fonction de répartition
Bcdf <- m * (1 - exp(-(p + q) * Tdelt))/(1 + (q/p) * exp(-(p + q) * Tdelt))
# Représentation
graphics.off(); x11();par(mfrow=c(1,2))
plot(Tdelt, Bpdf, xlab = "Temps", ylab = "Nombre d'adoptions", type = "l")
plot(Tdelt, Bcdf, xlab = "Temps", ylab = "Nombre cumulé d'adoptions", type = "l", col="red")
lines(t, cumsum(nadopt),)
points(t, cumsum(nadopt),)
```

3.6.2.2. Prévision à partir du modèle de BASS

```
# Prévision du nombre d'adopteurs sur un horizon de 5 ans:
Tdelt_prev <- seq(from=length(nadopt), to= (length(nadopt)+5), by=0.1)
Bpdf_prev <- m * ((p + q)^2/p) * exp(-(p + q) * Tdelt_prev)/(1 + (q/p) * exp(-(p + q) * Tdelt_prev))^2 # fonction de densité sur l'horizon prévu
graphics.off() ;x11()
plot(Tdelt, Bpdf, xlim=c(min(Tdelt), max(Tdelt_prev)),ylim=c(min(rbind(Bpdf,Bpdf_prev)), max(rbind(Bpdf,Bpdf_prev))), xlab = "Temps", ylab = "Nombre d'adoptions", type = "l") # tracé de la courbe initiale Bpdf
```

```
lines(Tdelt_prev,Bpdf_prev, col="red") # ajout de la courbe de prévision
# Prévision du cumul d'adopteurs sur un horizon de 5 ans:
Tdelt_prev <- seq(from=length(nadopt), to= (length(nadopt)+5), by=0.1)
Bcdf_prev <- m * (1 - exp(-(p + q) * Tdelt_prev))/(1 + (q/p) * exp(-(p + q) * Tdelt_prev))
graphics.off() ; x11()
plot(Tdelt, Bcdf,xlim=c(min(Tdelt), max(Tdelt_prev)),ylim=c(min(rbind(Bcdf,Bcdf_prev)), max(rbind(Bcdf,Bcdf_prev))), xlab = "Temps", ylab = "Nombre cumulé d'adoptions", type = "l")# tracé de la courbe initiale Bcdf
lines(Tdelt_prev,Bcdf_prev, col="red")
```

3.7. Le modèle de régression logistique binaire (logit)

3.7.1. But du modèle logit

Dans le modèle de régression linéaire, on mesure l'influence des variables explicatives (quantitatives et catégorielles) sur une variable quantitative. Par exemple, expliquer la pression diastolique des patients par leurs caractéristiques biologiques et leurs antécédents cliniques. Dans la régression logistique binaire, il s'agira d'expliquer une variable catégorielle binaire par un ensemble de variables explicatives (qui peuvent être à la fois quantitatives et catégorielles). Ici, on raisonne en termes de probabilité.

Par exemple, on va expliquer la probabilité de survenue du diabète chez un patient en fonction de ses caractéristiques biologiques et de ses antécédents cliniques.

Pour cela, on va choisir les mêmes variables explicatives que pour le cas du modèle linéaire.

3.7.2. Estimation du modèle logit

La fonction de base pour estimer un modèle logit est glm(). Mais cette fonction peut-être complétée par d'autres fonctions provenant d'autres packages notamment pour le calcul d'odd-ratio ou d'effets marginaux.

Exemple :

```
mylogit1 <- glm(diabete ~ sexe+ age+ poids +glucose +depression+imc+ tabac+sport, family=binomial(link="logit"), data=mydata)
summary(mylogit1) # Tableau de régression
```

Calcul des odd-ratios

```
cbind(coefficients<-round(coef(mylogit1),4),          OddRatio<-
round(exp(coef(mylogit1)),4)) # calcule les odd-ratios et les
joint aux coefficients initiaux
On pouvait obtenir les odd-ratios et les effets marginaux en
utilisant le package mfx comme suit:
library(mfx)
mylogit2 <- logitor(diabete ~   sexe+ age+ poids +glucose
+depression+imc+ tabac+sport, data=mydata)
mylogit2
```

Calcul des effets marginaux

Les odd-ratios mesurent les rapports des probabilités alors que les effets marginaux mesurent les variations de probabilités.

```
library(mfx)
mylogit3 <- logitmfx(diabete ~   sexe+ age+ poids +glucose
+depression+imc+ tabac+sport, data=mydata)
mylogit3
```

3.7.3. Prédiction de probabilités

Après avoir estimé les paramètres du modèle logit, on peut prédire les probabilités de survenu de diabète sur des patients provenant d'un nouveau échantillon.

Exemple:

```
newsample <- mydata[1:10,]  # échantillon fictif constitué des
10 premières observations de la table mydata
predprob    <-    predict(mylogit1,    newdata=    newsample,
type="response") # Prédiction de probabilité
predprob # le vecteur des probabilités prédites.
```

3.8. Modèle de régression probit

3.8.1. Estimation

```
myprobit    <-    glm(admit    ~    gre    +    gpa    +    rank,
family=binomial(link="probit"), data=mydata)
summary(myprobit)
```

3.8.2. Prédiction probabilités

```
probpred<-    predict(myprobit,    newdata=mydata,    type    =
"response", se.fit=FALSE)[-3]
probpred
```

3.9. Analyse en composantes principales : la fonction princomp()

Exemple : soit mydata la table de données formées uniquement de variables quantitatives. On peut faire l'ACP sur cette table comme suit :

```
mypca <- princomp(mydata, cor = T, scores = T) #  l'option scores = T dit à  R de calculer les scores sur les composantes
print(mypca)
summary(mypca)
```

La fonction princomp fournit les écarts-types associés aux axes. Le carré correspond aux variances = valeur propres. Elle fournit également le pourcentage cumulé.

```
attributes(mypca)
compos<-mypca$scores[,1:3]  # Retient les trois premières composantes
print(compos)
```

Pour obtenir les variances associées aux axes c.-à-d. les valeurs propres, on fait :

```
val.propres <- acp.autos$sdev^2
print(val.propres)
```

Réaliser le scree plot (graphique des valeurs propres)

```
plot(1:6,val.propres,type="b",ylab="Valeurspropres",xlab="Composante",main= "Scree plot")
```

Valeurs propres associés aux axes : Calcul, intervalles de confiance et Scree Plot

```
val.low <- val.propres * exp(-1.96 * sqrt(2.0/(n-1)))
val.high <- val.propres * exp(+1.96 * sqrt(2.0/(n-1)))
#Affichage sous forme de tableau
tableau <- cbind(val.low,val.propres,val.high)
colnames(tableau) <- c("B.Inf.","Val.","B.Sup")
print(tableau,digits=3)
```

Cercle des corrélations: Variables actives

```
c1 <- acp.autos$loadings[,1]*acp.autos$sdev[1]
c2 <- acp.autos$loadings[,2]*acp.autos$sdev[2]
correlation <- cbind(c1,c2)
print(correlation,digits=2)
```

Les carrés de la corrélation (cosinus²)

```
print(correlation^2,digits=2)
```

Cumul carrés de la corrélation

```
print(t(apply(correlation^2,1,cumsum)),digits=2)
```

Cercle des corrélations - variables actives

```
plot(c1,c2,xlim=c(-1,+1),ylim=c(-1,+1),type="n")
abline(h=0,v=0)
text(c1,c2,labels=colnames(autos.actifs),cex=0.5)
symbols(0,0,circles=1,inches=F,add=T)
```

Carte des individus sur les 2 premiers axes: Individus actifs

```
plot(acp.autos$scores[,1],acp.autos$scores[,2],type="n",xlab="
Comp.1 -
74%",ylab="Comp.2 - 14%")
abline(h=0,v=0)
text(acp.autos$scores[,1],acp.autos$scores[,2],labels=rownames
(autos.actifs),cex=0.75)
```

3.10. Analyse factorielle

Le module base offre la fonction factanal() pour faire les AFC.

Exemple:

```
fit <- factanal(mydata, 3, rotation="varimax")
print(fit, digits=2, cutoff=.3, sort=TRUE)
```

Représentation graphique facteur 1 et 2

```
load <- fit$loadings[,1:2]
plot(load,type="n")
text(load,labels=names(mydata),cex=.7) # add variable names
```

Chapitre 4 : Data visualisationn avec R

Ce chapitre présente les différentes méthodes de construction de graphiques sous R. Tout comme pour les méthodes d'analyses descriptives, le choix du type de visualisation graphique dépend de la nature des variables. Pour les graphiques univariés sur variables qualitatives, on choisira par exemple les diagrammes de fréquences (barres de fréquences) et les diagrammes circulaires (camemberts). Pour les graphiques univariés sur variables quantitatives, on choisira les histogrammes, les box-plots, les barres de moyennes, etc… Et pour les analyses croisées, entre deux variables quantitatives, par exemple on choisira un nuage de points ou une courbe d'évolution, etc.. Tandis que pour une analyse croisée entre une variable quantitative et une variable qualitative, on choisira les histogrammes, les box-plots (ou les barres de moyennes) présentés selon les différentes modalités de la variable qualitative. L'objet de ce chapitre est de présenter brièvement chacun de ces cas.

4.1. Préparation du cadre du graphique

4.1.1. Ouverture et gestion de fenêtres de graphiques

Les fenêtres graphiques sont nommées X11 sous Unix/Linux et windows sous Windows. Dans tous les cas, on peut ouvrir une nouvelle fenêtre graphique avec la commande x11(). Ci-dessous quelques exemples:

```
x11() # Ouvre une première fenêtre graphique
x11() # Ouvre une seconde fenêtre graphique (suite du premier
x11())
windows() # Ouvre une nouvelle fenêtre sous windows
(alternative à x11())
win.graph() # Ouvre une nouvelle fenêtre sous windows
(alternative à x11())
dev.new() # Ouvre une nouvelle après l'appel de x11()
dev.list() # Liste toutes les fenêtre graphiques ouvertes
dev.set(3) # active la fenêtre numéro 3
dev.cur() # Renvoie le numéro de la fenêtre active (1 pour la
console).
dev.off(3) # Ferme la fenêtre graphique numéro 3
graphics.off() # Ferme toutes les fenêtres graphiques ouvertes
```

4.1.2. Découpage de la fenêtre graphique en cadrans

Pour avoir une vue synthétique, on peut vouloir placer plusieurs graphiques sur la même zone de graphique dans des cadres séparés. Pour ce faire, la zone graphique doit d'abord être découpée en plusieurs cadrans. La fonction de base permettant ce découpage est la fonction par() avec l'option mfrow. On peut aussi utiliser la fonction layout().

4.1.2.1. Utilisation de la fonction par()

Exemples

```
graphics.off() # Ferme toutes les fenêtres graphiques préalablement ouvertes
x11() # Ouvre une nouvelle fenêtre graphique
par( mfrow=c(2,2)) # Découpe la zone en 4 cadrans ( fenêtre principale découpée en 2 lignes et 2 colonnes)
Dès lors en traçant successivement 4 graphiques, les cadrans seront successivement remplis (d'abord par ligne). Exemples:
plot(1:10) ; plot(11:20); plot(21:30);plot(31:40) # Trace successivement 4 graphiques arbitraires
```

4.1.2.2. Utilisation de la fonction layout()

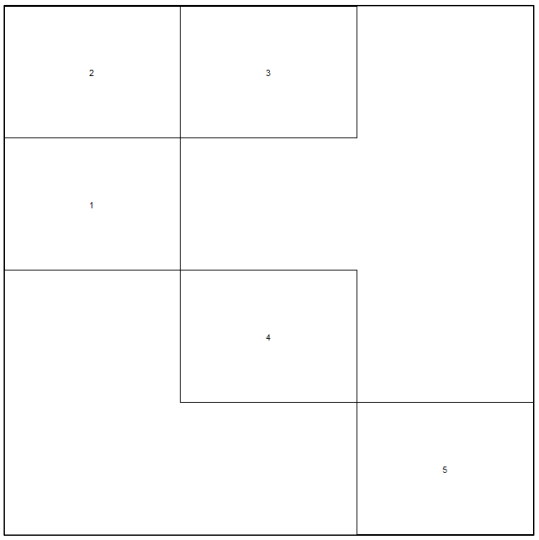
La fonction layout() permet d'obtenir un decoupage plus evolué que l'utilisation de la fonction par(). Par exemple le découpage peut se faire en utilisant une matrice pour définir les sous cadres.

Exemple: Soit la matrice suitvante:

```
mat <- matrix(c(2,3,0,1,0,0,0,4,0,0,0,5),4,3,byrow=TRUE)
```

Utilisons cette matrice pour définir les cadrans en utilisant la fonction layout(). On a:

```
graphics.off() # ferme tous les graphs
x11()
layout(mat) # Crée un cadran de 4 lignes, trois colonnes (soit 12 cadres).
layout.show(5) # affiche les cadres crées.
```

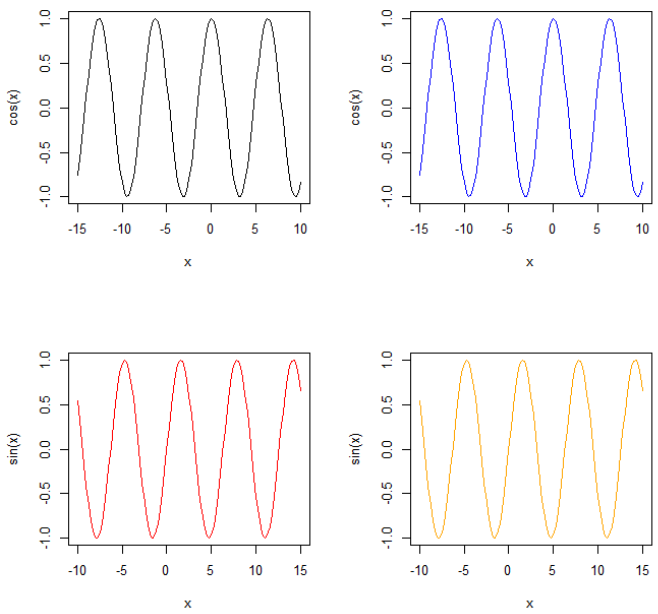


Le vecteur indiqué attribue des numéros à chacun des 12 cadres. Un cadre qui a le numéro 0 ne sera pas affiché sur la fenêtre principale du graphique

Notons qu'a chaque appel d'une fonction R de tracé, les différents graphiques obtenus seront successivement affichés dans les cases numerotées, en suivant l'ordre croissant de ces cases.

Exemple

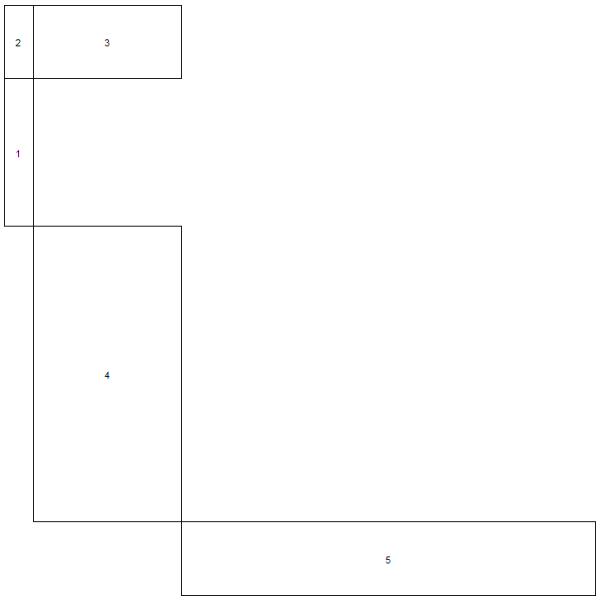
```
graphics.off()
x11()
mat <- matrix(c(1,2,3,4),2,2,byrow=TRUE)
layout(mat)
layout.show(4)
curve(cos(x),xlim=c(-15,10),col="black") #
curve(cos(x),xlim=c(-15,10), col="blue",add=F) #
curve(sin(x),xlim=c(-10,15), col="red",add=F) #
curve(sin(x),xlim=c(-10,15), col="orange",add=F) #
```



Par ailleurs, Il est aussi interessant de noter que l'on peut spécifier, au moyen du paramètre widths, les largeurs respectives de toutes les colonnes de mat les unes par rapport aux autres. On peut faire de même pour les hauteurs respectives des lignes, au moyen du parametre heights.

Exemple:

```
graphics.off()
x11()
mat <- matrix(c(2,3,0,1,0,0,0,4,0,0,0,5),4,3,byrow=TRUE)
layout(mat,widths=c(1,5,14),heights=c(1,2,4,1))
layout.show(5)
```



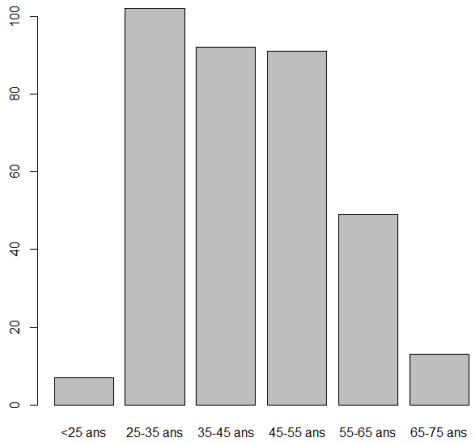
4.2. Graphiques univariés sur les variables qualitatives

4.2.1. Le diagramme de fréquences : barplot()

Le barplot est généralement utilisé lorsqu'on veut visualiser la répartition de l'échantillon selon les modalités d'une variable qualitative (nominale ou ordinale). Il permet de voir l'importance relative e chaque modalité par rapport à l'échantillon total.

Exemple: Répartition de l'échantillon des patients selon les classes d'âge (patientdata)

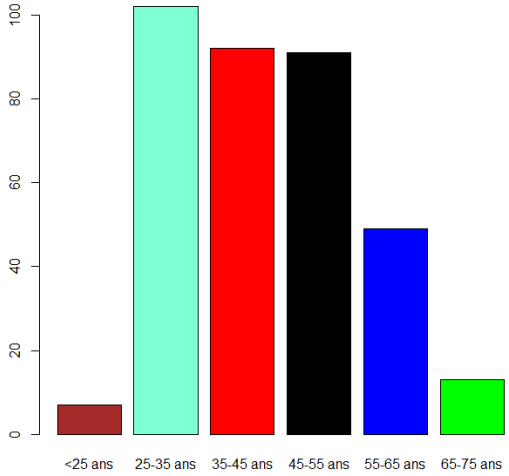
```
graphics.off() ;x11()
barplot(table(mydata$classe_age), horiz = FALSE) # Barre des fréquences (verticales)
```



Par défaut, toutes les barres ont les mêmes couleurs. Pour attribuer des couleurs personnalisées aux barres, il faut définir un vecteur de couleurs et les attribuer aux barres avec l'option col=.

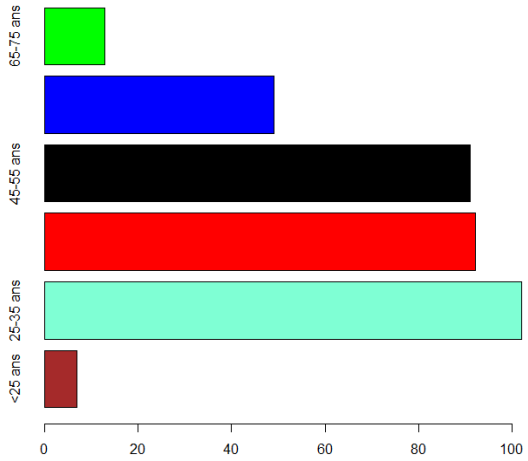
Exemple:

```
mycol<-c( "brown", "aquamarine", "red", "black", "blue","green") # Vecteur de 6 couleurs
barplot(table(mydata$classe_age), horiz = FALSE, col=mycol) # ajoute les couleurs aux barres.
```



Pour tracer des barres horizontales, on modifie la valeur de l'option horiz telle que horiz=TRUE

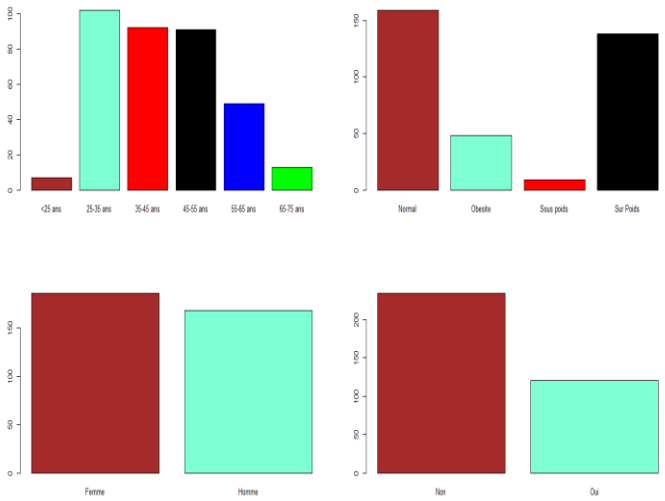
```
barplot(table(mydata$classe_age), horiz = TRUE, col=mycol) # ajoute les couleurs aux barres.
```



Pour ajouter les valeurs des fréquences sur les barres, on peut soit utiliser la fonction text() en indiquant les positions de chaque valeur, soit la fonction locator() pour indiquer la position des valeurs par clic. Nous y reviendrons dans la section consacrée à la mise en forme des graphiques.

Pour tracer plusieurs barplots dans la même zone de graphique dans des cadrans spécifiques, on utilise la fonction par() comme décrit précédemment. Exemple:

```
graphics.off();x11()
par(mfrow=c(2,2))
barplot(table(mydata$classe_age), col=mycol)
barplot(table(mydata$imc_cat), col=mycol)
barplot(table(mydata$sexe), col=mycol)
barplot(table(mydata$maladie_chronique), col=mycol)
```



Pour enregistrer le graphique sur le disque, on utilise saveplot()

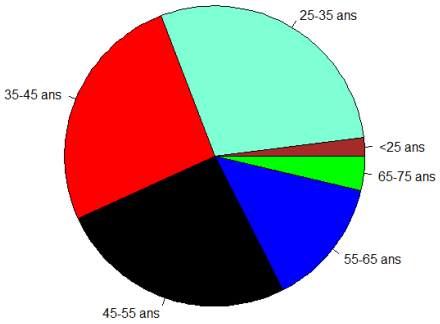
```
savePlot(filename="myBarplot",type="jpeg",device=dev.cur())
```

Ici, le graphique est stocké dans le répertoire courant. Les types de sauvegarde sont: "wmf", "png", "jpeg", "jpg", "bmp","ps" et "pdf"

4.2.2. Le diagramme circulaire: pie()

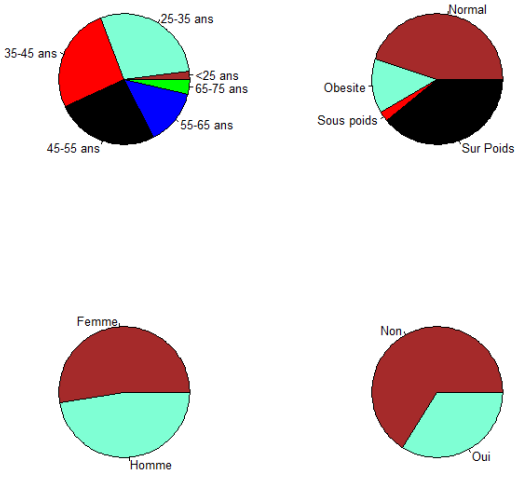
Le diagramme circulaire se réalise comme le diagramme des fréquences en barres en remplaçant la fonction barplot() par la fonction pie(). Exemple:

```
graphics.off() ;x11()
mycol<-c( "brown", "aquamarine", "red", "black",
"blue","green") # définit un vecteur de couleurs
pie(table(mydata$classe_age), col=mycol) # pie
```



Pour tracer plusieurs pie plots dans la même zone de graphique dans des cadrans spécifiques, on utilise la fonction par(). Exemple:

```
graphics.off() ; x11()
par(mfrow=c(2,2))
pie(table(mydata$classe_age), col=mycol)
pie(table(mydata$imc_cat), col=mycol)
pie(table(mydata$sexe), col=mycol)
pie(table(mydata$maladie_chronique), col=mycol)
```



4.3. Graphiques univariés sur les variables quantitatives

4.3.1. L'histogramme: hist()

L'histogramme permet de visualiser la distribution d'une variable quantitative sur un échantillon. Il permet d'illustrer soit les fréquences absolues des valeurs, soit les densités de probabilités des valeurs prises par la variable.

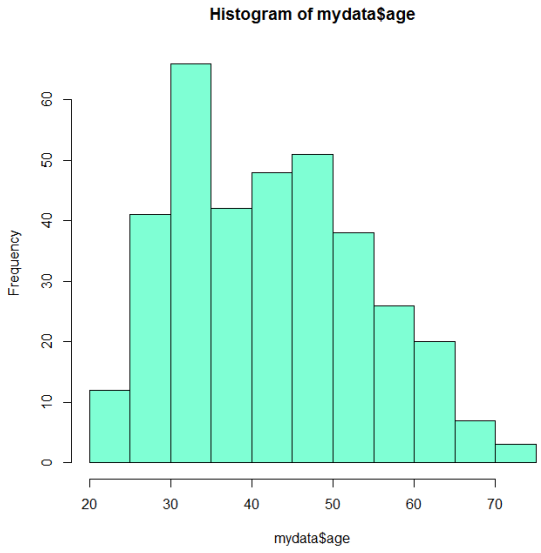
4.3.1.1. Histogramme simple

Pour tracer l'histogramme, on utilise la fonction hist().

Exemple: Histogramme de l'âge des patients de la table de données patientdata

Histogramme avec les fréquences absolues

```
graphics.off();x11()
hist(mydata$age,freq=TRUE,col="aquamarine")
```

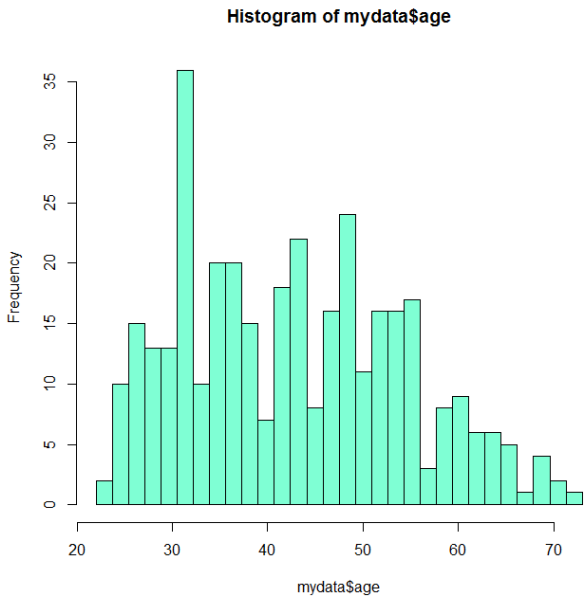


Pour tracer l'histogramme avec les fonctions de densité, on met l'option freq=FALSE

4.3.1.2. Fixer le nombre de bins avec l'option breaks

Pour le fixer le nombre de barres de fréquences de l'histogramme (bins), on ajoute l'option breaks en indiquant un vecteur de valeurs auxquels le découpage doit être fait. Exemple:

```
graphics.off();x11()
hist(mydata$age,freq=TRUE,breaks=seq(min(mydata$age),max(mydat
a$age),by=(     range(mydata$age)[2]-range(mydata$age)[1])/30),
col="aquamarine")
```

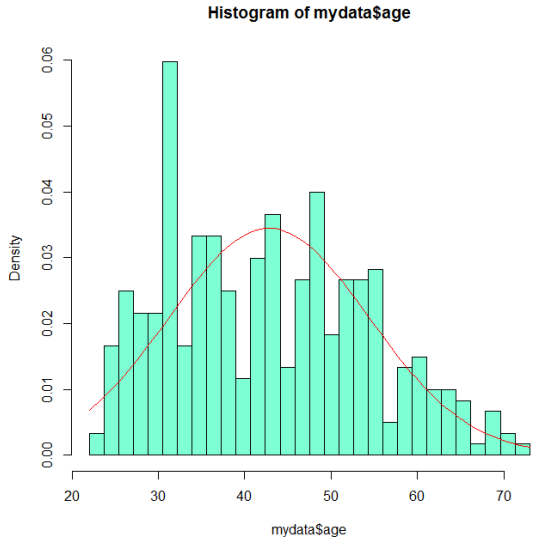


Dans cet exemple, le nombre de bins est fixé à 30 en partant de la valeur minimum de l'âge à la valeur maximum

4.3.1.3. Ajouter la courbe de la loi normale à l'histogramme

Ajouter la courbe de la loi normale à l'histogramme revient à superposer cette courbe à l'histogramme. La superposition se généralement fait en ajoutant l'option add=TRUE (nous reviendrons sur ces aspects plus bas). Il suffit alors de tracer la courbe de la loi normale avec l'option add=TRUE après avoir tracé l'histogramme.
Exemple:

```
hist(mydata$age,freq=FALSE,breaks=seq(min(mydata$age),max(myda
ta$age),by=(    range(mydata$age)[2]-range(mydata$age)[1])/30),
col="aquamarine")
m<-mean(mydata$age, na.rm = TRUE) # moyenne de l'âge
std<-sqrt(var(mydata$age, na.rm = TRUE)) # écart-type de l'âge
curve(dnorm(x, mean=m, sd=std), col="red",add=TRUE) # courbe
```



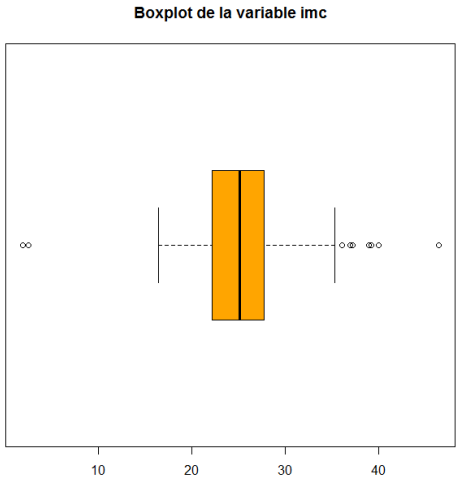
4.3.2. Le Box-Plot

Alors que l'histogramme permet de visualiser la distribution de fréquences ou de probabilités d'une variable quantitative, le Box-Plot permet quant à lui d'illustrer de façon synthétique les caractéristiques de tendance centrale et de dispersion de la variable (moyenne, médiane, étendue, écart interquartile, etc…).

Le Box-plot se réalise avec la fonction boxplot().

Exemple: Box-plot sur l'IMC des patients de l'échantillon

```
graphics.off();x11()
boxplot(mydata$imc, horizontal = TRUE, col="orange") # Box plot horizontal
```



Pour tracer un box-plot vertical, on modifie l'option horizontal telle que horizontal = FALSE

4.4. Elaboration de graphiques croisés

4.4.1. Principe général des graphiques croisés

Les graphiques croisés sont des représentations dans lesquelles on met en relation deux ou plusieurs variables. Plusieurs cas de graphiques croisés peuvent être distingués: cas 1: Croisement entre deux variables qualitatives ; cas 2: croisement entre une variable quantitative et une variable qualitative ; cas 3: croisement entre deux variables quantitatives.

Cas 1: Croisement entre deux variables qualitatives

Pour élaborer le graphique croisé entre deux variables qualitatives, on représente un diagramme de fréquences (ou diagramme circulaire) de la première variable selon chaque modalité de la deuxième variable. En pratique, il s'agit de réaliser des graphiques par sous-groupes.

NB: Ces diagrammes de fréquences (diagrammes circulaires) peuvent être représentés dans un même cadran ou dans des cadrans séparés.

Cas 2: Croisement entre une variable quantitative et une variable qualitative

Dans le cas du croisement entre une variable quanti et une variable quali, on réalise l'histogramme, le box-plot, la courbe d'évolution ou les barres de moyennes de la variable quanti selon les modalités de la variable quali. Là aussi, il s'agit de réaliser des graphiques par sous-groupes.

Il est aussi possible de placer ces sous-graphiques dans un même cadran ou dans des cadrans séparés.

Cas 3: Croisement entre deux variables quantitatives

Le croisement entre deux variables quantitatives se fait généralement au moyen d'un nuage de points ou par une courbe d'évolution (ex: Evolution d'une variable x en fonction du temps).

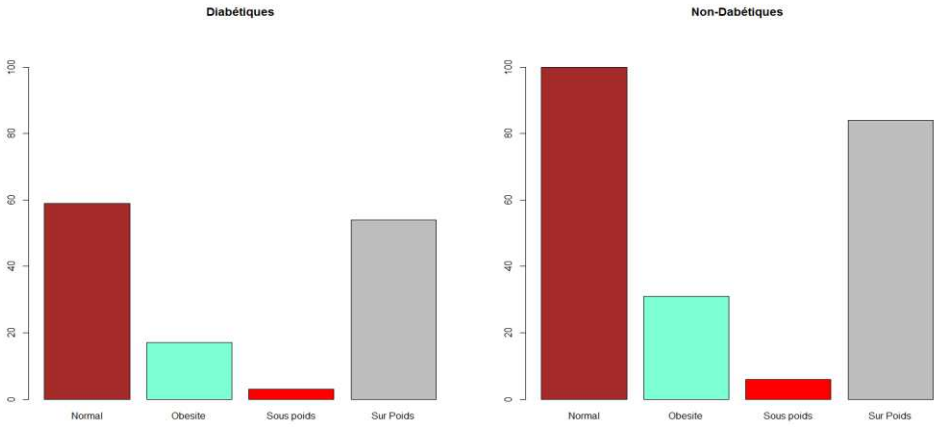
4.4.2. Graphiques croisés entre deux variables qualitatives

Exemple : Réaliser le diagramme de fréquences de la variable imc_cat selon les diabétiques et les non-diabétiques.

Représentation dans des cadrans séparés

```
mycol<-c(     "brown",     "aquamarine",     "red",     "gray",
"blue","green") # Choix du vecteur des couleurs
ymaxv<-
max(rbind(table(mydata$imc_cat[which(mydata$diabete==1)]),tabl
```

```
e(mydata$imc_cat[which( mydata$diabete==0)])))  # choix de la
limite maximal de l'axe y
graphics.off();x11()
par(mfrow=c(1,2))
barplot(table(mydata$imc_cat[which(mydata$diabete==1)]),ylim=c
(0,ymaxv*1.10),  col=mycol,  main=  "Diabétiques",    horiz  =
FALSE) # graphique sur les diabétiques
barplot(table(mydata$imc_cat[which(mydata$diabete==0)]),ylim=c
(0,ymaxv*1.10),col=mycol,  main="Non-Dabétiques",horiz  =  FALSE)
# graphique sur les non diabétiques
```

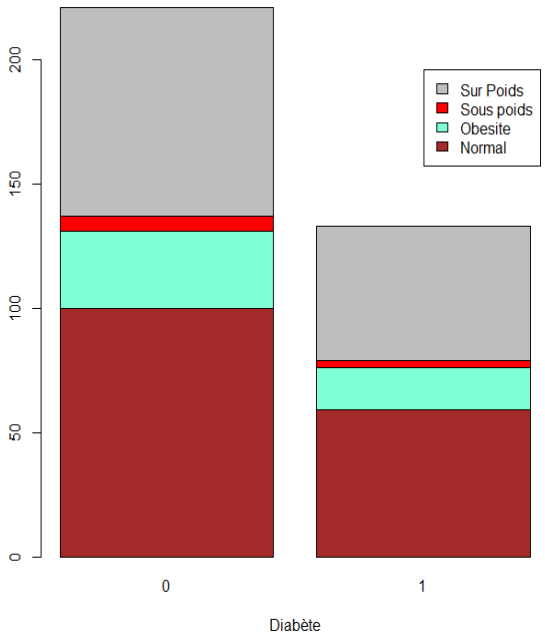


Représentation dans le même cadran

Pour présenter les sous-graphiques dans le même cadran, on fait le barplot sur le tableau de contingence entre les deux variables. Là aussi, deux possibilités se présentent: les barres empilées ou les barres « côte à côte ».

Cas des barres empilées:

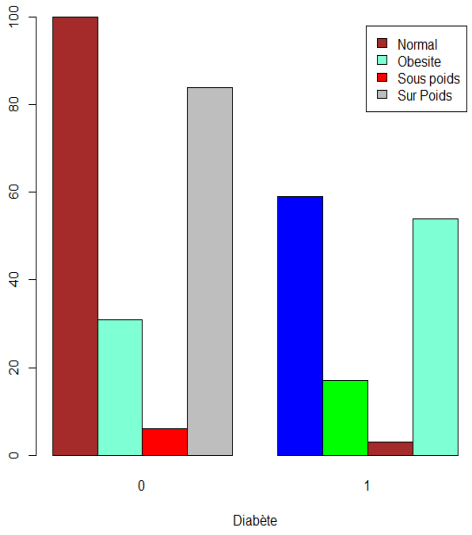
```
mycol<-c(     "brown",     "aquamarine",     "red",     "gray",
"blue","green") # Choix du vecteur des couleurs
graphics.off();x11()
barplot(table(mydata$imc_cat,                    mydata$diabete),
beside=FALSE,col=mycol,     xlab=c("Diabète"),     legend     =
levels(mydata$imc_cat) ,horiz = FALSE)
```



Cas des barres présentées « côte à côte »

Pour présenter les barres « côte à côte » on change la valeur de l'option beside telle que beside=TRUE

```
mycol<-c(     "brown",     "aquamarine",     "red",     "gray",
"blue","green") # Choix du vecteur des couleurs
graphics.off()
x11()
barplot(table(mydata$imc_cat,                    mydata$diabete),
beside=TRUE,col=mycol,     xlab=c("Diabète"),     legend     =
levels(mydata$imc_cat) ,horiz = FALSE)
```

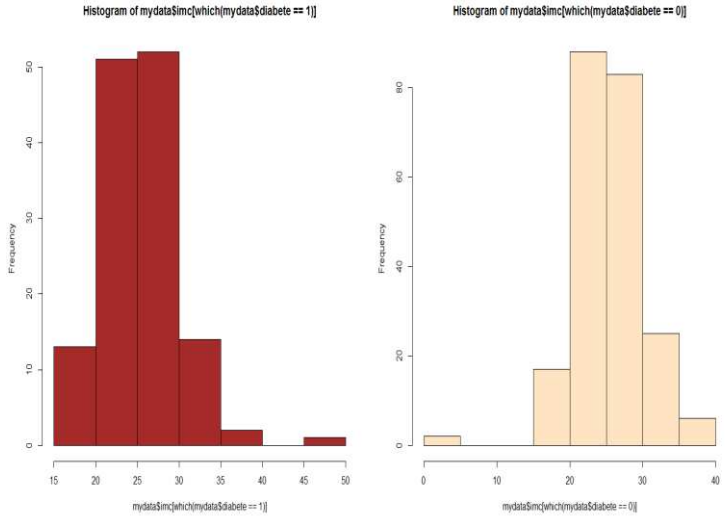


4.4.3. Graphiques croisés entre une variable quantitative et une variable qualitative

Exemple: Histogramme de l'IMC selon les diabétiques et les non-diabétiques

Présentation dans des cadrans séparés

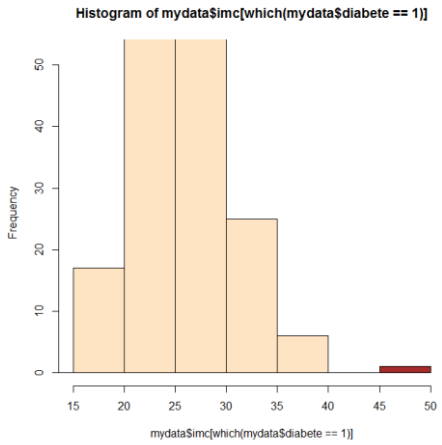
```
graphics.off();x11()
par(mfrow=c(1,2))
hist(mydata$imc[which(mydata$diabete==1)], col="brown")
hist(mydata$imc[which(mydata$diabete==0)], col="bisque")
```



Présentation dans le même cadran

Pour une représentation dans le même cadran, on utilise l'option add=TRUE à la deuxième représentation

```
graphics.off();x11()
hist(mydata$imc[which(mydata$diabete==1)], col="brown")
hist(mydata$imc[which(mydata$diabete==0)],          add=TRUE,
col="bisque")
```



NB: Lorsque les deux distributions sont quasi-équivalentes, le second graphique tend à masquer le premier.

4.4.4. Graphiques croisés entre deux variables quantitatives

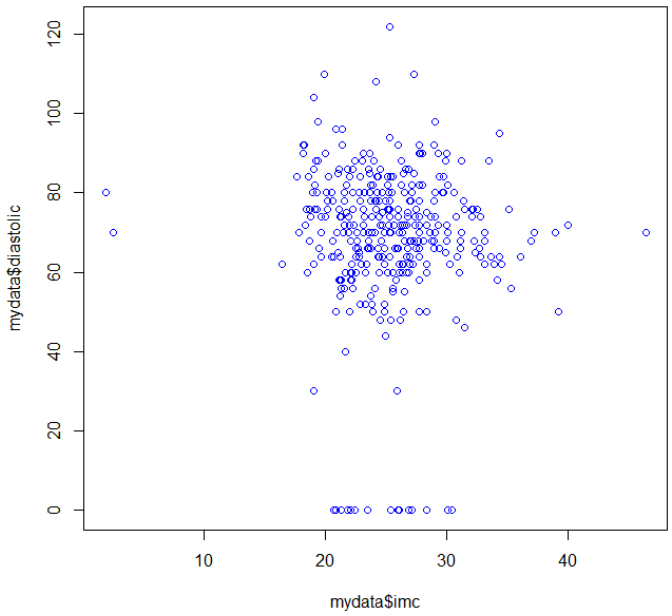
Pour le croisement entre deux variables quantitatives, on utilise généralement la fonction plot().

4.4.4.1. Le nuage de points

Le tracé de nuage de points (scatter plot) est l'une des utilisations les plus courantes de la fonction plot()

Exemple: Le nuage de points entre l'IMC et la pression diastolique des patients.

```
graphics.off();x11()
plot(mydata$imc, mydata$diastolic, type="p", col="red")
```



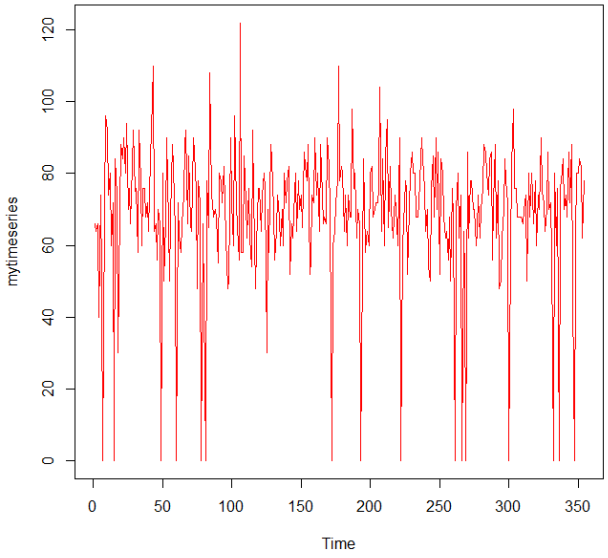
4.4.4.2. La courbe d'évolution

La courbe d'évolution est généralement utilisée pour les données en séries temporelles. Dans ce cas, il s'agit d'un graphique croisé dans lequel le temps intervient comme la seconde variable quantitative.

Pour représenter la courbe d'évolution d'une variable, on utilise une variante de la fonction plot() adaptée aux séries temporelles: plot.ts(). Pour cela la variable à représenter doit d'abord être déclarée comme une série temporelle avec la fonction ts(). Nous donnerons plus de détails sur cette fonction dans le chapitre consacré à l'étude des séries temporelles.

Pour l'instant, considérons un cas fictif de série temporelle myseries défini comme suit:

```
mytimeseries<-ts(mydata$diastolic)
plot.ts(mytimeseries, col="red")
```



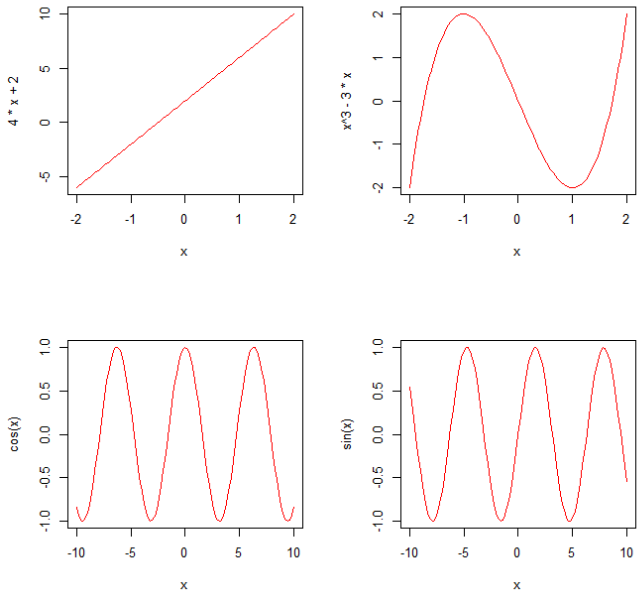
4.5. Tracer un graphique à partir d'une fonction analytique : la fonction curve()

La fonction curve() permet de tracer n'importe quelle fonction analytique que nous lui fournissons en argument: fonction sinus, cosinus, exponentielle, polynomiale, etc…

Contrairement aux fonctions de tracé, la fonction curve ne nécessite pas une variable préalable. Il suffit alors simplement d'indiquer une fonction de x sur un intervalle de tracé (appartenant bien sûr au domaine de définition de la fonction).

Exemples:

```
graphics.off() ; x11() ; par(mfrow=c(2,2))
curve(4*x+2,col="red", xlim=c(-2,2)) # fonction linéaire de -2
à 2
curve(x^3-3*x,col="red", xlim=c(-2,2)) # fonction polynomiale
curve(cos(x), col="red",xlim=c(-10,10)) # une fonction cosinus
curve(sin(x), col="red",xlim=c(-10,10)) # une fonction sinus
```



4.6. Superposition de graphiques

4.6.1. Principe général de superposition de graphiques

Par défaut, lorsqu'on trace un graphique, celui-ci remplace automatiquement le précédent dans la fenêtre graphique. En utilisant la fonction par(mfrow=) pour définir les cadrans, chaque graphique tracé par la suite se positionnera dans un cadran spécifique (dans la limite du nombre de cadrans).

Mais, il arrive très souvent de vouloir superposer plusieurs graphiques, soit sur la même fenêtre graphique principale, soit dans le même cadran. Dans ce cas, il faut utiliser les fonctions de superposition de graphiques:

Les principales fonctions de superposition de graphiques sont:

- l'option add=TRUE: disponible dans la plupart des fonctions de tracées comme barplot(), hist(), curve(),…
- La fonction lines(): permettant d'ajouter une ligne à un graphique précédemment tracé
- La fonction abline(): permettant d'ajoutant une droite dont les paramètres sont à indiquer.

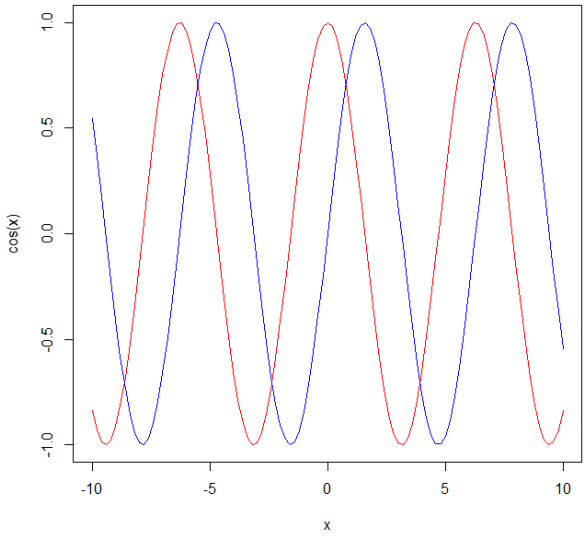
Ci-après les détails sur l'utilisation de chacune des fonctions.

4.6.2. L'utilisation de l'option add=TRUE

L'option add=TRUE est disponible dans la plupart des fonctions de tracé graphique. Dans ces fonctions l'option add est fixé par défaut à FALSE. Pour superposer le graphique au tracé précédent, il faut donc activer cette option avec la valeur TRUE.

Exemple:

```
graphics.off() ; x11() # Ouvre une nouvelle fenêtre graphique
curve(cos(x), col="red",xlim=c(-10,10)) # Trace un premier
graphique
curve(sin(x),col="blue",add=TRUE) # Trace un second graphique
et le superpose au précédent.
```

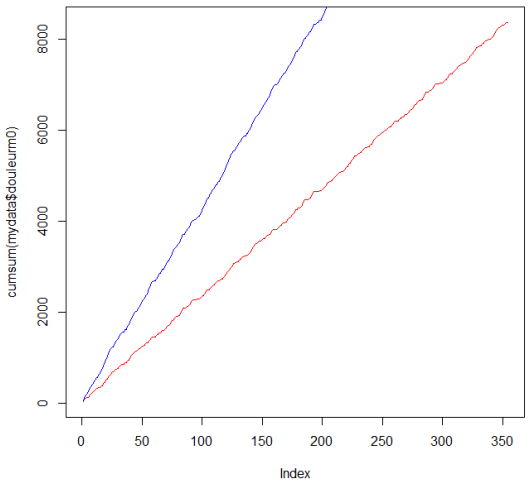


4.6.3. L'utilisation de la fonction lines()

La fonction lines() permet de superposer une ligne à un graphique précédemment tracé. Elle est généralement utilisée après une fonction plot().

Exemple:

```
graphics.off() ; x11()
plot(cumsum(mydata$douleurm0), type="l", col="red") # Trace la
ligne douleurm0
lines(cumsum(mydata$douleurm1), col="blue") # ajoute la ligne
douleurm1
```

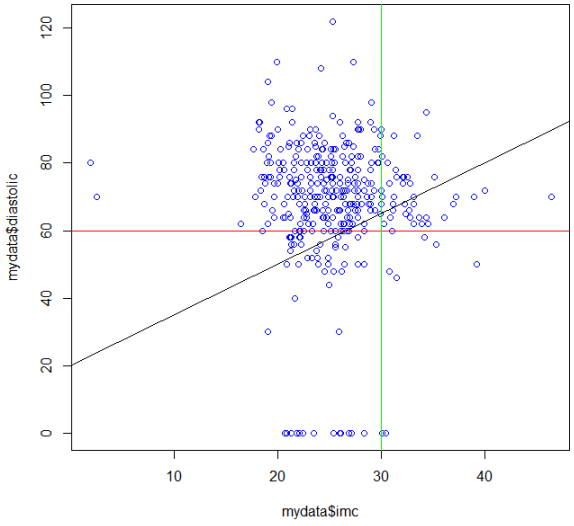


4.6.4. L'utilisation de la fonction abline()

La fonction abline() permet soit de tracer une droite d'équation y = a + bx (spécifiée par les paramètres a et b). Elle permet également de tracer une ligne horizontale (paramètre h) ou une ligne verticale (parametre v). Elle est généralement utilisée après une fonction plot() notamment un scatter plot.

Exemple:

```
graphics.off() ; x11()
plot(mydata$imc, mydata$diastolic, type="p", col="blue")
abline(h=60, col="red") # ajoute une droite horizontale à
l'ordonnée 60
abline(v=30, col="green") # ajoute une droite verticale à
l'abscisse 30
abline(a=20,b=1.5) # ajoute une droite (a+bx) de pente b=1.5
et d'ordonnée à l'origine a=20
```



4.7. Jointure des points d'un graphique : les fonctions segments() et arrows()

La fonction segments() permet de superposer un graphique en joignant les points par des segments de droite.

Quant à la fonction arrows(), elle superpose le graphique en joignant les points par des flèches.

Ces fonctions sont généralement utilisées après une plot(). Et nécessitent une double coordonnée.

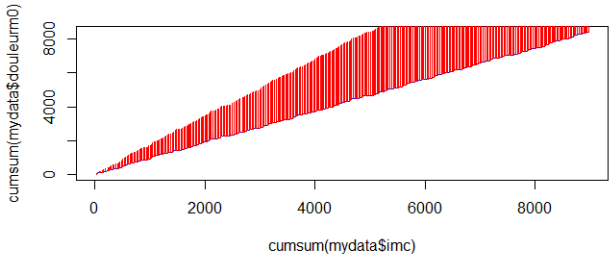
Exemple:

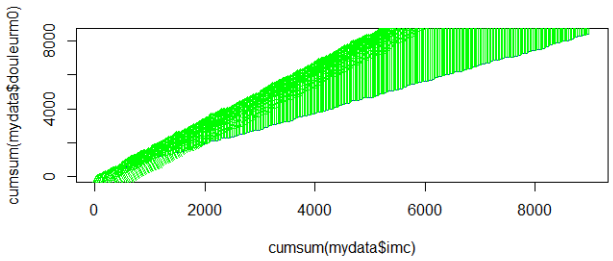
Traçons la somme cumulée de la variable « douleurm0 » en fonction de la somme cumulée de la variable « imc ». Ensuite ajoutons successivement:

-Les segments représentant les écarts entre la somme cumulée de douleurm0 et la somme cumulée de douleurm1

- Les flèches représentant les écarts entre la somme cumulée de douleurm0 et la somme cumulée de douleurm1

```
graphics.off() ; x11() ;par(mfrow=c(2,1))
# cadran avec les segments
plot(cumsum(mydata$imc), cumsum(mydata$douleurm0), type="l",
col="blue")
segments(cumsum(mydata$imc), cumsum(mydata$douleurm0),
cumsum(mydata$imc), cumsum(mydata$douleurm1), col="red") #
Trace les segments
# cadran avec les flèches
plot(cumsum(mydata$imc), cumsum(mydata$douleurm0), type="l",
col="blue")
arrows(cumsum(mydata$imc), cumsum(mydata$douleurm0),
cumsum(mydata$imc), cumsum(mydata$douleurm1), col="green")
#ajoute les flèches
```





4.8. Mise en forme du graphique

4.8.1. Gestion des couleurs du graphique

Il existe 657 couleurs prédéfinies dans le système R. Pour accéder à la liste de ces couleurs, on utilise la fonction color():

color()

Mais en utilisant le système codage RVB(Rouge-Vert-Bleu) en anglais RGB (Red-Green-Blue), on peut constituer jusqu'à 16 millions de couleurs. Voir la documentation sur la fonction rgb().

Pour ce qui concerne l'utilisation des couleurs pour la mise en forme des graphiques, on distingue plusieurs niveaux d'utilisation:

- Coloriage de l'arrière-plan de la fenêtre du graphique
- Couleur des cadres et pourtours de la fenêtre de graphique
- Couleur de l'arrière-plan du graphique
- Couleur des cadres et pourtours du graphique
- Couleur des annotations (titres, légendes, labels des axes, etc…)

Plusieurs coloriages peuvent être faits lors du paramétrage de la fenêtre graphique avec la fonction par()

Exemple:

```
graphics.off();x11()
```

```
par(mfrow=c(1,1),
bg="lightgray",col.axis="darkgreen",col.lab="darkred",col.main
="purple",col.sub="black", fg="blue")
```
Dans cet exemple:

- La couleur d'arrière plan de la fenêtre principale « bg » est fixée à "lightgray"
- Couleur des labels des graduations des axes « col.axis » fixée à "darkgreen"
- Couleur des titres des axes « col.lab » fixée à "darkred"
- La couleur du titre général du graphique « col.main » fixée à "purple"
- La couleur du sous-titre général du graphique « col.sub » fixée à black"
- La couleur du tracé des axes « fg » fixée à "blue"

Le paramétrage défini dans la fonction par() se répercute alors sur tous les graphiques tracés à la suite de la fonction

Exemple:

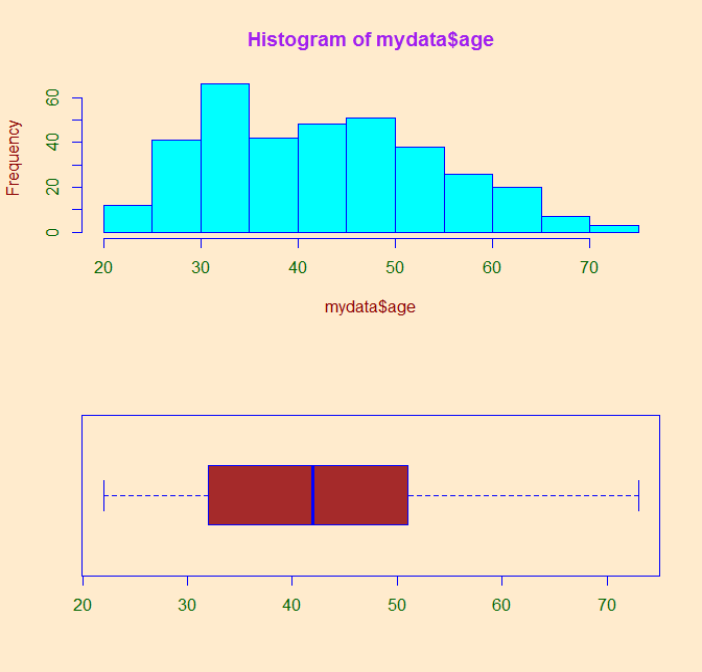
```
graphics.off()
x11()
par(mfrow=c(1, 1), bg ="lightgray",col.axis= "darkgreen",
col.lab="darkred",col.main="purple",col.sub="black",
fg="blue")
hist(mydata$age) # trace un histogramme
```

4.8.2. Couleur du tracé du graphique: l'option col=

La couleur du tracé d'un graphique se gère à partir de l'option « col » disponible dans chaque fonction graphique.

Exemple:

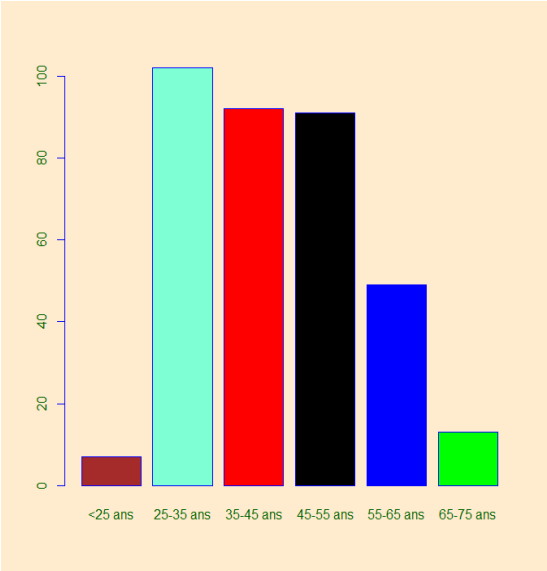
```
graphics.off();x11()
par(mfrow=c(2, 1), bg ="blanchedalmond",col.axis= "darkgreen",
col.lab="darkred",col.main="purple",col.sub="black",
fg="blue")
hist(mydata$age, col="cyan") # histogramme avec col="cyan"
boxplot(mydata$age, horizontal = TRUE, col= "brown") # boxplot
avec col="brown"
```



NB: Lorsque plusieurs couleurs doivent être utilisées pour tracer un graphique (par exemple un graphique en barres avec une couleur par barre), il est préférable de définir en amont un vecteur de couleurs et d'associer ce vecteur à l'option col.

Exemple:

```
graphics.off();x11()
par(mfrow=c(1, 1), bg ="blanchedalmond",col.axis= "darkgreen",
col.lab="darkred",col.main="purple",col.sub="black",
fg="blue")
mycol<-c( "brown", "aquamarine", "red", "black",
"blue","green") # Vecteur de 6 couleurs
barplot(table(mydata$classe_age), horiz = FALSE, col=mycol) #
ajoute les couleurs aux barres
```

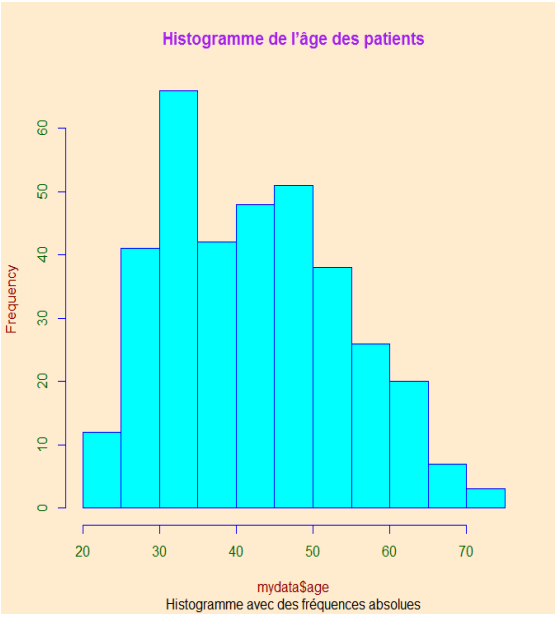


4.8.3. Titre et sous-titre du graphique: les options main= et sub=

Le titre et le sous-titre du graphique sont spécifiés respectivement avec les options main et sub de la fonction de tracé du graphique:

Exemple:

```
graphics.off();x11()
par(mfrow=c(1, 1), bg ="blanchedalmond",col.axis= "darkgreen",
col.lab="darkred",col.main="purple",col.sub="black",
fg="blue")
hist(mydata$age, col="cyan", main="Histogramme de l'âge des
patients", sub="Histogramme avec des fréquences absolues")
```

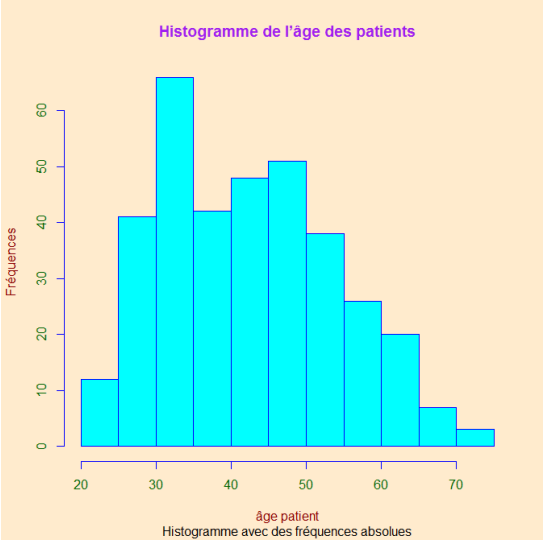


4.8.4. Labels des axes: les options xlab= et ylab=

Les options xlab et ylab permettent d'indiquer respectivement les labels de l'axe des abscisses et de l'axe des ordonnées.

Exemple:

```
graphics.off();x11()
par(mfrow=c(1, 1), bg ="blanchedalmond",col.axis= "darkgreen",
col.lab="darkred",col.main="purple",col.sub="black",
fg="blue")
hist(mydata$age, col="cyan", main="Histogramme de l'âge des
patients", sub="Histogramme avec des fréquences absolues",
xlab="âge patient", ylab="Fréquences")
```



4.8.5. Utilisation de la fonction title() pour les titres des axes et du graphique

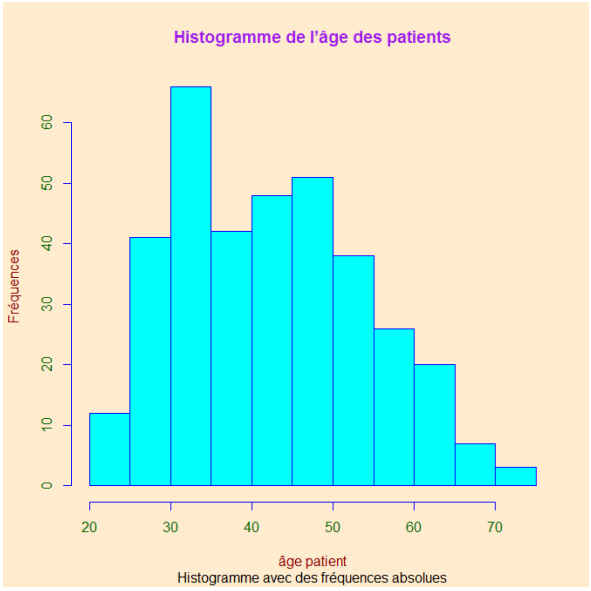
La fonction title() permet de gérer dans une seule instruction toutes les mises précédemment décrites à savoir : le titre général du graphique, le sous-titre du graphique et les titres des axes.

Cette fonction est utilisée après le tracé du graphique

Exemple:

```
graphics.off() ;x11()
par(mfrow=c(1, 1), bg ="blanchedalmond",col.axis= "darkgreen",
col.lab="darkred",col.main="purple",col.sub="black",
fg="blue")
hist(mydata$age, col="cyan",main="", sub="", xlab="", ylab=""
) # Trace l'histogramme sans titre
```

```
title(main="Histogramme de l'âge des patients", sub="Histogramme avec des fréquences absolues", xlab="âge patient", ylab="Fréquences") # Ajoute les titres
```



4.8.6. Personnalisation des axes du graphique: la fonction axis()

En plus de la mise en forme effectuée dans les paramètres généraux avec la fonction par(), on peut effectuer les mises en formes spécifiques des axes en utilisant la fonction axis().

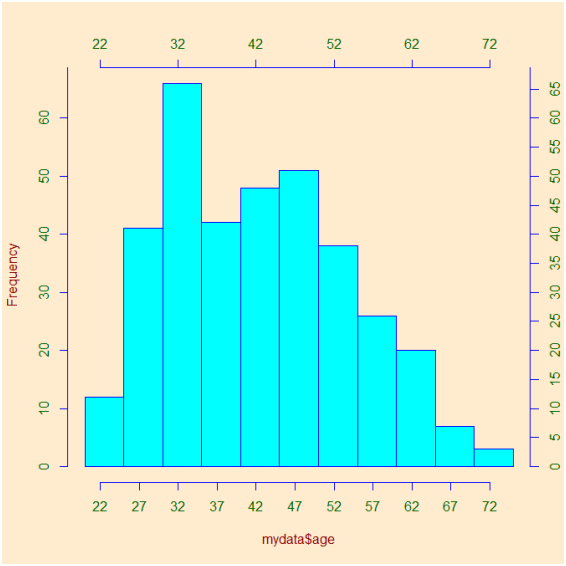
Les principales options de la fonction axis() sont:

- side: indique l'axe concerné : side=1 (axe d'en bas), side=2 (axe de gauche), side=3 (axe d'en-haut), side=4 (axe de droite).
- at: un vecteur indiquant les points de graduation.
- labels: un vecteur indiquant les labels des graduations apparaissant aux points définis par at
- tick: Variable booléen (TRUE ou FALSE) qui indique si les tirets de graduations doivent être tracés ou non.
- col: indique la couleur de l'axe (remplace alors la couleur définie pour cet axe par la fonction par().

Exemple

```
par(mfrow=c(1, 1), bg ="blanchedalmond",col.axis= "darkgreen",
col.lab="darkred",col.main="purple",col.sub="black",
fg="blue")
hist(mydata$age, col="cyan", main="", axes=FALSE)
```

```
axis(side=2, at=c(seq(0,70, by=10)), labels=c(seq(0,70, by=10))) # Graduation de l'axe y1 entre 0 et 70 avec des labels de 0 à 70
axis(side=1, at=c(seq(min(mydata$age),max(mydata$age), by=5)), labels=c(seq(min(mydata$age),max(mydata$age), by=5))) # graduation axe x1
axis(side=3, at=c(seq(min(mydata$age),max(mydata$age), by=10)), labels=c(seq(min(mydata$age),max(mydata$age), by=10))) # graduation axe x2
axis(side=4, at=c(seq(0,70, by=5)), labels=c(seq(0,70, by=5))) # Graduation de l'axe y2 entre 0 et 70 avec des labels de 0 à 70
```

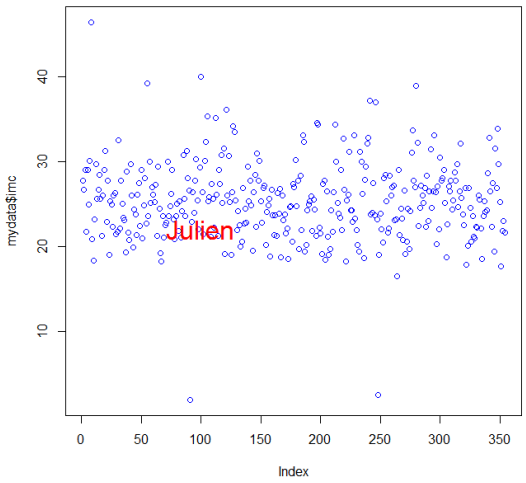


4.8.7. Placer du texte sur le graphique : les fonctions text() et locator()

La fonction text() permet ajouter du texte ou une expression mathématique sur le graphique à un emplacement indiqué par les coordonnées x et y. Tandis que la fonction locator() combinée avec la fonction text() permet de placer le texte sur le graphique à un emplacement indiqué par simple clic.

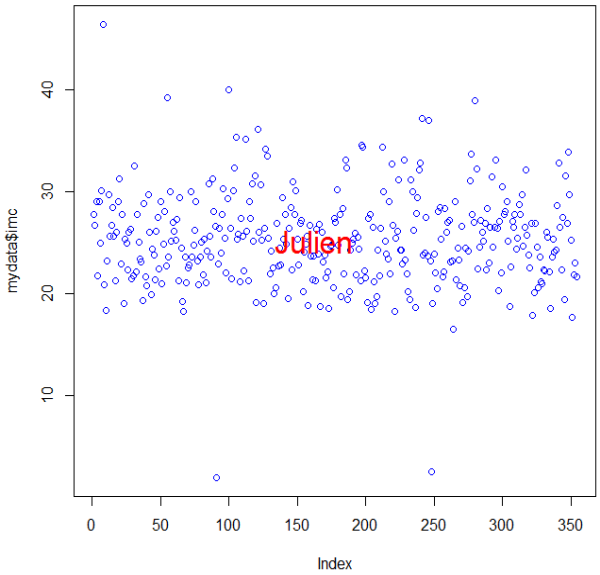
Exemple: Utilisation de la fonction text()

```
graphics.off();x11()
plot(mydata$imc, type="p", col="blue")
text(100,22,"Julien", col="red", cex=2) # ajout du texte à l'emplacement de coordonnés 100 et 22.
```



Exemple: La fonction text() combinée avec la fonction locator()

```
graphics.off();x11()
plot(mydata$imc, type="p", col="blue")
text(locator(1),labels=c("Julien"), col="red", cex=2) # Cliquez alors 1 fois sur la fenêtre graphique à l'emplacement souhaité et à la fin le texte "Julien" s'affichera.
```

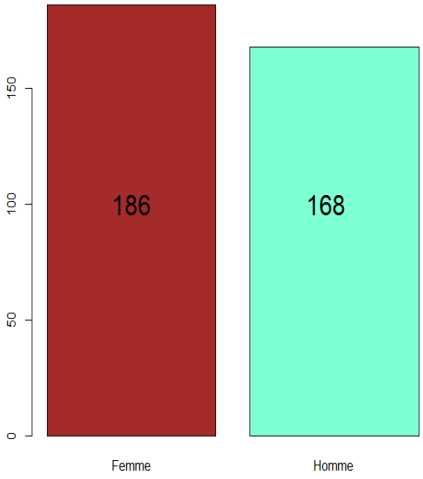


NB: Les fonctions text() et locator() peuvent être utilisées pour faire afficher sur le graphique les chiffres associés à un graphique. Par exemple les valeurs des fréquences pour un diagramme de fréquences, etc…

Toutefois avec la fonction text(), il faut indiquer l'emplacement des valeurs en précisant leurs coordonnées. Exemple:

```
val<-table(mydata$sexe) # Construit le tableau
```

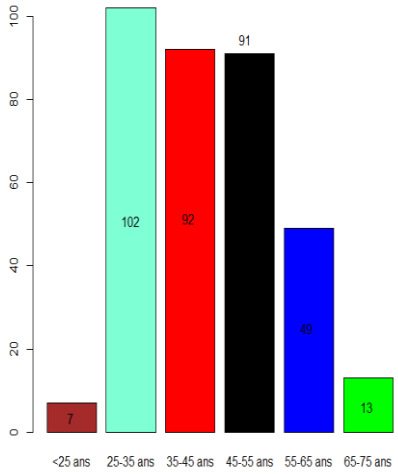
```
graphics.off();x11()
barplot(table(mydata$sexe), col=c("brown", "aquamarine"))
text(0.7,100,val[1], col="black", cex=2) # ajoute val[1] au
point (0,7; 100)
text(1.85,100,val[2], col="black", cex=2) #ajoute val[2] au
point (1,85; 100)
```



L'inconvénient de la fonction text() est qu'il faut la spécifier autant de fois qu'il y a de valeurs à afficher. En plus, il faut indiquer, pour chaque texte, les coordonnées de l'emplacement. Ce qui est très fastidieux.

On peut alors combiner la fonction text() avec la fonction locator() afin de placer les valeurs avec des simples clics sur les emplacements souhaités. **Exemple:**

```
mycol=c("brown", "aquamarine", "red", "black", "blue","green")
# couleurs
val<-table(mydata$classe_age) # tableau de fréquences
graphics.off();x11()
barplot(table(mydata$classe_age), col=mycol)
text(locator(6),labels=c(val)) # Cliquez sur la fenêtre
graphique 6 fois
```

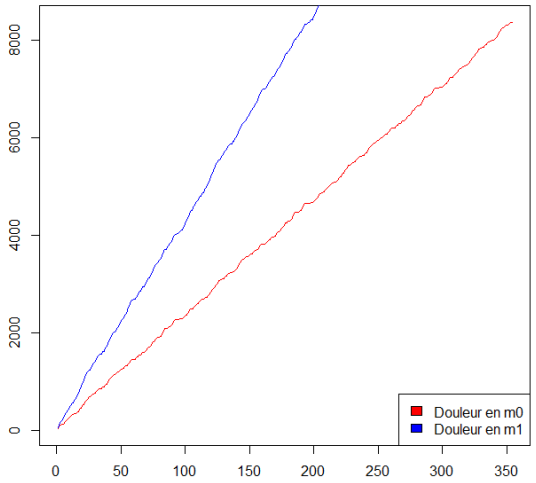


4.8.8. Ajouter une légende au graphique : la fonction legend()

Exemple: Représentons sous forme de courbe la somme cumulée des deux variables « douleurm0 » et « douleurm1 » et attribuons des légendes. On a:

```
graphics.off() ;x11()
plot(cumsum(mydata$douleurm0), col="red", type="l", main="", sub="", xlab="", ylab="")
lines(cumsum(mydata$douleurm1), col="blue", main="", sub="", xlab="", ylab="")
legend("bottomright", legend=c("Douleur en m0", "Douleur en m1") , fill=c("red", "blue"))# Attribue une légende avec les couleurs correspondantes
```

Les emplacements standards sont: "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" and "center"

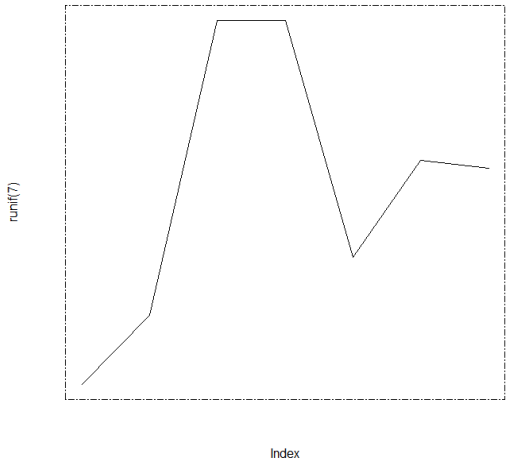


4.8.9. Encadrer le graphique: la fonction box()

La fonction box() permet d'ajouter une boîte autour du graphique courant. Le paramètre bty permet de gérer le type de boîte ajoute, le parametre lty spécifie le type de ligne utilisé pour tracer la boîte.

Exemple

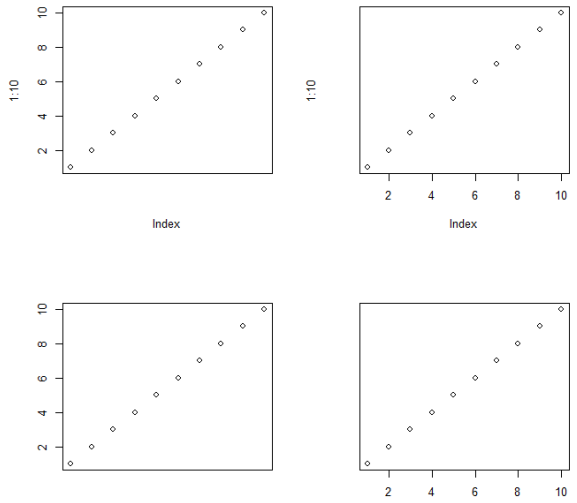
```
plot(runif(7), type = "l", axes = FALSE) # tracé du graphique
sans graduation (axes=FALSE)
box(lty = "1373") # Ajout d'un box
```



4.8.10. Enlever les graduations par défaut sur l'axe des abcisses et l'axe des ordonnées : les options xaxt='n' et yaxt='n'

Exemples :

```
graphics.off()
x11()
par(mfrow=c(2,2))
plot(1:10, xaxt='n') # supprime les graduations de l’axe x
plot(1:10, yaxt='n') # supprime les graduations de l’axe x
plot(1:10,  xaxt='n',  ann=FALSE)  #  Supprime  toutes  les
annotations de l’axe x
plot(1:10,  yaxt='n',  ann=FALSE)  #  Supprime  toutes  les
annotations de l’axe y
```



4.9. Utilisation du package ggplot2 pour des graphiques plus élaborés

Jusqu'à présent, nos présentations ont été essentiellement basées sur l'utilisation du package « base » pour le tracé des graphiques. Il existe néanmoins d'autres packages puissants tels que ggplot2 qui permettent de construire des graphiques plus élaborés et plus personnalisés. Ici, nous faisons une présentation sommaire de la fonction ggplot() du package ggplot2 pour le tracé des graphiques courants.

4.9.1. Principe général de l'utilisation de la fonction ggplot()

Avant de présenter la fonction ggplot(),installons d'abord le package ggplot2 et le charger dans l'environnement R.

```
install.packages("ggplot2")
```

```
library("ggplot2")
```

La syntaxe générale du tracé d'un graphique quelque soit le type de graphique se présente comme suit :

```
ggplot(data, aes(x, y)) + geom_*()
```

data représente la table de données

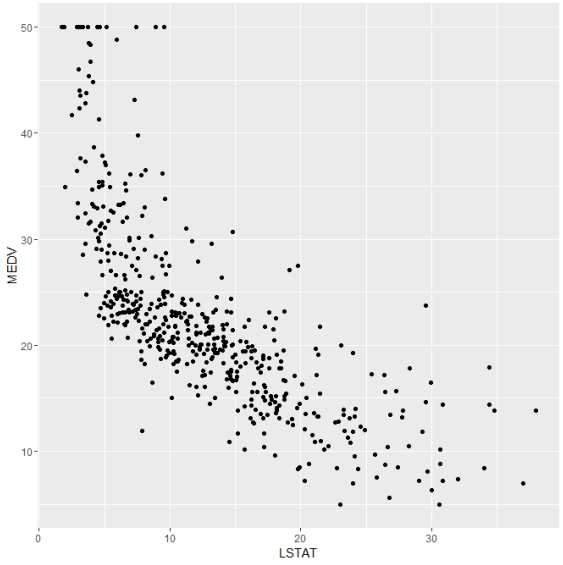
le paramètre aes() permet d'indiquer les variables x et y sont les variables à représenter. On spécifie une seule variable lorsq'il s'agit d'un graphique univariés.

Le paramètre geom_*() permet de définir le type de graphique à tracer ainsi que les mides en formes du tracé. Les principales fonctions sont les suivantes :

- geom_point() : trace des points ;
- geom_line() : trace une ligne ;
- geom_polygon() : tracer de polygoles
- geom_path() : tracer des points dans l'ordre du data.frame ;
- geom_step() : trace un graphique en escalier ;
- geom_boxplot() : tracer une boîte à moustache (blox-plot) ;
- geom_jitter() : mettre des points côte à côte pour une variable catégorielle ;
- geom_smooth() : ajoute une courbe de tendance ;
- geom_histogram() : trace un histogramme ;
- geom_bar() : trace un diagramme en bâton ;
- geom_density() : tracer une estimation de densité.

Exmple : tracé d'un nuage de points entre la variable LSTAT et MEDV

```
graphics.off() ;x11()
library("ggplot2")
ggplot(data = mydata, aes(x = LSTAT, y = MEDV)) + geom_point()
# utilisation de geom_point() sans option
```



NB : Toutes les fonctions geom_*() possèdent les paramètres optionnels suivants : data, mapping, geom (ou stat) et position. S'ils sont omis, les valeurs par défaut seront prises à partir des valeurs définie par le paramètre aes() de la fonction ggplot(). Si à l'inverse ils sont renseignés, alors leur valeur vient remplacer celle héritée du paramètre aes() de ggplot(). Certaines fonctions ont d'autres paramètres ;

L'un des grands avantages de l'utilisation de la fonction ggplot() par rapport aux fonctions de base c'est qu'il est possible de faire des superpositions de graphiques en utilisant simplement l'opérateur "+" au graphique précédent. Cette forme générale se présente alors comme suit :

mygraph <- ggplot(data, aes(x, y)) + geom_*() # définit un graphique mygraph

mygraph+ new_ geom_*() # ajoute un nouveau graphique et le superpose à mygraph. Cette propriété permet par exemple de tracer plusieurs types de garphique sur la même zone.

4.9.2. Tracé de nuage de points

4.9.2.1. Nuage de points simple

Exemples

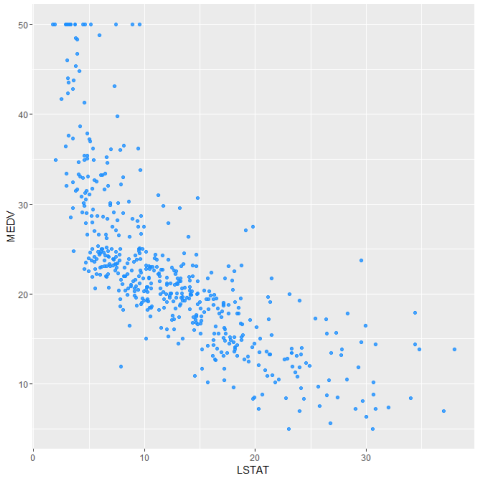
```
graphics.off()
x11()
par(mfrow=c(1,2))
library("ggplot2")
ggplot(data = mydata, aes(x = LSTAT, y = MEDV)) + geom_point()
# utilisation de geom_point() sans option
```

Parmi les paramètres de mise en forme de geom_point que l'on peut modifier, il y a :

- colour : la couleur ;
- shape : la forme ;
- size : la taille ;
- alpha : la transparence ;
- fill : le remplissage.

Exemple :

```
ggplot(data = mydata, aes(x = LSTAT, y = MEDV)) +
geom_point(colour = "dodger blue", alpha = .8) # layer
geom_point()# utilisation de geom_point() sans option
```



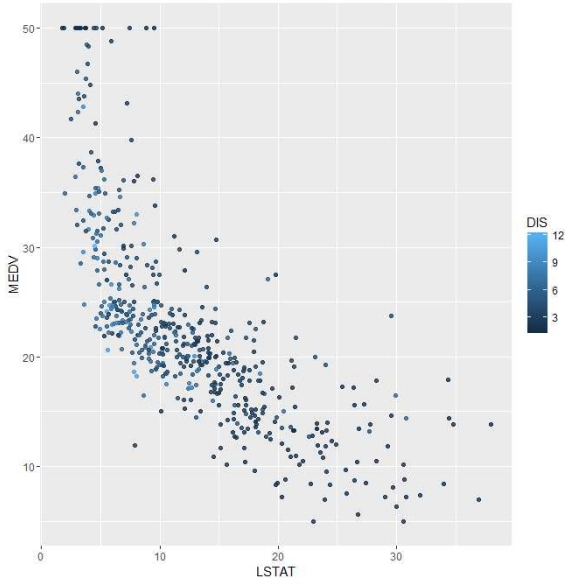
4.9.2.2. Mise en forme du nuage de points

Fixer la couleur des points en fonction des valeurs d'une variable

Il est possible de faire dépendre la couleur prise par une variable continue, une échelle de couleurs sera utilisée ; si la variable est discrète, une palette de couleurs est utilisée. Pour ce faire, il faut indiquer le mapping à l'intérieur de la fonction aes().

Exemple :

```
graphics.off()
x11()
library("ggplot2")
ggplot(data = mydata, aes(x = LSTAT, y = MEDV)) +
geom_point(alpha = .8, aes(colour = DIS))
```

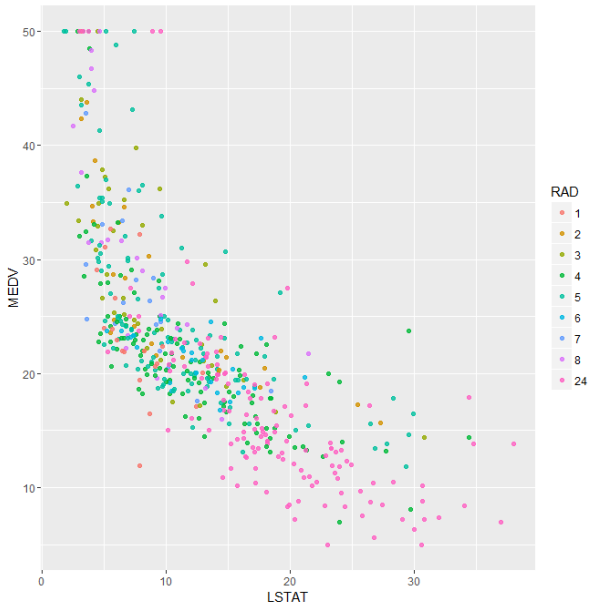


Dans cette représentation, la couleur des points varie en fonction des valeurs de la variable DIS

Lorsque la variable en fonction de laquelle les couleurs sont fixées est catégorielle (factor), R définit une palette de couleurs.

Exemple:

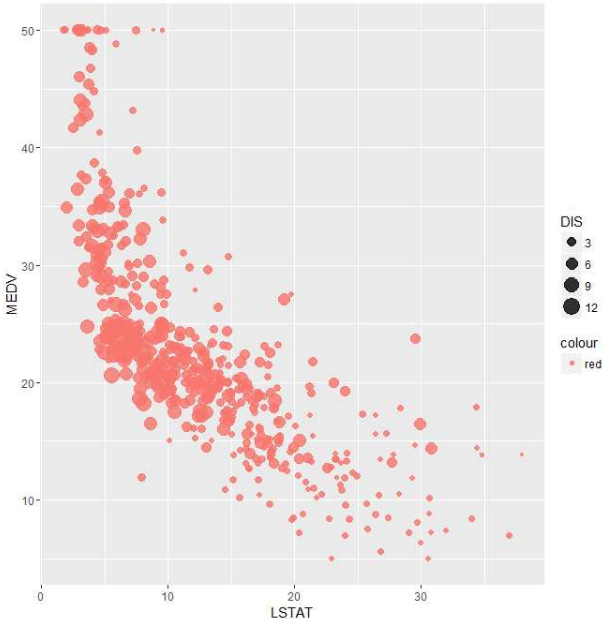
```
graphics.off()
x11()
library("ggplot2")
ggplot(data = mydata, aes(x = LSTAT, y = MEDV)) +
geom_point(alpha = .8, aes(colour = RAD)) # variation de la
couleur en fonction de la variable DIS
```



Agrandir le volume des points en fonction des valeurs d'une variable

Exemple

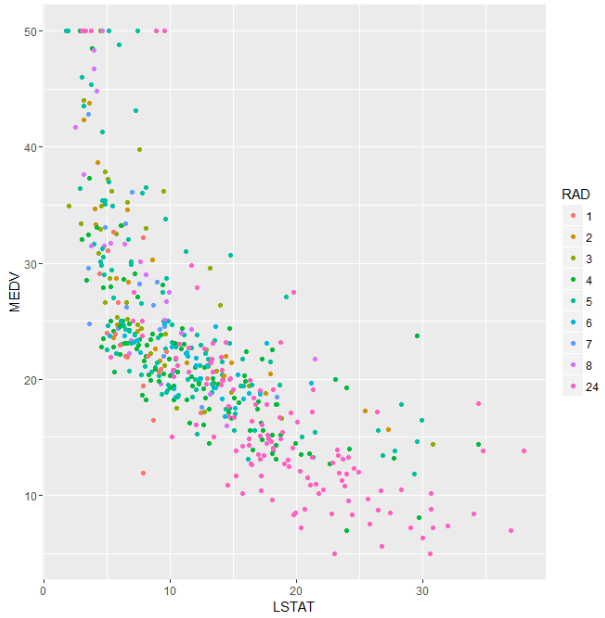
```
graphics.off()
x11()
library("ggplot2")
ggplot(data = mydata, aes(x = LSTAT, y = MEDV)) +
geom_point(alpha = .8, aes(colour= "red",size = DIS)) #
variation de la taille en fonction de la variable DIS
```



Tracer un nuage de points par catégorie

Pour tracer un nuage de points par groupe (selon les valeurs d'une variable catégorielle), on ajoute l'option color=GroupeVariable dans le membre aes principal.
Exemple:

```
graphics.off()
x11()
library("ggplot2")
ggplot(data = mydata, aes(x = LSTAT, y = MEDV, color=RAD)) +
geom_point()
```

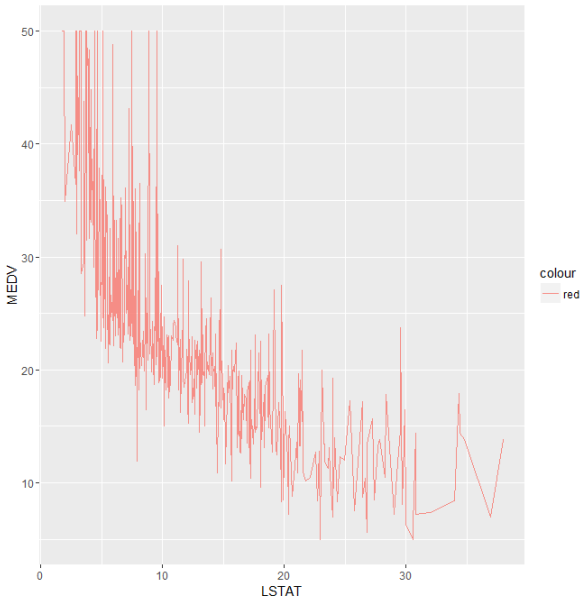


4.9.3. Tracé de graphique en ligne

Pour tracer un graphique en ligne, il faut utiliser la fonction geom_line().

Exemple:

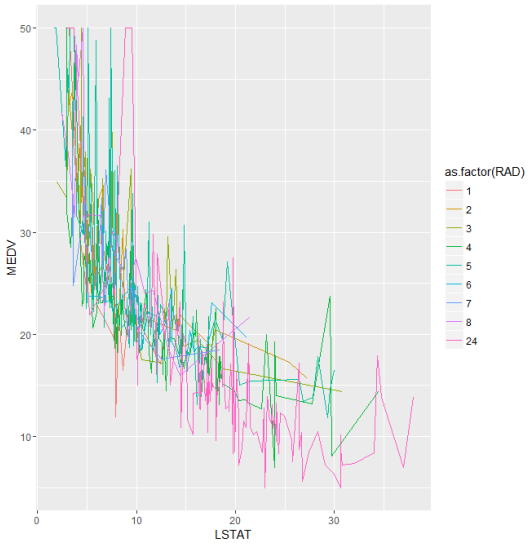
```
graphics.off()
x11()
library("ggplot2")
ggplot(data = mydata, aes(x = LSTAT, y = MEDV)) +
geom_line(alpha = .8, aes(colour= "red"))
```



Tracer des lignes par catégorie

Pour tracer les lignes par groupe, on ajoute l'option color=GroupeVariable dans le

```
graphics.off()
x11()
library(ggplot2)
ggplot(data = mydata, aes(x = LSTAT, y = MEDV,
color=as.factor(RAD))) + geom_line()
```



Chapitre 5 : Analyse des séries temporelles sous R : lissages, modélisations et prévisions

Ce chapitre a pour but de présenter les différentes étapes de traitement de séries temporelles sous R. Les principaux étapes discutées sont les méthodes de lissages (lissages linéaires et exponentiels), les méthodes de modélisations stochastiques (processus AR, MA, ARMA, ARIMA, SARIMA) et les prévisions (prévisions par les méthodes ad-hoc, prévisions par les méthodes stochastiques).

5.1. Préparation des données

5.1.1. Déclarartion de l'objet time series ts

A l'instar des objets classiques du langage R (vector, matrix, array, data frame, …), les séries temporelles sont des objets à part entière et bénéficient à ce titre d'un traitement spécifique.

Les objets *times series* se déclarent avec la fonction ts(). La syntaxe de base se présente comme suit:

myseries<- ts(data = , start =, end = , frequency =)

- **data**: représente l'objet contenant les données à mettre sous forme de séries temporelles
- **start**: indique la date de début de la série
- **end**: indique la date de fin de la série
- **frequency**: la périodicité de la série (annuelle, trimestrielle, mensuelle, hebdomadaire, journalière,…)

Exemple:

```
pbrent<- ts(data = mydata$pbrent, start = c(1990, 01), end = c(2009,06), frequency = 12)
```

On extrait la série pbrent de l'échantillon lsample et on la déclare comme objet ts.

Pour connaître la date de début et la date de fin sans avoir à ouvrir toute la table de données, utiliser les fonctions head() et tail() comme suit:

```
head(mydata, n=1L) # affiche la première observation de l'échantillon lsample
tail(mydata, n=1L) # affiche la dernière observation de l'échantillon lsample
```

On pourra alors aisément renseigner les paramètres start et end de la fonction ts()

NB: le paramètres « end » devient optionnel lorsque les paramètres « start » et « freqency » sont déjà spécifiés. R gère automatiquement la valeur « end » dans ce cas.

Exemple:

```
pbrent<- ts(data = mydata$pbrent, start = c(1990, 01), frequency = 12)
pbrent
```

5.1.2. Périodicité de la série et choix du paramètre de fréquence

Le paramètre « frequency » de la fonction ts() permet de définir la périodicité de la série temporelle.

La fréquence représente le nombre de valeurs pour une période complète d'observation (cycle d'observation). Les valeurs de frequency les plus courantes sont: 1 (cycle annuelle), 4 (trimestrielle), 12 (mensuelle), 7 (cycle hebdomadaire), 24 (cycle journalière).

Il existe plusieurs autres valeurs de fréquency. Les plus courantes sont résumées dans ce tableau:

Valeur frequency

	Minute	Heure	jour	semaine	Année
Données journalières				7	365
Données horaires			24	168	8766
Données demi-horaires			48	336	17532
Données par minute		60	1440	10080	525960
Données par seconde	60	3600	86400	604800	31557600

NB: Le choix de la valeur de frequency n'est pas figé. Compte tenu de la nature des données, l'utilisateur choisit la valeur de frequency qui lui semble pertinente pour son analyse. Mais il est commode de choisir la fréquence par rapport à une périodicité supérieure d'un cran par rapport à la fréquence d'observation. Par exemples, pour les données par seconde, choisir la minute comme le cycle et ainsi prendre 60 comme frequency. Pour les données horaires, choisir le jour comme le cycle et prendre 24 comme frequency. Et pour les données journalières, choisir la semaine comme cycle et considérer 7 comme valeur de frequency.

5.1.3. Quelques fonctions utiles pour le traitement de l'objet ts

L'objet ts étant défini, il est possible de lui appliquer un certain nombre de fonctions pour récupérer et renvoyer quelques informations utilisables dans le reste du programme de modélisation. Ci-dessous quelques-unes de ces fonctions:

- **cycle():** crée une variable contenant la position de chaque observation dans le cycle définit par frequency. Ex: cycle(pbrent)
- **start():** renvoie la date de fin de la série. Ex: start(pbrent)
- **end()**: renvoie la date de fin de la série. Ex: end(pbrent)
- **frequency()**: récupère le paramètre de fréquence (i.e nombre d'observations par cycle). Ex: frequency(pbrent)
- **time()**: crée une variable continue représentant le temps depuis la première observation de la série. Cette variable sert le plus souvent à capter la tendance de la série. Ex: time(pbrent)
- **window()**: permet de sélectionner une fenêtre de données dans la serie initiale. Ex: window(pbrent, start = c(2005,1), end = c(2005,12)) # extrait une partie de la série intiale.

5.1.4. Représentation graphique de la série

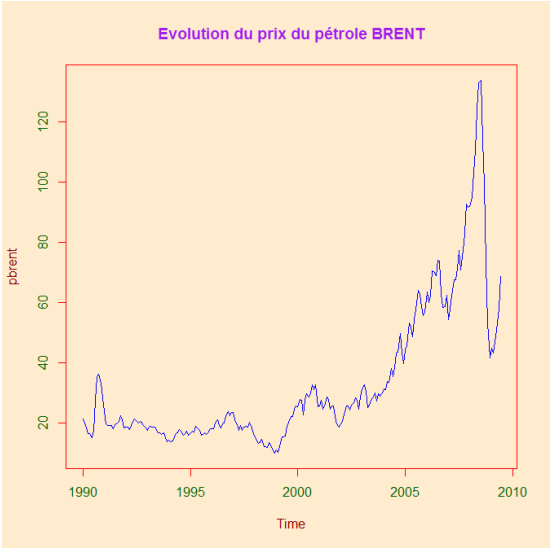
La représentation graphique d'une série temporelle se fait avec la fonction plot.ts(). Exemple:

```
plot.ts(pbrent) # effectue une représentation rapide de la série pbrent.
```

Mais il faut noter que la fonction plot.ts() a les mêmes propriétés que la fonction plot() classique car ayant les mêmes options et paramétrages généraux (Cf. Chapitre 4 Data Visualization).

Exemple:

```
graphics.off();x11()
par(bg ="blanchedalmond",col.axis= "darkgreen",
col.lab="darkred",col.main="purple",col.sub="black", fg="red")
plot.ts(pbrent, col="blue", main="Evolution du prix du pétrole
BRENT") # représentation graphique
```



La fonction plot.ts(pbrent) est, simplement, un condensé de la fonction plot(pbrent,time(pbrent)) où time() est la fonction qui renvoie une variable contenant le temps d'observation de la série.

5.1.5. Transformation de la série : log, lag et diff

De nombreux travaux de modélisation portent plutôt sur la série en log, en lag ou en diff que sur la série intiale observée.

- La série en **log** correspond au logarithme (à base 10) de la série intiale. Le calcul en log permet à la fois de diminuer les effets d'échelle mais aussi la forte dispersion de la série initiale.
- La série en **lag** représente la valeur retardée de série d'une certaine période (retard d'ordre 1, 2, …p). La variable en lag est généralement utilisée dans les modélisations autorégressives.
- La série en **diff** est la valeur différenciée de la série à un ordre d c'est-à-dire la d-ième différence entre la valeur actuelle de la série et la valeur retardée d'un ordre p. Cette transformation est couramment utilisée afin d'obtenir une série ayant les bonnes propriétés statistiques comme la stationnarité(nous reviendrons plus tard sur ces notions)

NB: Ces trois transformations ne sont pas mutuellement exclusives. On peut effectuer une transformation diff sur une série en lag obtenue après une transformation log.

Calcul du log

Le log de la série se calcule en appliquant simplement la fonction log() sur l'objet ts. Toutefois la série nouvellement obtenue doit aussi être déclaré comme un objet ts avec les mêmes paramètres que la série initiale. Exemple:

```
logpbrent<- ts(data = log(pbrent), start = c(1990, 01), end =
c(2009, 06), frequency = 12)
Logpbrent
```

Calcul du lag

Pour calculer le lag de la série pbrent, on va utiliser la fonction Lag() du package quantmod combinée avec la fonction as.vector() si on veut convertir l'objet d'abord en vecteur. Les étapes sont les suivantes:

```
#install.packages("quantmod")
library(quantmod)
lag1pbrent<-Lag(as.vector(pbrent), k=1) # calcul le lag
d'ordre 1 de pbrent après conversion en vector
lag1pbrent<- lag1pbrent[-1] # Exclut la première observation
qui contient NA (principe des lags).
lag1pbrent<- ts(data = lag1pbrent, start = c(1990, 02), end =
c(2009, 06), frequency = 12) # objet ts
lag1pbrent
```

NB: La transformation lag (sur un objet vecteur), nécessite de décaler le paramètre « start » d'une période car par principe la première observation de la série en lag a une valeur NA (première observation n'ayant pas d'antécédent). Le paramètre sera décalé de deux périodes dans le cas d'un lag d'ordre 2. Et ainsi de suite. Pensez à supprimer les lignes correspondantes à ces valeurs (dans certaines situations)

Calcul de la diff

Pour calculer la diff de la série pbrent, on va utiliser la fonction diff() directement disponible dans le package base. Les étapes sont les suivantes:

```
diff1pbrent<-diff(pbrent, lag = 1, differences = 1) # calcule
la différence première avec un lag d'ordre 1
diff1pbrent<- ts(data = diff1pbrent, start = c(1990, 02), end
= c(2009, 06), frequency = 12) # objet ts
diff1pbrent
```

NB: Tout comme la transformation lag, la transformation diff nécessite de décaler le paramètre « start » d'une période car la première observation de la série en diff a une valeur NA (observation n'ayant pas d'antécédent). Le paramètre sera décalé de deux périodes dans le cas d'un diff d'ordre 2(ou avec un lag d'ordre 2). Et ainsi de suite.

Calcul de la valeur avancée : le next

En complément du lag d'une série, on est souvent amené à calculer le next d'une série. Pour cela, on adopte la démarche suivante :

```
#install.packages("quantmod")
library(quantmod)
```

```
mydata <- mydata[order(mydata2$Date,decreasing = FALSE),] # Tri par ordre croissant de la date
mydata$next1P_Brent=Next(mydata2$P_Brent, k=1) # application de la fonction lag() avec k=1 (lag d'ordre 1)
```

5.2. Les méthodes « ad-hoc » de prévision

5.2.1. Présentation générale

Les méthodes « ad-hoc » sont des méthodes basiques permettant de réaliser des prévisions rapides sur les valeurs futures d'une série à partir simples hypothèses.

Les principales méthodes « ad-hoc » de prévision sont:

- **la méthode des moyennes (Means method**): la valeur prévue à tout instant est égale à la moyenne des valeurs passées.
- **la méthode naïve (Naive method):** la valeur prévue à tout instant est égale à la valeur observée à la période précédente.
- **la méthode naïve saisonnière (Seasonal Naive method):** la valeur prévue à tout instant est égale à la valeur observée la saison passée à la même date.
- **la méthode de dérive (Drift method):** la valeur prévue à tout instant est égale à la valeur observée à la période précédente à laquelle on ajoute la variation moyenne observée dans le passé.

5.2.2. La méthode des moyennes (Means method)

Principe

La Means method consiste à prévoir la valeur future d'une série à partir la moyenne de ses valeurs passées.

Cette méthode convient, en particulier aux séries relativement stables avec une moyenne constante au cours du temps

Application

Pour mettre en œuvre la méthode des moyennes, considérons la série ihpc représentant l'indice des prix à la consommation en France entre janvier 1990 et Août 2016.

ihpc<- ts(data = mydata$ihpc, start = c(1990, 01), end = c(2016,08), frequency = 12)

Effectuons la prévision sur les 20 prochains mois de l'indice des prix à la consommation en utilisant la méthode des moyennes. On a:

```
library(forecast)
```

```
meanfpred<-meanf(ihpc,h=20, level=95) # applique la mean
method avec la fonction meanf
summary(meanfpred)
graphics.off(); x11()
plot(ihpc,xlim=c(1990,2020), lwd=1.5) # représente la série
observée
lines(meanfpred$fitted, col="red", lwd=1.5) # représente la
série ajustée
lines(meanfpred$mean, col="blue", lwd=1.5) # Représente les
valeurs prévues
legend(x=2011,y=72,legend=c("Série observée","Série
ajustée","Série prévue"),fill=c("black", "red", "blue"))
```

5.2.3. La méthode naïve (Naive method)

Principe

La méthode naïve consiste à prévoir la valeur future d'une série à partir de la valeur observée à la période précédente. Il s'agit donc d'une simple transformation lag d'ordre 1 de la série.

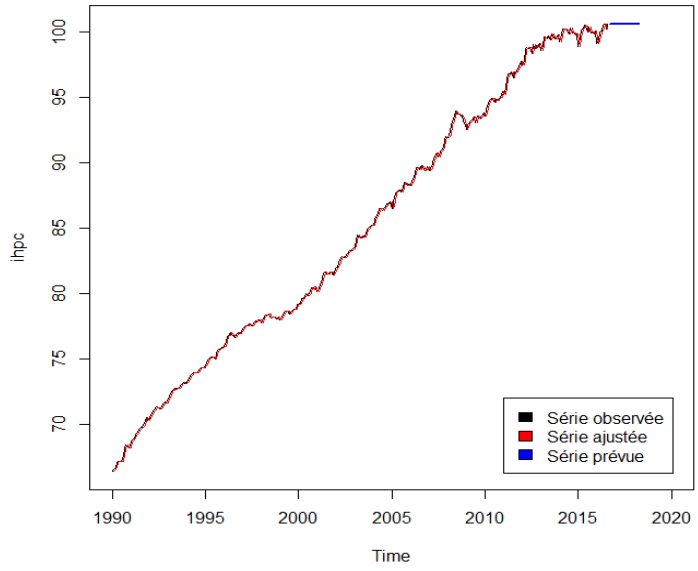
Cette méthode convient, en particulier, aux actifs financiers dont les prix sont très volatiles à moyen-long terme. On privilégie alors une prévision à partir des valeurs plus récentes.

Application

La méthode naïve de prévision peut s'effectuer avec la fonction naive() du package forecast.

Exemple: Considérons toujours la série ihpc et effectuons une prévision sur les 20 prochains mois.

```
naivepred<-naive(ihpc, h=20, level=95, fan=FALSE, lambda=NULL)
summary(naivepred)
graphics.off(); windows()
plot(ihpc,xlim=c(1990,2020), lwd=1.5)
lines(naivepred$fitted, col="red", lwd=1.5)
lines(naivepred$mean, col="blue", lwd=2.5)
legend(x=2011,y=72,legend=c("Série observée","Série
ajustée","Série prévue"),fill=c("black", "red", "blue"))
```



5.2.4. La méthode naïve saisonnière (Seasonal naive method)

Principe

La méthode naïve saisonnière consiste à prendre la valeur observée à la même période cette saison comme valeur future pour une date correspondante la saison suivante.

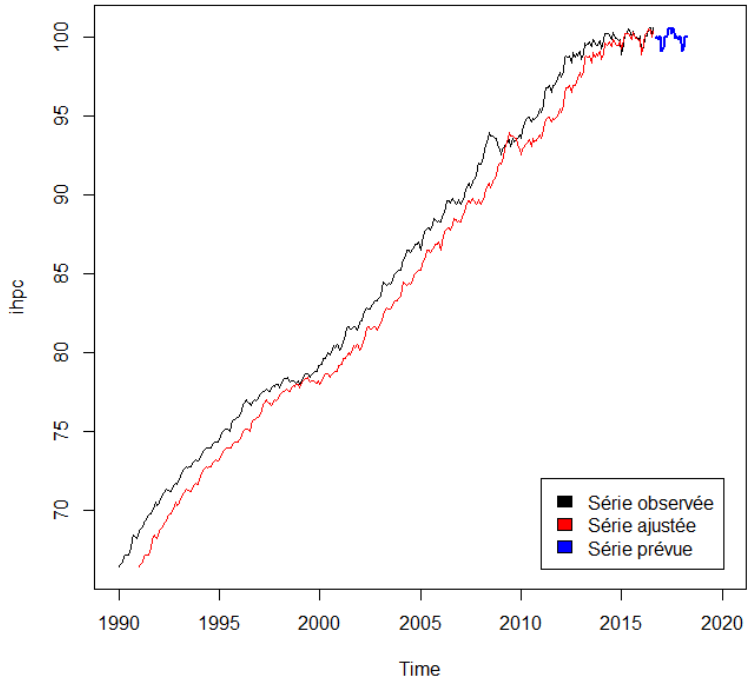
Cette méthode convient mieux aux séries stables avec des saisonnalités régulières.

Application

La méthode naïve saisonnière peut s'effectuer avec la fonction snaive() du package forecast.

Exemple: Effectuons une prévision sur les 20 prochains mois en considérant la série ihpc.

```
snaivepred<-snaive(ihpc, h=20, level=95, fan=FALSE, lambda=NULL)
summary(snaivepred)
graphics.off(); windows()
plot(ihpc, xlim=c(1990,2020), lwd=1.5)
lines(snaivepred$fitted, col="red", lwd=1.5)
lines(snaivepred$mean, col="blue", lwd=2)
legend(x=2011,y=72,legend=c("Série observée","Série ajustée","Série prévue"),fill=c("black", "red", "blue"))
```



5.2.5. La méthode de dérive (Drift method)

Principe

Dans la Drift method, la valeur prévue à un instant donné est égale à la valeur observée à la période précédente plus la variation moyenne observée dans le passé.

La méthode Drift est une forme améliorée de la méthode naïve avec un facteur d'ajustement prenant en compte toute l'historique de la série.

Cette méthode est adoptée en particulier pour les séries suivant une marche aléatoire (Nous reviendrons sur la notion de marche aléatoire plus tard).

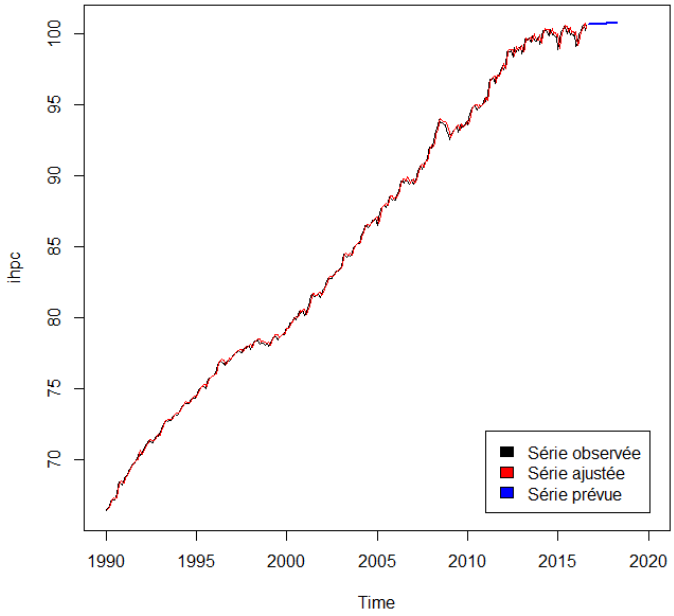
Application

La méthode Drift peut se mettre en œuvre avec la fonction rwf() du package forecast.

Exemple: Effectuons une prévision sur les 20 prochains mois en considérant la série ihpc.

```
driftpred<-rwf(ihpc, h=20, drift=TRUE, level=95, fan=FALSE,
lambda=NULL, biasadj=FALSE)
summary(driftpred)
graphics.off(); windows()
plot(ihpc, xlim=c(1990,2020), lwd=1.5)
lines(driftpred$fitted, col="red", lwd=1.5)
lines(driftpred$mean, col="blue", lwd=2)
```

```
legend(x=2011,y=72,legend=c("Série observée","Série ajustée","Série prévue"),fill=c("black", "red", "blue"))
```



5.3. Les méthodes de lissages linéaires

5.3.1. Généralités sur les méthodes de lissage

Le lissage d'une série consiste éliminer les saisonnalités et les irrégularités dans l'optique d'une prévision.

On distingue trois grandes familles de méthodes de lissage: les méthodes de lissage linéaire, les méthodes de lissage exponentiel et les méthodes stochastiques.

- Les méthodes linéaires sont basées sur l'estimation de modèles de régressions linéaires ou l'utilisation des moyennes mobiles à pondérations fixes
- Les méthodes exponentielles sont une généralisation de la méthode linéaire de moyennes mobiles avec des pondérations à croissance exponentielle.
- Quant aux méthodes stochastiques, elles sont basées sur l'estimation des modèles autorégressifs (AR), des modèles de moyennes mobiles stochastiques (MA) ou une combinaison des deux types de modèles dans des variantes plus élargies (ARMA, ARIMA, SARIMA, ARCH, GARCH, etc...)

5.3.2. Lissage par régression linéaire

5.3.2.1. Cas d'une série avec tendance linéaire (sans saisonnalités)

Exemple: Série de l'indice de prix ihpc:

```
library(xlsx)
mydata<- read.xlsx("dataworkbook.xlsx", sheetName= "ihpcdata"
,startRow=1, endRow=321, colIndex=c(1:2),
header=TRUE,as.data.frame= TRUE)
mydata <- mydata[order(mydata$Date, decreasing = FALSE),]
ihpc<- ts(data = mydata$ihpc, start = c(1990, 01), end =
c(2016,08), frequency = 12)
graphics.off(); x11()
plot.ts(ihpc)
```

Construction de l'échantillon d'apprentissage

```
lsample<-mydata[1:ceiling(0.75*nrow(mydata)),] # Echantillon
d'apprentissage (première 75% d'obs)
ihpc<- ts(data = lsample$ihpc, start = c(1990, 01), end =
c(2009,12), frequency = 12)
trend<-as.numeric(time(ihpc)) # Générer la variable temps
(pour capter la tendance)
lsample<-as.data.frame(cbind(trend, ihpc)) # Echantillon
d'apprentissage
rm(trend, ihpc) # supprime les objets libres trend et ihpc
```

Construction de l'échantillon de validation

```
vsample<-mydata[(ceiling(0.75*nrow(mydata))+1):nrow(mydata),]
# Sélection de 25% restants
ihpc<- ts(data = vsample$ihpc, start = c(2010, 01), end =
c(2016,08), frequency = 12)
trend<-as.numeric(time(ihpc))# Générer la variable temps(pour
capter la tendance)
vsample<-as.data.frame(cbind(trend, ihpc))
rm(trend, ihpc) # supprime les objets libres trend et ihpc
```

Estimation du modèle sur l'échantillon d'apprentissage

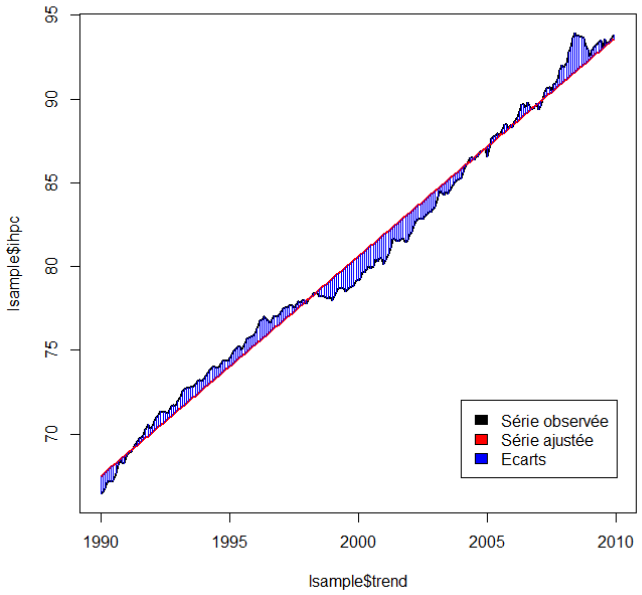
```
reg1 = lm(ihpc ~ poly(trend,1), data=lsample) # Régression
linéaire de X par t.
summary(reg1) # Résumé des résultats de la régression.
```

NB: L'estimation du modèle linéaire ne nécessite pas que l'objet soit déclaré en ts

Qualité de l'estimation du modèle sur l'échantillon d'apprentissage

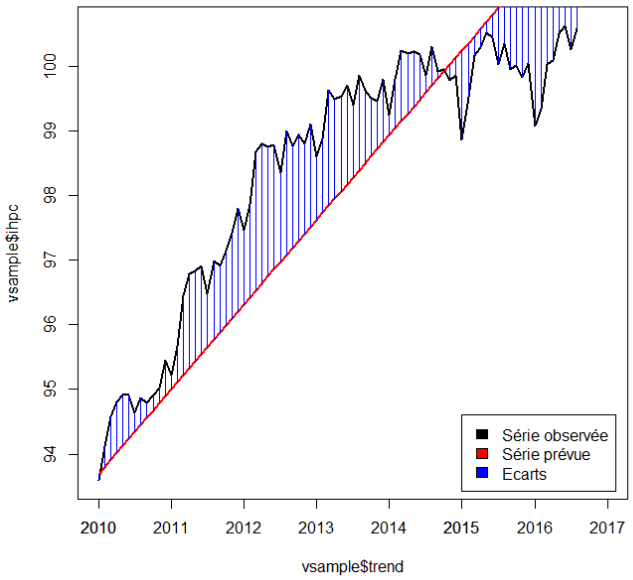
```
plot(lsample$trend,lsample$ihpc ,type='l', xlim=c(1990,2010),
lwd=2)
axis(1,at=seq(from=1990,to=2010,by=5),labels=as.character(seq(
from=1990,to=2010,by=5))) .
lines(lsample$trend, reg1$fit, col="red", lwd=2) # Graphe de
la droite ajustée
```

```
segments(lsample$trend, reg1$fit, lsample$trend, lsample$ihpc,
col = "blue", lwd=1.5) # Résidus.
legend(x=2004,y=72,legend=c("Série observée","Série
ajustée","Ecarts"),fill=c("black", "red", "blue"))
```



Prévision et validation du modèle (échantillon de validation)

```
ihpc_prev<-predict(reg1, newdata=vsample) # Prévision sur
l'échantillon de validation
ihpc_prev<-as.vector(ihpc_prev) # convertir en vecteur
plot(vsample$trend,vsample$ihpc, type="l", xlim=c(2010,2017),
lwd=2) # valeurs observées
axis(1,at=seq(from=2010,to=2017,by=5),labels=as.character(seq(
from=2010,to=2017,by=5)))
lines( vsample$trend,ihpc_prev, col="red", lwd=2) # valeurs
prévues
segments(vsample$trend, vsample$ihpc, vsample$trend,
ihpc_prev, col = "blue", lwd=1.5) # résidus.
legend(x=2015,y=94.6,legend=c ("Série observée","Série
prévue","Ecarts"), fill=c("black", "red", "blue"))
```



5.3.2.2. Cas d'une série avec tendance linéaire (avec saisonnalités) : l'approche de BUYS-BALLOT

L'approche de BUYS-BALLOT consiste à introduire des variables indicatrices correspondant à chaque saison définit par le cycle d'observation. Pour les données trimestrielles, on intègre 4 variables indicatrices. Et pour les données mensuelles, on intègre 12 variables indicatrices.

Le modèle doit alors être estimé (sans constante) avec ces variables indicatrices.

Exemple: la série econso: consommation d'énergie aux Etats-Unis entre Mai 1990 et Mai 2008

```
mydata<-        read.xlsx("dataWorkbook.xlsx",        sheetName
="energyUsa",startRow=1,endRow=218,colIndex=c(1:2),
header=TRUE,as.data.frame= TRUE) # importation des données
mydata$mois<- as.Date(mydata$mois, format = "%y%B")
mydata <- mydata[order(mydata$mois, decreasing = FALSE),]
econso<- ts(data = mydata$econso, start = c(1990, 05), end =
c(2008,05), frequency = 12)
graphics.off(); x11(); plot.ts(econso)
```

Construction de l'échantillon d'apprentissage

```
lsample<-mydata[1:ceiling(0.75*nrow(mydata)),]   #   Echantillon
d'apprentissage (première 75% d'obs)
econso<- ts(data = lsample$econso,start = c(1990, 05), end =
c(2003,11), frequency = 12)
trend<-as.numeric(time(econso)) # Générer la variable temps
```

```
frq<-frequency(econso)   #   extraire   la   valeur   de   la
fréquence(but: créer des variables indicatrices saisons)
saison  <-rep(1:frq,ceiling(length(econso)/frq))#  Générer  des
valeurs pour chaque saison (fréquence). ici: 1 à 12 repetés  x
fois
saison  <-saison[1:length(econso)]  #  ajuste  la  dimension  de
saison à la dimension de econso
saison  <-as.factor(saison)  #  transformer  saison  en  variable
factor
lsample<-as.data.frame(cbind(trend,saison,econso))            #
reconstitue lsample
rm(frq, trend,saison,econso) # supprime les objets libres frq,
trend,saison et econso
```

Construction de l'échantillon de validation

```
vsample<-mydata[(ceiling(0.75*nrow(mydata))+1):nrow(mydata),]
# dernières 25% d'obs
econso<- ts(data = vsample$econso,start = c(2003, 12), end =
c(2008,05), frequency = 12)
trend<-as.numeric(time(econso)) # Générer la variable temps
frq<-frequency(econso) # extraire la valeur de la fréquence
saison  <-rep(1:frq,ceiling(length(econso)/frq))#  Générer  des
valeurs pour chaque saison (fréquence).
saison  <-saison[1:length(econso)]  #  ajuste  la  dimension  de
saison à la dimension de econso
saison  <-as.factor(saison)  #  transformer  saison  en  variable
factor
vsample<-as.data.frame(cbind(trend,saison,econso))            #
reconstitue vsample
rm(frq, trend,saison,econso) # supprime les objets libres frq,
trend,saison et econso
```

Estimation du modèle sur l'échantillon d'apprentissage

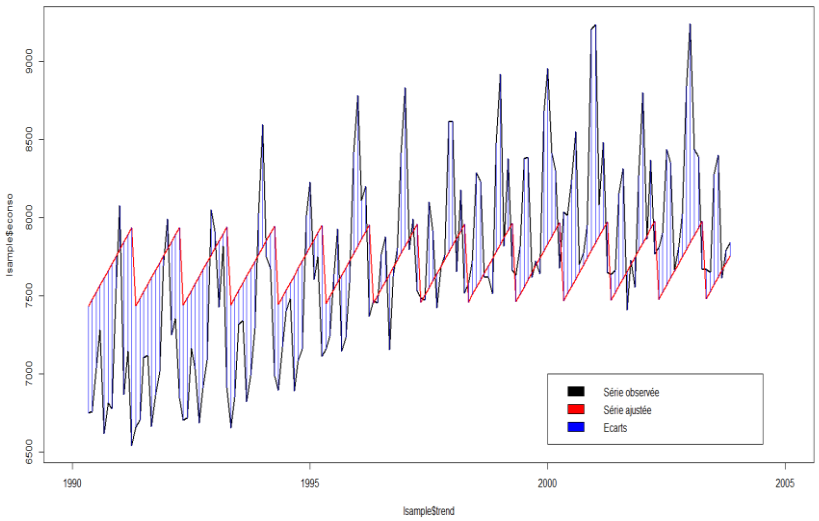
```
desaireg<-lm(econso~0+trend+saison, data=lsample)
summary(desaireg)
```

NB: Le modèle est estimé sans la constante pour pouvoir identifier les coefficients saisonniers.

Qualité de l'estimation du modèle sur l'échantillon d'apprentissage

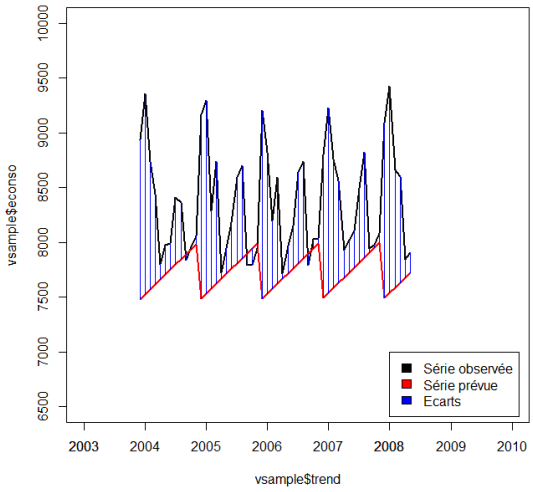
```
graphics.off(); x11()
plot(lsample$trend,lsample$econso                    ,type='l',
xlim=c(1990,2005), lwd=2)
```

```
axis(1,at=seq(from=1990,to=2005,by=5),labels=as.character(seq(
from=1990,to=2005,by=5))) .
lines(lsample$trend, desaireg$fit, col="red", lwd=2) # droite
ajustée
segments(lsample$trend, desaireg$fit, lsample$trend,
lsample$econso, col = "blue", lwd=1.5) # Visualisation des
résidus.
legend(x=2000,y=7000,legend=c("Série observée","Série
ajustée","Ecarts"),fill=c("black", "red", "blue"))
```



Prévision et validation du modèle (échantillon de validation)

```
econso_prev<-predict(desaireg, newdata=vsample) # Prévision
sur l'échantillon de validation
econso_prev<-as.vector(econso_prev) # convertir en vecteur
graphics.off(); x11()
plot(vsample$trend,vsample$econso, type="l",
xlim=c(2003,2010), ylim=c(6500,10000),lwd=2) # observées
axis(1,at=seq(from=2003,to=2010,by=5),labels=as.character(seq(
from=2003,to=2010,by=5)))
lines( vsample$trend,econso_prev, col="red", lwd=2) # valeurs
prévues
segments(vsample$trend, vsample$econso, vsample$trend,
econso_prev, col = "blue", lwd=1.5) # résidus.
legend(x=2008,y=7000,legend=c("Série observée","Série
prévue","Ecarts"),fill=c("black", "red", "blue"))
```



5.3.2.3. Remarques générales sur le lissage par régression linéaire

Les méthodes de lissage par régression que nous venons de présenter supposent que la tendance de la série soit linéaire. Il suffit alors simple de faire la régression avec la variable « temps »

Mais lorsque la tendance n'est pas linéaire, il faut introduire dans la régression la forme polynomiale du temps (c-à-d variable temps au carré, au cube, etc…).

Toutefois lorsque le degré du polynôme devient trop élevé, il faut alors se tourner vers les méthodes non paramétriques de régressions. Il s'agit notamment de la méthode LOWESS, de la méthode LOESS ou de la méthode KERNEL. Nous ne présenterons pas d'exemples pratiques de ces méthodes ici. Consulter la documentation R sur les fonction **lowess(), loess()** et **ksmooth()**.

5.3.3. Lissage par les moyennes mobiles

On distingue deux cas de lissage par les moyennes mobiles: le filtre-lissage et le lissage par décomposition.

Le filtre-lissage consiste à extraire la tendance d'une série tout en éliminant les saisonnalités et les irrégularités.

Quant au lissage par décomposition, il consiste à fractionner la série en différentes composantes que sont généralement la tendance, les saisonnalités et les irrégularités.

A la différence d'un filtre, la décomposition permet d'obtenir (sous forme de série) chaque composante de la série initiale. Cette méthode permet donc une désaisonnalisation de la série.

5.3.3.1. Filtre-lissage: la fonction filter()

Le filtre-lissage se met en œuvre avec la fonction filter() avec l'option "convolution"

Cette méthode de filtrage est plus adaptée aux séries avec tendance ayant des fortes irrégularités mais sans saisonnalités marquées.

Exemple: Série econso , consommation d'énergie aux USA entre Mai 1990 et Mai 2008

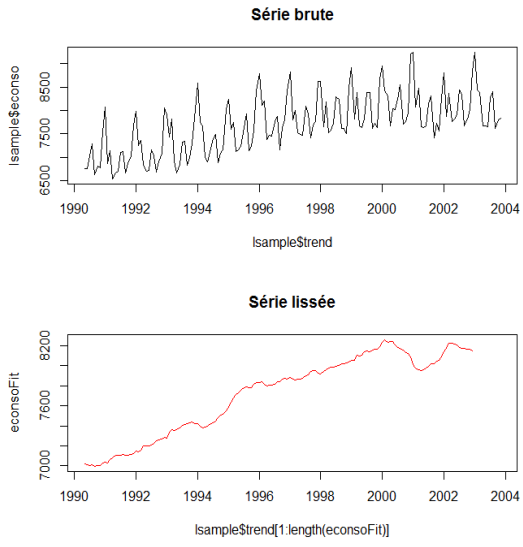
En guise d'application de la méthode de lissage, reprenons la série econso dont l'échantillon d'apprentissage et de validation ont été définies précédemment (lsample et vsample).

Lissage sur l'échantillon d'apprentissage:

```
econsoFit<-filter(x=lsample$econso, filter=rep(1/12, 12), method = "convolution", side = 2) # moyenne mobile de pondération=1/12
econsoFit<-econsoFit[!is.na(econsoFit)] # exclusion des valeurs NA
```

Visualisation de la série lissée

```
graphics.off(); x11() ;par(mfrow=c(2,1))
plot(lsample$trend,lsample$econso, type="l", xlim=c(min(lsample$trend), max(lsample$trend)), lwd=1.8, main="Série brute") # valeurs observées
plot(lsample$trend[1:length(econsoFit)],econsoFit, type="l",col="red", xlim=c(min(lsample$trend), max(lsample$trend)), lwd=1.8 , main="Série lissée") # valeurs lissées
```



Prévision avec la série lissée par filtrage

La méthode de filtre n'a pas vocation à faire une prévision de la série. Elle vise simplement à débarrasser la série des valeurs irrégulières pour obtenir une valeur lissée. Cette valeur lissée peut ensuite être utilisée pour effectuer des prévisions en utilisant les méthodes appropriées. On peut utiliser par exemples les méthodes ad-hoc de prévision: la méthode des moyennes, la méthode naive ou sa variante saisonnière ou encore la méthode de dérive (Drift).

Bien entendu, on peut utiliser les méthodes de régression linéaire ou les méthodes stochastiques (ARMA, ARIMA) pour élaborer des prévisions sur la série lissée (Nous présentons ces méthodes dans les dernières sections).

En guise d'application, utilisons la méthode Drift pour réaliser la prévision de la série lissée par filter()

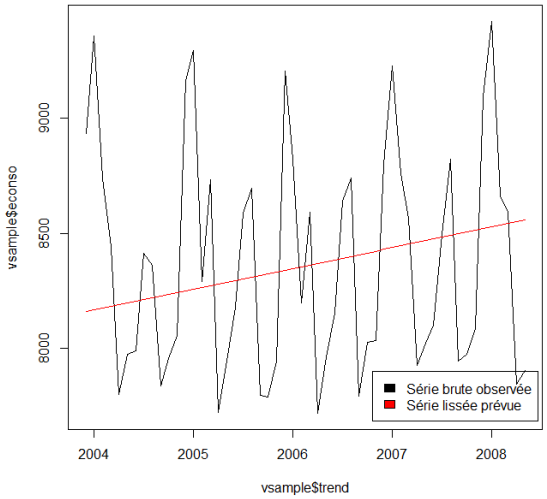
Effectuons la prévision de la série lissée « econsoFit » en utilisant la fonction rwf() du package forecast.

```
library(forecast)
rwfPred<-rwf(econsoFit, h=nrow(vsample), drift=TRUE, level=95, fan=FALSE, lambda=NULL, biasadj=FALSE) econsoPred<-rwfPred$mean
```

NB: La prévision est effectuée sur un horizon h égal à la dimension de vsample (pour la comparaison avec les valeurs observées)

Visualisation:

```
graphics.off(); x11() ;
plot(vsample$trend,vsample$econso, type="l", xlim=c(min(vsample$trend), max(vsample$trend)), lwd=1.8)
lines(vsample$trend[1:length(econsoPred)],econsoPred,col="red",xlim=c(min(vsample$trend), max(vsample$trend)), lwd=1.8 ) # valeurs lissées prévues
legend(x=2006.8,y=7900,legend=c("Série brute observée","Série lissée prévue"),fill=c("black", "red"))
```



5.3.3.2. Lissage par décomposition: la fonction decompose()

Contrairement à la méthode de filtre, la méthode de décomposition permet d'extraire toutes les composantes de la série sous formes de nouvelles séries. Ces composantes sont: la tendance, la saisonnalité et les irrégularités.

Cette méthode est plus adaptée à la séries présentant à la fois des tendances, des saisonnalités et des irrégularités fortement marquées.

Pour réaliser la décomposition sous R, on utilise la fonction decompose()

Pour rappel: la saisonnalité peut être additive, multiplicative ou mixte. Avec la fonction decompose() la nature de la saisonnalité est spécifiée avec le paramètre type qui peut être « additive » ou « multiplicative ».

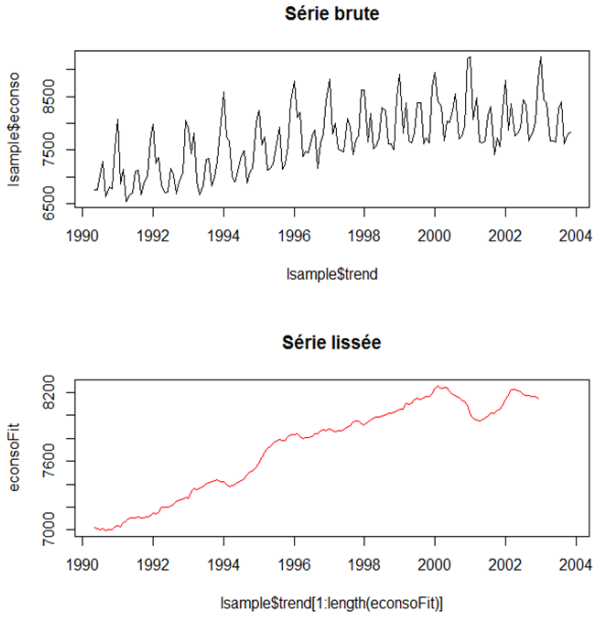
Exemple: Série econso: consommation d'énergie aux USA entre Mai 1990 et Mai 2008

Lissage sur l'échantillon d'apprentissage:

```
econso<- ts(data = lsample$econso,start = c(1990, 05), end = c(2003,11), frequency = 12) # Création de l'objet ts( obligatoire dans le cas de la fonction decompose)
decomp1<-decompose(econso,type="additive", filter = rep(1/12,12)) # Décompostion modèle additif
decomp1$trend # composante tendance
decomp1$seasonal # composante saisonnalités
decomp1$random # composante irrégularités (résidus aléatoires)
econsoFit<-decomp1$trend[!is.na(decomp1$trend)] # Récupération séries tendance (exclusion de NA)
```

Visualisation de la série lissée

```
graphics.off(); x11() ;par(mfrow=c(2,1))
plot(lsample$trend,lsample$econso,                    type="l",
xlim=c(min(lsample$trend),    max(lsample$trend)),    lwd=1.8,
main="Série brute") # représentation des valeurs observées
plot(lsample$trend[1:length(econsoFit)],econsoFit,
type="l",col="red",                  xlim=c(min(lsample$trend),
max(lsample$trend)),   lwd=1.8   ,   main="Série   lissée")   #
représentation des valeurs lissées
```



Prévision avec la série lissée par décomposiation

Tout comme la fonction filter(), la fonction decompose() ne vise pas à effectuer des prévisions à la suite du lissage. Des méthodes spécifiques de prévision doivent être utilisées pour réaliser ces prévisions. Voir l'exemple dans le cas de la fonction filter() où on utilise la méthode drift sur la série des tendances extraites.

Toutefois, la particularité de la fonction decompose() est qu'elle permet d'obtenir les coefficients saisonniers. Dès lors, après avoir effectué la prévision en utilisant la série des tendances, il faut ensuite ajouter ces coefficients saisonniers aux valeurs prévues afin d'obtenir la prévision finale.

NB: Pour les données trimestrielles, on obtient quatre coefficients saisonniers distincts et pour les données mensuelles on obtient douze coefficients distincts. Ces valeurs doivent être ajoutées aux valeurs de tendance appartenant à la même saison. Pour afficher les coefficients saisonniers on fait:

```
unique(decomp1$seasonal)
```

5.4. Les méthodes de lissage exponentiel

5.4.1. Généralités sur les méthodes de lissages exponentiels

Les méthodes de lissages exponentiels sont une généralisation de la méthode de décomposition par les moyennes mobiles où le poids de chaque observation décroît de manière exponentielle quand elle s'éloigne dans le passé.

On distingue trois grandes familles de méthodes de lissages exponentiels:

- Le lissage exponentiel simple: dans lequel la tendance est ajustée à une constante
- Le lissage exponentiel double: dans lequel la tendance est ajustée à une droite locale
- Et les méthodes de lissage Holt-Winters qui ajustement la tendance à une droite locale tout en contrôlant les saisonnalités et les irrégularités de la série.

Sous R, les méthodes de lissage exponentiels se mettent en œuvre la fonction HoltWinters().

On peut aussi utiliser les fonctions disponibles dans le package « forecast ». Mais ces fonctions ne sont pas présentées ici.

5.4.2. Le lissage exponentiel simple LES

Le LES sert à lisser une série ayant une tendance localement constante en utilisant la méthode des moyennes mobiles attribuant des pondérations de plus en plus faibles aux observations les plus éloignées.

NB: Il arrive aussi d'attribuer des pondérations plus faibles aux observations récentes et des pondérables plus élevées aux valeurs éloignées. Tout dépend, en fait du choix de paramètre de lissage (Cf. théorie).

Exemple: Série econso, consommation d'énergie aux USA entre Mai 1990 et Mai 2008

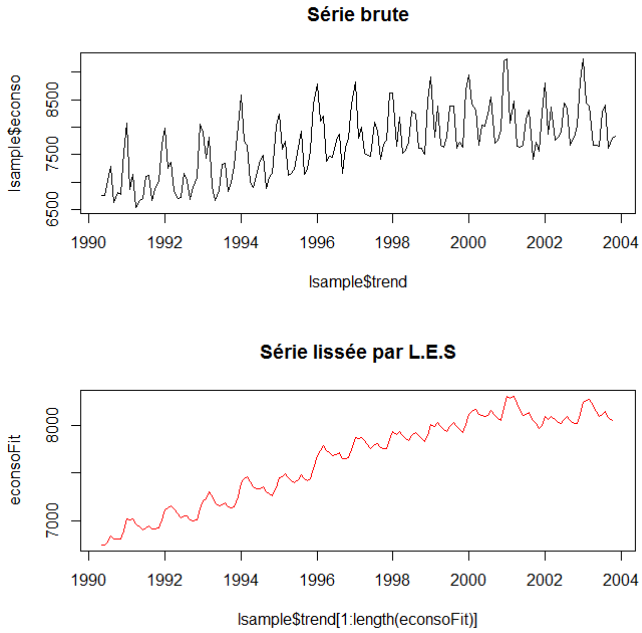
Considérons la série econso préalablement scindé en lsample et vsample.

Les étapes de mise en œuvre du LES sont présentées ci-dessous:

```
les <-
HoltWinters(lsample$econso,alpha=NULL,beta=FALSE,gamma=FALSE)
# Lissage avec détermination automatique des valeurs
optimales des paramètres de lissage
print(les)
econsoFit<-les$fitted[,1] # Récupère la série lissée
```

Visualisation de la série lissée:

```
graphics.off(); x11() ;par(mfrow=c(2,1))
plot(lsample$trend,lsample$econso,                    type="l",
xlim=c(min(lsample$trend),    max(lsample$trend)),    lwd=1.8,
main="Série brute") # valeurs observées
plot(lsample$trend[1:length(econsoFit)],econsoFit,
type="l",col="red",                  xlim=c(min(lsample$trend),
max(lsample$trend)), lwd=1.8 , main="Série lissée par L.E.S")
#  valeurs lissées
```



Prévision de la série lissée

Contrairement aux fonctions filter() et decompose(), il existe une fonction predict() associée à la fonction HoltWinters() qui permet de réaliser directement les prévisions sur un horizon bien défini.

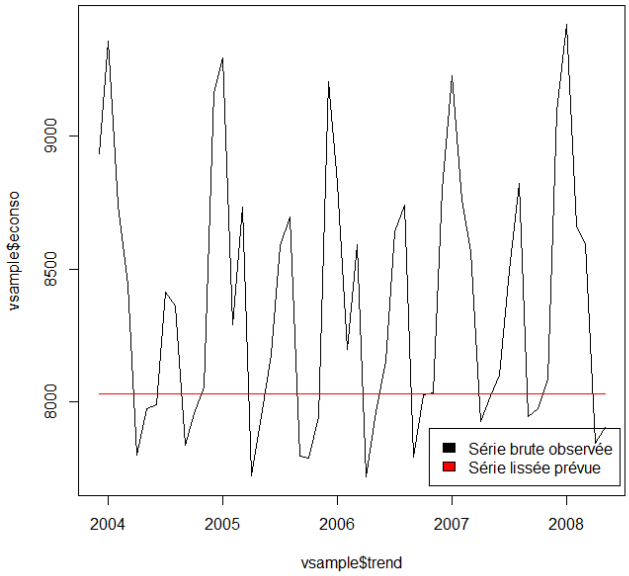
L'exemple ci-dessous réalise une prévision sur un horizon égal à la dimension de l'échantillon de validation vsample afin de faire des comparaisons avec les valeurs brutes observées:

```
econsoPred<-
predict(les,n.ahead=nrow(vsample),prediction.interval=FALSE)  #
Effectue la prévision
```

Visualisation de la série prévue :

```
graphics.off(); x11() ;
plot(vsample$trend,vsample$econso,                    type="l",
xlim=c(min(vsample$trend),   max(vsample$trend)),   lwd=1.8)   #
valeurs observées
```

```
lines(vsample$trend[1:length(econsoPred)],econsoPred,col="red"
,xlim=c(min(vsample$trend), max(vsample$trend)), lwd=1.8 ) #
valeurs lissées prévues
legend(x=2006.8,y=7900,legend=c("Série brute observée","Série
lissée prévue"),fill=c("black", "red"))
```



5.4.3. Le lissage exponentiel double LED

Le LED est utilisé lorsque la tendance de la série peut être localement ajustée par une droite. Tout comme dans le LES, le LED réalise un lissage par les moyennes mobiles attribuant des pondérations à croissance (décroissance) exponentielle. Sa mise en œuvre nécessite également le choix des paramètres de lissage.

Exemple: Série econso, consommation d'énergie aux USA entre Mai 1990 et Mai 2008

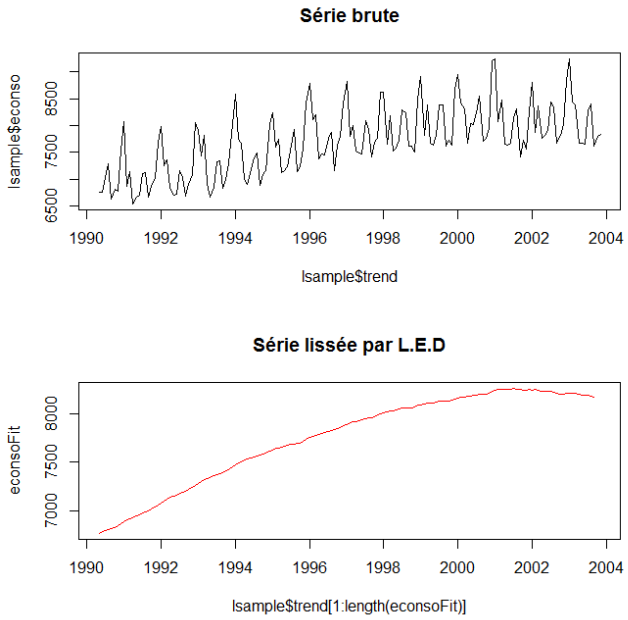
Les étapes de mise en œuvre du LED sont présentées ci-dessous:

```
led <-
HoltWinters(lsample$econso,alpha=NULL,beta=NULL,gamma=FALSE) #
Lissage avec détermination automatique des paramètres optimaux
print(led)
econsoFit<-led$fitted[,1] # la série lissée
```

Visualisation de la série lissée:

```
graphics.off(); x11() ;par(mfrow=c(2,1))
plot(lsample$trend,lsample$econso, type="l",
xlim=c(min(lsample$trend), max(lsample$trend)), lwd=1.8,
main="Série brute") # valeurs observées
```

```
plot(lsample$trend[1:length(econsoFit)],econsoFit,
type="l",col="red",                    xlim=c(min(lsample$trend),
max(lsample$trend)),  lwd=1.8  ,  main="Série  lissée  par  L.E.D")
#  valeurs lissées
```



Prévision de la série lissée

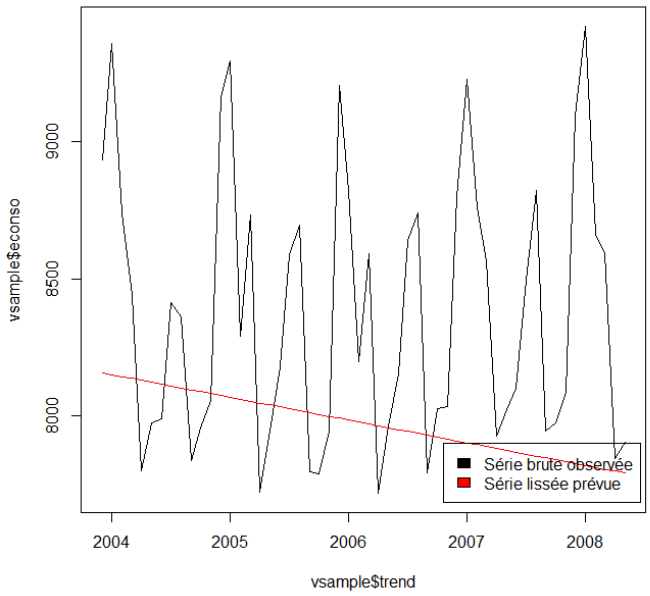
La prévision à la suite d'un LED se fait comme la prévision après un LES en utilisant la fonction predict() avec l'horizon de prévision fixé.

Dans l'exemple suivant, on réalise une prévision sur un horizon égal à la dimension de l'échantillon de validation vsample afin de comparer les valeurs prévues avec les valeurs brutes observées:

```
econsoPred<-
predict(led,n.ahead=nrow(vsample),prediction.interval=FALSE)  #
Effectue la prévision
```

Visualisation de la série prévue :

```
graphics.off(); x11()
plot(vsample$trend,vsample$econso,                    type="l",
xlim=c(min(vsample$trend),   max(vsample$trend)),   lwd=1.8)   #
valeurs observées
lines(vsample$trend[1:length(econsoPred)],econsoPred,col="red"
,xlim=c(min(vsample$trend),  max(vsample$trend)),  lwd=1.8   )  #
valeurs lissées prévues
legend(x=2006.8,y=7900,legend=c("Série  brute  observée","Série
lissée prévue"),fill=c("black", "red"))
```



5.4.4. Le lissage exponentiel Holt-Winters HW

La méthode de lissage Holt-Winters est utilisée lorsque la série présente non seulement une tendance (localement ajustable par une droite) mais aussi une saisonnalité (de type additive ou multiplicative). Il existe toutefois une version du HW sur les séries sans saisonnalité. Il s'agit dans ce cas de l'estimateur Holt qui est très proche de la méthode LED.

Exemples d'application de la méthode HW:

Cas d'une série avec saisonnalité additive: Série econso, consommation d'énergie aux USA entre Mai 1990 et Mai 2008

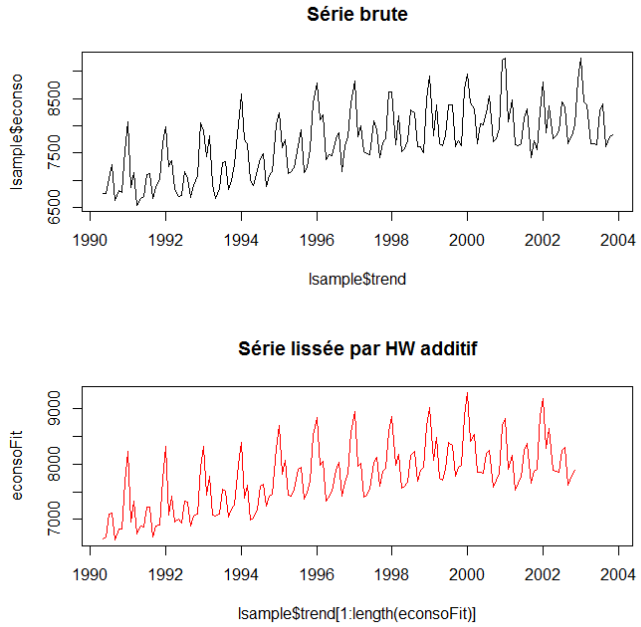
```
econso<- ts(data = lsample$econso,start = c(1990, 05), end = c(2003,11), frequency = 12)
hwAd <- HoltWinters(econso,alpha=NULL,beta=NULL,gamma=NULL, seasonal='add') # détermination automatique des paramètres optimaux.
print(hwAd)
```

Pour une série avec saisonnalité multiplicative, indiquer simplement: seasonal='multi'

Visualisation de la série lissée:

```
graphics.off(); x11() ;par(mfrow=c(2,1))
plot(lsample$trend,lsample$econso, type="l", xlim=c(min(lsample$trend), max(lsample$trend)), lwd=1.8, main="Série brute") # valeurs observées
plot(lsample$trend[1:length(econsoFit)],econsoFit, type="l",col="red", xlim=c(min(lsample$trend),
```

```
max(lsample$trend)), lwd=1.8 , main="Série lissée par HW
additif") # valeurs lissées
```



Prévision de la série lissée

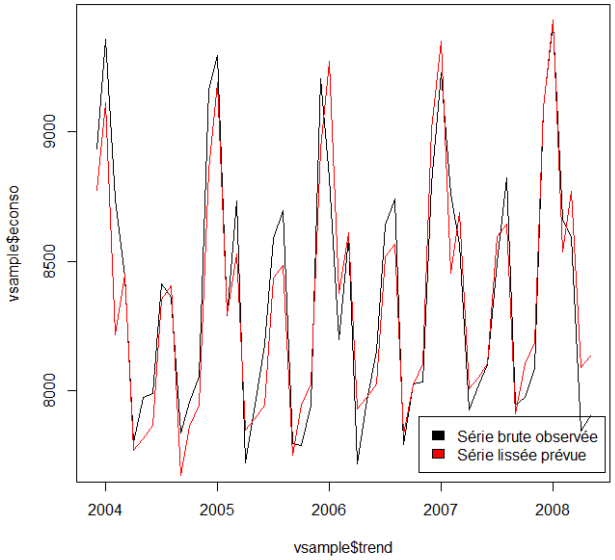
La prévision à la suite d'un HW se fait comme dans le cas d'un LES ou d'un LED en utilisant la fonction predict() avec l'horizon de prévision fixé.

Dans l'exemple suivant, on réalise une prévision sur un horizon égal à la dimension de l'échantillon de validation vsample afin de comparer les valeurs prévues avec les valeurs brutes observées:

```
econsoPred<-
predict(hwAd,n.ahead=nrow(vsample),prediction.interval=FALSE)
```

Visualisation de la série prévue :

```
graphics.off(); x11() ;
plot(vsample$trend,vsample$econso, type="l",
xlim=c(min(vsample$trend), max(vsample$trend)), lwd=1.8) #
valeurs observées
lines(vsample$trend[1:length(econsoPred)],econsoPred,col="red"
,xlim=c(min(vsample$trend), max(vsample$trend)), lwd=1.8 ) #
valeurs lissées prévues
legend(x=2006.8,y=7900,legend=c("Série brute observée","Série
lissée prévue"),fill=c("black", "red"))
```



5.5. Les modélisations stochastiques

5.5.1. Généralités sur méthodes stochastiques

Les méthodes stochastiques visent à modéliser le processus générateur des données dans le but de prédire les valeurs futures de la série. On distingue deux grandes approches de modélisation stochastique d'une série: l'approche autorégressive et l'approche moyennes mobiles.

L'approche autorégressive (**AR**) consiste à modéliser la série à partir de ses propres valeurs passées et l'approche moyenne mobile (**MA**) modélise la série à partir de ses erreurs (irrégularités) passées.

Ces deux approches peuvent aussi être combinées pour donner plusieurs variantes de modélisation. On distingue notamment:

- La modélisation **ARMA**: qui combine un modèle AR et un modèle MA
- La modélisation **ARIMA**: qui est une modélisation ARMA lorsque la série n'est pas stationnaire
- La modélisation **SARIMA**: qui est une modélisation ARMA pour une série non stationnaire et présentant des saisonnalités.

Il existe aussi des versions plus sophistiquées de modèle stochastiques dont le but est de répondre aux insuffisances des modèles de type AR. Il s'agit notamment des modèles de type ARCH-GARCH. Ces modèles sont utilisés lorsque les erreurs sont fortement hétéroscédastiques (variances des erreurs non constantes) et influencées par les valeurs passées.

Cette discussion vise à présenter les différentes étapes de la modélisation stochastique d'une série à travers des exemples d'application pratiques. Les principales étapes présentées sont:

- Diagnostic de la série
- Choix de modélisation
- Diagnostic sur la qualité du modèle
- Réajustement du modèle
- Validation du modèle
- Prévision

5.5.2. Diagnostic de la série

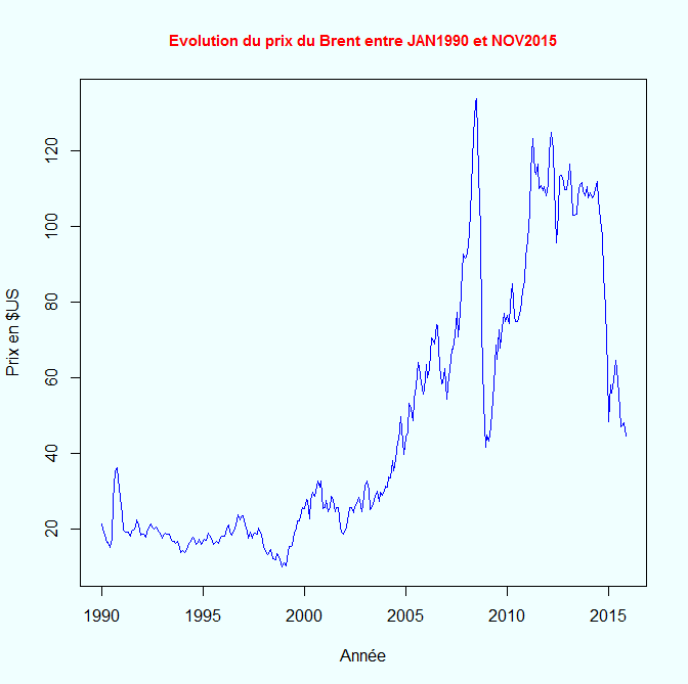
Pour la mise en pratique des concepts présentés, nous allons utiliser la série « pbrent » qui représente le prix du pétrole brent entre Janvier 1990 et Novembre 2015.

5.5.2.1. Préparation de la série et création de l'objet ts

```
library(xlsx)
mydata<-                          read.xlsx("dataWorkbook.xlsx",
sheetName="ppetrole",startRow=1,endRow=312,colIndex=c(1:2),
header=TRUE,as.data.frame= TRUE) # importation des données
mydata$date<- as.Date(mydata$date, format = "%Y-%m-%d")
mydata <- mydata[order(mydata$date, decreasing = FALSE),]
head(mydata,n=1LL)
tail(mydata,n=1L)
pbrent<- ts(data = mydata$pbrent, start = c(1990, 01), end =
c(2015,11), frequency = 12)
```

5.5.2.2. Visualisation de la série: courbe d'évolution

```
graphics.off();x11();par(mfrow=c(1, 1), bg ="azure",col.axis=
"black",        col.lab="black",col.main="red",col.sub="black",
fg="black",cex.main=0.9)
plot.ts(pbrent, col="blue", main="Evolution du prix du Brent
entre JAN1990 et NOV2015", xlab="Année", ylab="Prix en $US",
lwd=1.8)
```



5.5.2.3. Test de stationnarité

Le test de stationnarité (ou test de racine unitaire) vise à vérifier la régularité de la série c'est à une dire une série ayant les bonnes propriétés statistiques pour une modélisation ARMA.

Pour rappel, la stationnarité faible nécessite que la série ait:

- Une moyenne constante (i.e indépendante du temps)
- Une variance finie (variance constante- homoscédasticité)
- Une variance nulle (au-delà d'un certain retard p).

➢ Le test le plus couramment utilisé pour tester la stationnarité est le test Dickey Fuller Augmenté ADF (Voir partie théorique).

➢ Le test **ADF** peut aussi être accompagné par le test de **Philipe-Perron** et le test **KPSS** pour la robustesse de la conclusion.

Test de stationnarité: le test ADF

Le test ADF se met en œuvre séquentiellement (en partant du modèle 3 vers le modèle 1).

Etape 1 : Estimation du modèle avec constante et tendance (modèle 3)

Si la tendance n'est pas significative on passe alors à l'étape 2. Mais si la tendance est significative, on conclut le test à partir des résultats.

Etape 2 : Estimation du modèle sans tendance mais avec constante (modèle 2)

Si la constante n'est pas significative on passe alors à l'étape 3. Mais si la constante est significative, on conclut le test à partir des résultats.

Etape 3 : Estimation du modèle du modèle sans constante, ni tendance (modèle 1)

On conclut le test quels que soient les résultats.

NB: Le choix du nombre de retards p peut influer sur les conclusions. Choisir un p optimal (critère AIC)

Mise en œuvre du test ADF

Le test ADF se met en œuvre facilement sous R avec la fonction ur.df() du package urca:

Ci-dessous les étapes de mise en œuvre du test ADF (étapes **séquentielles avec les 3 modèles ADF**):

```
library(urca)
model3<-ur.df(y=pbrent,lags=6,selectlags="AIC",   type='trend')
# modèle avec constante et tendance (avec retard automatique
AIC)
summary(model3)
model2        <-        ur.df(y=pbrent,lags=6,selectlags="AIC",
type='drift') # modèle avec dérive (constate)
summary(model2)
model1  <-  ur.df(y=pbrent,lags=6,selectlags="AIC",  type='non')
# modèle sans tendance sans dérive.
summary(model1)
```

Interprétation du test ADF avec ur.df()

Pour le modèle 3, la ligne « Value of test-statistic is: » présentent trois statistiques tau3, phi2, phi3:

- tau3 correspond au coefficient de la valeur retardée yt-1

- phi2 correspond au coefficient de l'intercept (la constante)

- phi3 correspond au coefficient de la tendance

Les valeurs critiques correspondant à ces trois paramètres sont présentées au niveau de « Critical values for test statistics ».

Ici on retient le modèle 3 car la tendance est significative (voir la significativité du coefficient de la variable tt) dans la tableau de régression.

Conclusion du test

H0 :Présence de racine unitaire (i.e coefficient estimé nul, série non stationnaire)

Règle de décision:

Si la valeur calculée de tau3 est inférieure à la valeur critique affichée, on rejette H0 (coefficient significativement non nul, la série est stationnaire dans ce cas). En revanche si la valeur calculée de tau3 est supérieure à la valeur critique affichée, on accepte H0. La série est non stationnaire

Remarque: Si la valeur calculée de la statistique est inférieure à la valeur critique, la p-value du test sera inférieure au seuil d'erreur (alpha), on rejette H0. Par contre si la valeur calculée de la statistique est supérieure à la valeur critique, la p-value sera supérieure au seuil d'erreur (alpha), on accepte H0.

NB: La numérotation des paramètres du test (tau, phi) dépend du modèle:

- Pour le modèle 3 (modèle avec constante et tendance), on a: tau3, phi2, phi3;
- Pour le modèle 2 (modèle avec constante mais sans tendance), on a: tau2, phi1
- Pour le modèle 1 (modèle sans constante ni tendance), on a: tau1

Résultats du test (modèle 3)

```
###############################################
# Augmented Dickey-Fuller Test Unit Root Test #
###############################################

Test regression trend


Call:
lm(formula = z.diff ~ z.lag.1 + 1 + tt + z.diff.lag)

Residuals:
     Min       1Q   Median       3Q      Max
-18.8391  -1.9031   0.0721   2.0573  14.3906

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.095910   0.511453   0.188  0.85138
z.lag.1     -0.035888   0.013032  -2.754  0.00625 **
tt           0.010633   0.005196   2.046  0.04159 *
z.diff.lag   0.426923   0.052961   8.061 1.82e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.281 on 300 degrees of freedom
Multiple R-squared:  0.1842,    Adjusted R-squared:  0.176
F-statistic: 22.58 on 3 and 300 DF,  p-value: 3.306e-13


Value of test-statistic is: -2.7538 2.6251 3.9202

Critical values for test statistics:
      1pct  5pct 10pct
tau3 -3.98 -3.42 -3.13
phi2  6.15  4.71  4.05
phi3  8.34  6.30  5.36
```

Le coefficient de la tendance est significative: p-value=0.041. Alors on retient le modèle 3.

La valeur calculée de la statistique (-2.75) est supérieure à la valeur critique du test au seuil de 5% (-3.42), alors on accepte l'hypothèse H0 de présence de racine unitaire. Ce qui veut dire que la série non stationnaire.

Conclusion: la série pbrent est non stationnaire

5.5.2.3. Calcul de la différence première

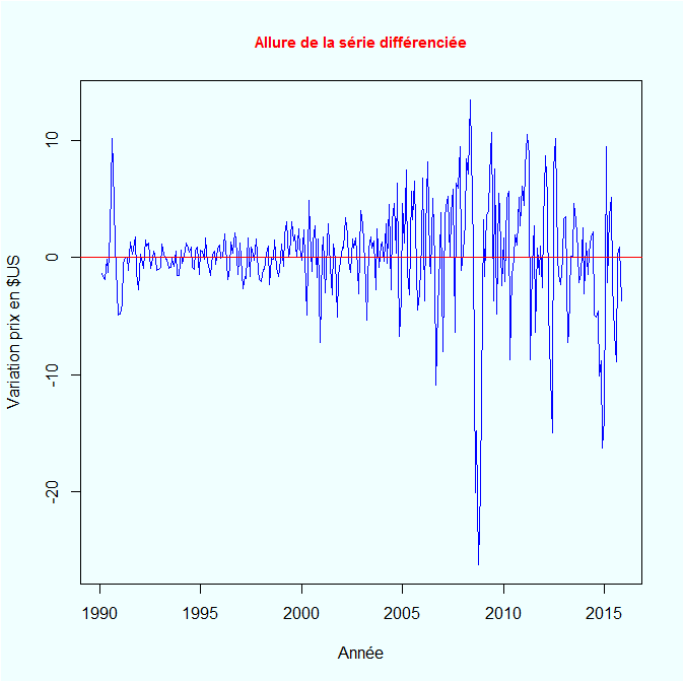
Comme la série pbrent est non stationnaire, on va alors calculer la différence première. Ensuite réaliser le test ADF sur la série différenciée.

Ci-dessous les étapes de différenciation:

```
mydata <- mydata[order(mydata$date, decreasing = FALSE),] # Tri par ordre croissant de la date
dpbrent<-diff(mydata$pbrent, lag = 1, differences = 1) # différence première avec un lag d'ordre 1
dpbrent<- ts(data = dpbrent, start = c(1990, 02), end = c(2015,11), frequency = 12) # définir l'objet ts
```

Visualisation de la série différenciée:

```
graphics.off();x11()
par(mfrow=c(1, 1), bg ="azure",col.axis= "black", col.lab="black",col.main="red",col.sub="black",
fg="black",cex.main=0.9)
plot.ts(dpbrent, col="blue", main="Allure de la série différenciée", xlab="Année", ylab="Variation prix en $US", lwd=1.8)
abline(h=mean(dpbrent), col="red") # ajoute une droite horizontal d'ordonnée égale à la moyenne de la série différenciée
```



5.5.2.4. Test ADF sur la série en différence première

Etapes séquentielles avec les 3 modèles:

```
library(urca)
model3        <-        ur.df(y=dpbrent,lags=6,selectlags="AIC",
type='trend') # modèle avec constante et tendance (avec retard
automatique AIC)
summary(model3)
model2        <-        ur.df(y=dpbrent,lags=6,selectlags="AIC",
type='drift') # modèle avec dérive (constate)
summary(model2)
model1 <- ur.df(y=dpbrent,lags=6,selectlags="AIC", type='non')
# modèle sans tendance sans dérive.
summary(model1)
```

Etapes séquentielles avec les 3 modèles:

Résultats des tests:

Modèle 3 non retenu car la tendance est non significative (p-value= 0.771).

Modèle 2 non retenu car la constante est non significative (p-value=0.951).

Par élimination, le modèle 1 est donc retenu pour conduire le test ADF

<u>Conclusion du test</u>

Résultats du test (modèle 1):

```
###############################################
# Augmented Dickey-Fuller Test Unit Root Test #
###############################################

Test regression none


Call:
lm(formula = z.diff ~ z.lag.1 - 1 + z.diff.lag)

Residuals:
     Min       1Q   Median       3Q      Max
-20.4206  -1.7389   0.2389   2.2138  15.0649

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
z.lag.1    -0.596786   0.062485  -9.551   <2e-16 ***
z.diff.lag -0.003379   0.057260  -0.059    0.953
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.296 on 301 degrees of freedom
Multiple R-squared:  0.3028,    Adjusted R-squared:  0.2982
F-statistic: 65.36 on 2 and 301 DF,  p-value: < 2.2e-16


Value of test-statistic is: -9.5508

Critical values for test statistics:
      1pct  5pct 10pct
tau1 -2.58 -1.95 -1.62
```

La valeur calculée de la statistique (-9.55) est inférieure à la valeur critique du test au seuil de 5% (-1.95), alors on rejette l'hypothèse H0 de présence de racine unitaire. Ce qui veut dire que la série dpbrent est stationnaire.

Conclusion: la série dpbrent est stationnaire.

NB: Le reste du travail de modélisation portera donc sur la série différenciée.

5.5.2.5. L'ordre d'intégration d'une série

L'ordre d'intégration d d'une série est le nombre fois qu'il faut différencier une série pour la rendre stationnaire.

Exemple: comme la différence première de la série pbrent est stationnaire, alors la série pbrent est intégrée d'ordre 1 (d=1).

Si la différence première dpbrent n'était pas stationnaire, on aurait différencié une seconde fois la série. Si cette différence seconde est stationnaire, cela signifierait alors que la série pbrent est intégrée d'ordre 2 (d=2).

L'ordre d'intégration est l'ordre de différenciation maximum qui rend la série stationnaire.

Pour une série déjà stationnaire: d=0.

5.5.2.6. Test d'autocorrélation: les corrélogrammes ACF et PACF

Après le test de stationnarité de la série et son éventuelle stationnarisation par « différence », la seconde étape du diagnostic de la série est l'analyse des autocorrélations. Deux indicateurs sont alors utilisés: L'autocorrélation totale (ACF) et l'autocorrélation partielle (PACF).

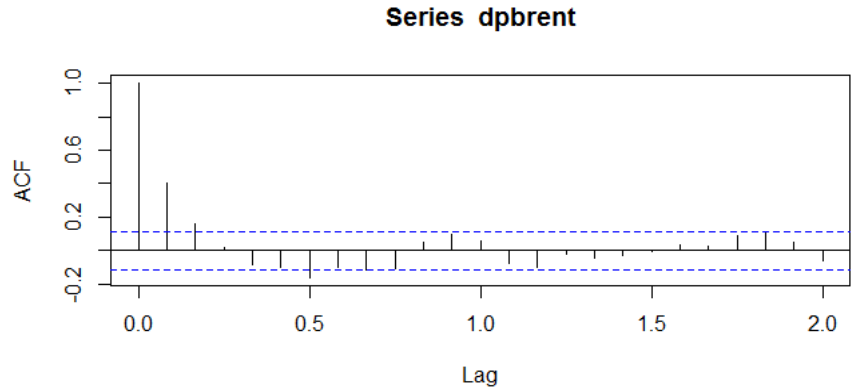
Ces indicateurs peuvent être examinés par visualisation graphique ou par analyses numériques.

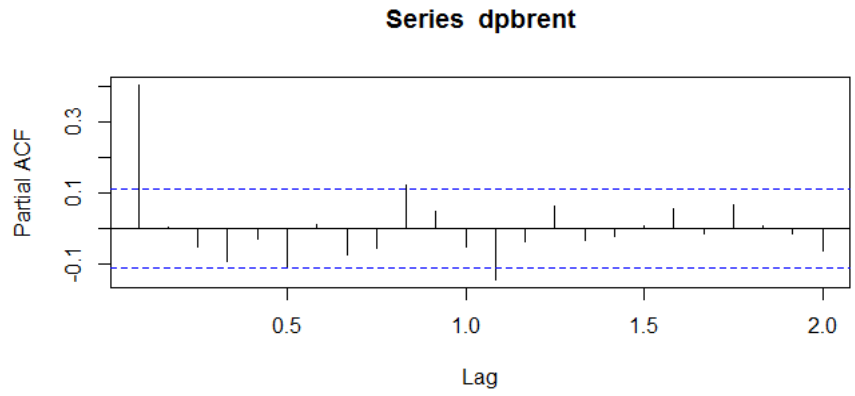
<u>Visualisation graphique</u>: les corrélogrammes

Les fonction R à utiliser pour afficher les corrélogrammes sont: acf() et pacf().

***Mise en œuvre sur la série dpbrent*:**

```
graphics.off(); windows() ;par(mfrow = c(2, 1))
acf(dpbrent, lag.max=24) # acf jusq'au retard d'ordre 24
pacf(dpbrent, lag.max=24) # pacf jusq'au retard d'ordre 24
```





Une autocorrélation totale (resp. une autocorrélation partielle) est significative à un retard d'ordre p (resp. d'ordre q) lorsque la « barre » située au niveau de ce retard sort de l'intervalle de confiance (exception faite du retard d'ordr 0, première barre).

On constate que pour l'autocorrélation totale (ACF), les autocorrélations d'ordre 1, 2 et 6 sont significatives.

Pour la PACF, les autocorrélations partielles significatives sont celles d'ordre 5, 9 et 12.

NB: L'ordre d'autocorrélation d'une série est l'ordre maximum observé. Ici 6 pour l'ACF et 12 pour la PACF.

<u>Test numérique</u>: le test de Box-Pierce

Les autocorrélations mises en évidence par l'analyse graphique peuvent être confirmées par une analyse numérique. On utilise, à cet effet, les tests dits « porte-manteau ».

Les tests « porte-manteau » les plus connus sont le test de **Ljung-Box** ou le test de **Box-Pierce**.

L'hypothèse nulle de ces tests est H0: absence d'autocorrélation (entre 1 et le retard maximum indiqué en paramètre).

Ci-dessous la mise en œuvre du test de Box-Pierce:

```
Box.test(dpbrent,lag=24, type = "Box-Pierce")
```

```
Box-Pierce test data: dpbrent
X-squared = 101.94, df = 24,
p-value = 1.402e-11
```

On rejette H0 car p-value=1.402e-11<< 5%.

5.5.2.7. Identification du modèle ARIMA(p,d,q)

Rappel: le modèle ARIMA(p,d,q) est le modèle ARMA(p,q) estimé sur une série intégrée d'ordre d.

- p: désigne l'ordre de la composante AR
- q: désigne l'ordre de la composante MA
- d: désigne l'ordre d'intégration de la série

L'ordre p se détermine à partir du PACF et l'ordre q se détermine à partir de l'ACF.

Dans chacun des deux cas, on choisit l'ordre maximum pour lequel la barre de corrélation sort de l'intervalle de confiance.

Pour la série dpbrent, on a: p=6 (ACF) et q=12 (PACF). Voir les corrélogrammes.

En se basant sur ces critères de choix, on a alors:

$$\begin{cases} p = 12 \\ q = 6 \end{cases} \Rightarrow \quad \text{AR(12)} \oplus \text{MA(6)} \Rightarrow \text{ARMA(12,6)}$$

Remarque:

Lorsque le modèle ARIMA est estimé sur une série déjà stationnarisée alors ARIMA(p,d,q)=ARIMA(p,0,q)

Ci-dessous d'autres équivalences:

Il existe de nombreuses équivalences entre les modèles ARIMA, ARMA, AR et MA

- **ARIMA(p,0,q)=ARMA(p,q)**

Un modèle ARIMA estimé sur une série stationnaire équivaut à estimer directement un modèle ARMA.

- **ARIMA(p,0,0)=AR(p)**

Un modèle ARIMA estimé sur une série stationnaire sans composantes moyennes mobiles équivaut à estimer directement un modèle AR(p).

- **ARIMA(0,0,q)=MA(q)**

Un modèle ARIMA estimé sur une série stationnaire sans composantes autorégressives équivaut à estimer un modèle MA(q).

Puisque nous travaillons sur la série dpbrent (série stationnarisée), alors d=0. Et au vu des ACF et des PACF, on a p=12 et q=6. Cela veut dire donc l'ordre maximum retenu pour le processus AR est 12 et celui pour le processus MA est 6.

On a donc ARIMA(12,0,6) → ARMA(12,6)

Attention: les ordres maximums p et q identifiés par les corrélogrammes n'indiquent pas nécessairement le modèle final à retenir pour les prévisions. Ces valeurs sont simplement des indications sur les plages de valeurs à l'intérieur desquelles le modèle final sera choisi.

Une pratique courante est d'estimer les modèles dans toutes les combinaisons possibles des valeurs de p et q (allant 0 à p et allant 0 à q). Ensuite retenir le modèle qui présente le minimum des critères d'information AIC ou BIC.

Dans cette présentation, nous allons nous limiter à trois modèles particuliers:

- Le modèle sans composantes moyennes mobiles (**AR**):

ARIMA(12,0,0) → ARMA(12,0)→AR(12)

- Le modèle sans composantes autorégressives(**MA**):

ARIMA(0,0,6) → ARMA(0,6)→MA(6)

- Le modèle avec composantes autorégressives et moyennes mobiles (**ARMA**):

ARIMA(12,0,6) → ARMA(12,6)→ARMA(12,6)

La fonction permettant d'estimer les processus stochastiques est la fonction arima(). Ci-après son application à chacun des modèles retenus.

5.5.3. Estimation d'un modèle AR(p)

Comme signalé précédemment, le modèle AR(p) est un cas particulier du modèle ARIMA(p,d,q) lorsque d=0 etq=0.

L'estimation du modèle AR(12) sur la série dpbrent se fait comme suit:

```
estimation_ar<-arima(dpbrent,order=c(12, 0, 0))
estimation_ar
library("lmtest")
coeftest(estimation_ar)   # pour voir la significativité des
coefs.
```

```
z test of coefficients:

           Estimate Std. Error z value  Pr(>|z|)
ar1        0.390850   0.056591  6.9065 4.967e-12 ***
ar2        0.024519   0.060692  0.4040   0.68621
ar3       -0.026299   0.060376 -0.4356   0.66314
ar4       -0.074673   0.060298 -1.2384   0.21557
ar5        0.013994   0.060224  0.2324   0.81625
ar6       -0.109530   0.060381 -1.8140   0.06968 .
ar7        0.043910   0.060532  0.7254   0.46821
ar8       -0.060949   0.060583 -1.0060   0.31440
ar9       -0.106625   0.060821 -1.7531   0.07959 .
ar10       0.106169   0.062382  1.7019   0.08877 .
ar11       0.071410   0.063339  1.1274   0.25956
ar12      -0.056227   0.060156 -0.9347   0.34996
intercept  0.087638   0.301977  0.2902   0.77165
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Dans cette estimation, seul l'ordre ar(1) apparaît significatif au seuil de 5%. Il peut alors être judicieux de réduire le nombre de termes AR dans le modèle. Toutefois, on peut rechercher le nombre d'AR optimal en se basant sur l'AIC (voir ci-dessous)

Recherche du nombre optimal d'AR

Le code ci-dessous estime les AR(p) possibles jusqu'à l'ordre 12 et retient et affiche le meilleur modèle (celui qui a le minimum de l'AIC).

```
p<-12 # l'ordre du AR
pvec<-c()  # vecteur vide qui dans lequel on a stocker les
valeurs de p
aicvec<-c() # pareil pour p
for (i in 0:p) {
  assign(paste("estimation",i,                          sep="_"),
arima(dpbrent,order=c(i, 0, 0)))
  results<-(get(paste("estimation",i, sep="_")))
  aic<-results$aic
  print(paste("p=",i, " ", "aic=",aic , sep=""))
  pvec<-c(pvec, i)
```

```
  aicvec<-c(aicvec, aic)
  rm(list=(paste("estimation",i,    sep="_")))    #    supprimer
l'objet
}
modelstat<-data.frame(p=pvec, aic=aicvec ) #
selected<-modelstat[which(modelstat$aic==min(modelstat$aic)),]
paicmin<-selected[1:1,"p"]
## Estimation du modèle final:
print( paste("La valeur optimale de p est", paicmin, sep=" "))
## On sélectionne cette valeur optimale pour estimer le modèle
finale
estimationF_ar<-arima(dpbrent,order=c(paicmin, 0, 0))
estimationF_ar
library("lmtest")
coeftest(estimationF_ar)
```

```
z test of coefficients:

          Estimate Std. Error z value  Pr(>|z|)
ar1       0.402586   0.051896  7.7576 8.653e-15 ***
intercept 0.063403   0.405199  0.1565    0.8757
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Ces estimations confirment bien que le modèle AR(1) est le meilleur modèle.

5.5.4. Estimation d’un modèle MA(q)

Le modèle MA(q) est un cas particulier du modèle ARIMA(p,d,q) lorsque d=0 etp=0.

L’estimation du modèle MA(6) sur la série dpbrent se fait comme suit:

```
estimation_ma<-arima(dpbrent,order=c(0, 0, 6))
estimation_ma
library("lmtest")
coeftest(estimation_ma)  #  pour  voir  la  significativité  des
coefs.
```

```
z test of coefficients:

           Estimate Std. Error z value  Pr(>|z|)
ma1        0.397718   0.056668  7.0183 2.245e-12 ***
ma2        0.169338   0.062056  2.7288  0.006357 **
ma3        0.024971   0.064113  0.3895  0.696914
ma4       -0.091502   0.071931 -1.2721  0.203345
ma5       -0.066219   0.075007 -0.8828  0.377322
ma6       -0.116587   0.056077 -2.0791  0.037612 *
intercept  0.077701   0.317048  0.2451  0.806398
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Dans cette estimation, les ordres ma(1), ma(2) et ma(6) apparaissent toutes significatives au moins au seuil de 5%. En revanche, les ordres ma(3), ma(4) et ma(5) ne sont pas significatifs au seuil de 5%. Cela incite alors à la recherche d'un ordre optimal de MA pour le modèle.

Recherche du nombre optimal de MA

Pour la recherche d'un nombre optimal de MA, on écrit une boucle qui estime toutes les combinaisons possibles de modèles jusqu'à l'ordre 6 et retient le meilleur modèle sur la base de l'AIC

```
q<-6 # l'ordre du AR
qvec<-c() # vecteur vide qui dans lequel on a stocker les
valeurs de p
aicvec<-c() # pareil pour q
for (i in 0:q) {
  assign(paste("estimation",i,                          sep="_"),
arima(dpbrent,order=c(0, 0, i)))
  results<-(get(paste("estimation",i, sep="_")))
  aic<-results$aic
  print(paste("q=",i, " ", "aic=",aic , sep=""))
  qvec<-c(qvec, i)
  aicvec<-c(aicvec, aic)
  rm(list=(paste("estimation",i,    sep="_")))    #    supprimer
l'objet
}
modelstat<-data.frame(q=qvec, aic=aicvec ) #
selected<-modelstat[which(modelstat$aic==min(modelstat$aic)),]
qaicmin<-selected[1:1,"q"]
## Estimation du modèle final:
print( paste("La valeur optimale de q est", qaicmin, sep=" "))
## On sélectionne cette valeur optimale pour estimer le modèle
finale
estimationF_ma<-arima(dpbrent,order=c(0, 0, qaicmin))
estimationF_ma
library("lmtest")
coeftest(estimationF_ma)
```

```
z test of coefficients:

          Estimate Std. Error z value  Pr(>|z|)
ma1       0.405342   0.056998  7.1115 1.148e-12 ***
ma2       0.186458   0.060824  3.0655  0.002173 **
ma3       0.090798   0.062909  1.4433  0.148928
intercept 0.062836   0.406444  0.1546  0.877138
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Ces estimations montrent que le meilleur modèle est le modèle MA(3) bien que le terme ma(3) ne soit pas significatif.

On peut donc exclure ce terme supplémentaire non significatif et estimer un MA(2)

```
estimation_ma2<-arima(dpbrent,order=c(0, 0, 2))
estimation_ma2
### pour voir la significativité
library("lmtest")
coeftest(estimation_ma2)
```

```
z test of coefficients:

          Estimate Std. Error z value  Pr(>|z|)
ma1       0.387365   0.054994  7.0438 1.871e-12 ***
ma2       0.160262   0.057947  2.7657   0.00568 **
intercept 0.064692   0.375279  0.1724   0.86314
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

5.5.5. Estimation d’un modèle ARMA(p, q)

Le modèle ARMA(p,q) est utilisé pour modéliser une série « stationnaire » ayant à la fois une composante autorégressive et une composante moyenne mobile.

Le modèle ARMA(p, q) est donc une généralisation des modèles AR(p) et MA(q). Il est donc estimé dans les mêmes conditions que les deux précédents modèles.

Toutefois, le choix des ordres p et q peut être soumis à une procédure de sélection en amont. Deux cas se présentent:

- soit on part des ordres maximums p et q initialement identifiés par les corrélogrammes;

- soit on utilise le couple (p,q) retenus à la suite des estimations individuelles des modèles AR(p) et MA(q).

La première approche est celle qui est recommandée car on part d’un cadre plus général.

5.5.6. Estimation du modèle ARMA(p,q)

Ci-dessous les étapes pour l’estimation du modèle ARMA initial:

```
estimation_arma<-arima(dpbrent,order=c(12, 0, 6))
estimation_arma
library("lmtest")
coeftest(estimation_arma) # significativité des coefs.
```

```
z test of coefficients:

           Estimate Std. Error z value  Pr(>|z|)
ar1        0.549280   0.144423  3.8033 0.0001428 ***
ar2       -0.652803   0.135528 -4.8167 1.459e-06 ***
ar3       -0.017916   0.096700 -0.1853 0.8530179
ar4       -0.462278   0.091774 -5.0371 4.726e-07 ***
ar5        0.558118   0.070068  7.9654 1.647e-15 ***
ar6       -0.926283   0.105786 -8.7562 < 2.2e-16 ***
ar7        0.293285   0.085647  3.4244 0.0006162 ***
ar8       -0.164327   0.061048 -2.6918 0.0071078 **
ma1       -0.128475   0.139197 -0.9230 0.3560242
ma2        0.636474   0.145804  4.3653 1.270e-05 ***
ma3        0.274354   0.169499  1.6186 0.1055289
ma4        0.546849   0.170400  3.2092 0.0013310 **
ma5       -0.411654   0.145676 -2.8258 0.0047161 **
ma6        0.754980   0.140007  5.3925 6.950e-08 ***
intercept  0.087086   0.330170  0.2638 0.7919644
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Recherche du nombre optimal de AR et de MA

Dans le modèle ARMA(12,6) estimé, plusieurs termes apparaissent non significatifs. Et vu le nombre très élevé de paramètres estimés, il peut être judicieux de procéder à une recherche optimale de nombre de retards p et q en élaborant une boucle. Et retenir au final le modèle qui présente le minimum de critère d'information AIC.

```
p<-12 # l'ordre du AR
q<-6 # l'ordre MA
pvec<-c() # vecteur vide qui dans lequel on a stocker les valeurs de p
qvec<-c() # pareil pour p
aicvec<-c() # pareil pour q et p
for (i in 0:p) {
  for (j in 0:q) {
    assign(paste("estimation",i,          j,          sep="_"), arima(dpbrent,order=c(i, 0, j)))
    results<-(get(paste("estimation",i, j, sep="_")))
    aic<-results$aic
    print(paste("p=",i,  "  ",  "q=",  j,"  ",  "aic=",aic  , sep=""))
    pvec<-c(pvec, i)
    qvec<-c(qvec, j)
    aicvec<-c(aicvec, aic)
    rm(list=(paste("estimation",i,  j,  sep="_")))  #  supprimer l'objet
  }
}
modelstat<-data.frame(p=pvec, q=qvec, aic=aicvec ) #
selected<-modelstat[which(modelstat$aic==min(modelstat$aic)),]
paicmin<-selected[1:1,"p"]
qaicmin<-selected[1:1,"q"]
```

```
## Estimation du modèle final:
print( paste("Les valeurs optimales de p et q sont",paicmin,
qaicmin, sep=" "))
estimationF_arma<-arima(dpbrent,order=c(paicmin, 0, qaicmin))
estimationF_arma
### pour voir la significativité
library("lmtest")
coeftest(estimationF_arma)
```

```
z test of coefficients:

           Estimate Std. Error z value  Pr(>|z|)
ar1        0.549280   0.144423  3.8033 0.0001428 ***
ar2       -0.652803   0.135528 -4.8167 1.459e-06 ***
ar3       -0.017916   0.096700 -0.1853 0.8530179
ar4       -0.462278   0.091774 -5.0371 4.726e-07 ***
ar5        0.558118   0.070068  7.9654 1.647e-15 ***
ar6       -0.926283   0.105786 -8.7562 < 2.2e-16 ***
ar7        0.293285   0.085647  3.4244 0.0006162 ***
ar8       -0.164327   0.061048 -2.6918 0.0071078 **
ma1       -0.128475   0.139197 -0.9230 0.3560242
ma2        0.636474   0.145804  4.3653 1.270e-05 ***
ma3        0.274354   0.169499  1.6186 0.1055289
ma4        0.546849   0.170400  3.2092 0.0013310 **
ma5       -0.411654   0.145676 -2.8258 0.0047161 **
ma6        0.754980   0.140007  5.3925 6.950e-08 ***
intercept  0.087086   0.330170  0.2638 0.7919644
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Ces estimations montrent que ARMA(8,6) est le meilleur modèle

5.5.7. Estimation d’un modèle ARIMA(p,d, q)

Le modèle ARIMA(p,d,q) est utilisé pour modéliser une série « non stationnaire » ayant à la fois une composante autorégressive et une composante moyenne mobile.

Le modèle ARIMA(p,d, q) est donc une généralisation du modèle ARMA(p,q).

Le modèle ARIMA estimé dans les mêmes conditions que le modèle ARMA à la seule différence que dans le premier cas la série est non stationnaire (intégrée d’ordre d) alors que dans le second, la série est déjà stationnarisée (intégrée d’ordre 0).

Par exemple, le modèle ARMA a été estimé sur la série « dpbrent ». Mais pour estimer un modèle ARIMA, il faut utiliser la série « pbrent ».

Sous R, le seul apport de l’utilisation du modèle ARIMA par rapport au modèle ARMA, c’est que le logiciel se charge directement de la différenciation de la série pour la rendre stationnaire et utilise ensuite le modèle ARMA pour estimer la série stationnarisée. Au final l’estimation du modèle ARMA (p,q) sans la constante sur la série dpbrent équivaut à l’utilisation de ARIMA(p,d,q) sur la série pbrent.

Estimation ARIMA(8,1,6)

Sous R, le modèle ARIMA est estimé de la même manière que le modèle ARMA en indiquant simplement les ordres p, d et q.

```
estimation_arima<-arima(pbrent,order=c(8,1,6))
estimation_arima
library("lmtest")
coeftest(estimation_arima) # significativité des coefs.
```

```
z test of coefficients:

     Estimate Std. Error z value  Pr(>|z|)
ar1  0.549481   0.144621  3.7995 0.0001450 ***
ar2 -0.652956   0.135985 -4.8017 1.573e-06 ***
ar3 -0.017972   0.097015 -0.1852 0.8530366
ar4 -0.462181   0.092055 -5.0207 5.149e-07 ***
ar5  0.558305   0.070248  7.9476 1.902e-15 ***
ar6 -0.926162   0.105971 -8.7397 < 2.2e-16 ***
ar7  0.293649   0.085753  3.4244 0.0006163 ***
ar8 -0.164038   0.061027 -2.6880 0.0071892 **
ma1 -0.128464   0.139384 -0.9217 0.3567068
ma2  0.636725   0.146293  4.3524 1.347e-05 ***
ma3  0.274623   0.170054  1.6149 0.1063291
ma4  0.547184   0.171071  3.1986 0.0013810 **
ma5 -0.411372   0.146228 -2.8132 0.0049047 **
ma6  0.755211   0.140418  5.3783 7.519e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

5.5.8. Estimation d’un modèle SARIMA(p,d, q,T)

Le modèle SARIMA(p,d,q,T) est utilisé pour modéliser une série « non stationnaire » ayant à la fois une composante autorégressive et une composante moyenne mobile et dont les composantes présentent des saisonnalités.

Le modèle SARIMA(p,d, q,T) est donc une généralisation du modèle ARIMA(p,d,q,T) où T représentent un vecteur de saisonnalités lié aux trois caractéristiques (p, d et q).

Sous R, le modèle SARIMA estimé dans les mêmes conditions que le modèle ARIMA en ajoutant un paramètre supplémentaire représentant les saisonnalités.

Pour le cas de la série pbrent, une modélisation SARIMA peut être comme suit:

```
estimation_sarima<-arima(pbrent,order=c(8,1,6),seasonal=list(
order=c(3,1,3),period=TRUE))
estimation_sarima
library("lmtest")
coeftest(estimation_sarima) # significativité des coefs.
```

Dans cet exemple, on choisit une saisonnalité d’ordre 3 pour la composante AR, une saisonnalité d’ordre 3 également pour la composante MA. Quant à la saisonnalité pour d, on choisit par défaut 1

```
z test of coefficients:

      Estimate Std. Error  z value  Pr(>|z|)
ar1  -0.181672   0.697005  -0.2606  0.794365
ar2  -0.244113   0.424685  -0.5748  0.565420
ar3   0.041460   0.353965   0.1171  0.906758
ar4  -0.631136   0.219906  -2.8700  0.004104 **
ar5  -0.038223   0.429031  -0.0891  0.929010
ar6   0.011107   0.284981   0.0390  0.968911
ar7  -0.185159   0.248785  -0.7443  0.456723
ar8  -0.146362   0.056200  -2.6043  0.009206 **
ma1   0.457128   0.776803   0.5885  0.556214
ma2   0.571587   0.198828   2.8748  0.004043 **
ma3  -0.302855   0.468714  -0.6461  0.518188
ma4  -0.320607   0.460742  -0.6959  0.486522
ma5  -0.844721   0.051568 -16.3807 < 2.2e-16 ***
ma6  -0.560513   0.576000  -0.9731  0.330497
sar1 -0.426154         NA       NA        NA
sar2 -0.634822   0.063018 -10.0736 < 2.2e-16 ***
sar3 -0.533135         NA       NA        NA
sma1 -0.462502   0.618228  -0.7481  0.454394
sma2  0.048025   0.732809   0.0655  0.947747
sma3  0.444443   0.460883   0.9643  0.334881
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

5.6. Prévision avec un modèle stochastique

Sous R, la prévision à la suite de l'estimation d'un modèle stochastique se fait avec la fonction predict() en spécifiant l'option *n.ahead* qui indique l'horizon souhaité.

La syntaxe de base de cette fonction est la suivante:

```
predict(object, n.ahead = 1, se.fit = TRUE)
```

- Objet désigne le nom sous lequel le modèle estimé a été stocké. Le modèle estimé peut être: AR, MA, ARMA, ARIMA, SARIMA.
- n.ahead: indique l'horizon de prévision souhaité. Ce paramètre est fixé par défaut à 1.
- se.fit : est un paramètre (TRUE ou FALSE) qui indique si les écart-types de prévision doivent être ajoutées à la série prévue

Exemple: Prévision avec un modèle ARMA(p,q)

Reprenons le modèle ARMA(8,6) obtenus dans les estimations précédentes avec la série dpbrent et effectuons une prévision sur 24 mois.

La syntaxe sera se présente comme suit:

```
prevresults_arma <- predict(estimationF_arma, n.ahead=24)
dpbrent_prev_arma<-prevresults_arma$pred # Variation prévue
pbrent_prev_arma<-
cumsum(c(pbrent[length(pbrent)],dpbrent_prev_arma))           #
reconstitution de la série en niveau (cumul)
pbrent_prev_arma<- ts(data = pbrent_prev_arma, start = c(2015,
12), frequency = 12)
```

```
graphics.off();x11()
par(mfrow=c(1, 1), bg ="azure",col.axis= "black",
col.lab="black",col.main="red",col.sub="black",
fg="black",cex.main=0.9)
plot.ts(pbrent_prev_arma, col="blue", main="Prévision du prix
du brent sur 24 mois", xlab="Année", ylab="Prix en $US",
lwd=1.8)
```

Remarque: Le modèle ARMA ayant été estimé sur la série en différence dpbrent, cela signifie qu'on a modélisé la variation (absolue) de la série et non le niveau de la série. Par conséquent les prévisions réalisées à partir du modèle estimé sont des prévisions des variations du prix du brent et non des prévisions des niveaux du prix du brent.

Pour obtenir les niveaux des prix prévus, il faut effectuer des calculs supplémentaires.

En effet, après avoir réalisé la prévision des variations du prix sur 24 mois, le principe pour déterminer les niveaux de prix correspondants sont les suivants:

On considère le dernier prix observé à la dernière date de l'échantillon. Notons ce prix p_t.

A l'horizon t+1, on ajoute ce dernier prix à la variation prévue à t+1. On a : $\hat{p}_{t+1} = p_t + \widehat{\Delta p}_{t+1}$

A l'horizon t +2, on ajoute la variation prévue $\widehat{\Delta p}_{t+2}$ au prix prévu (calculé) à l'horizon t +1 qui est $\hat{p}_{t+1}$. On a alors: $\hat{p}_{t+2} = \hat{p}_{t+1} + \widehat{\Delta p}_{t+2}$. Ce processus itératif continue jusqu'à l'horizon t+24.

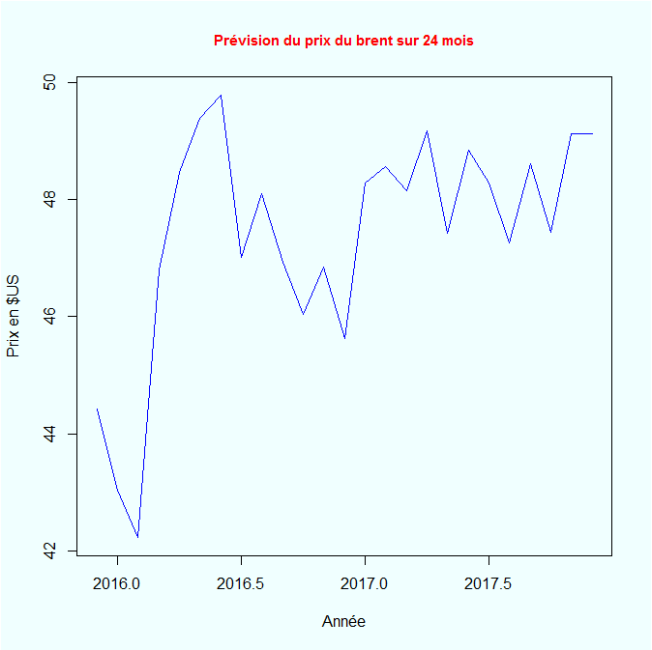
Cette itération est réalisée directement avec la syntaxe:

```
pbrent_prev_arma<-
cumsum(c(pbrent[length(pbrent)],dpbrent_prev_arma))
```

En réalité, il s'agit ici d'une somme cumulée sur un vecteur dont le premier élément est le dernier prix observé dans l'échantillon initial et dont les autres éléments sont les variations de prix prévues par le modèle ARMA estimé.

Voir la représentation graphique

NB: La modélisation avait été effectuée sur pbrent, les prévisions obtenues avec la fonction predict() correspondrait directement au niveaux des prix.



Bibiographie

Aragon Yves,(2011), Séries temporelles avec R :Méthodes et cas, Springer-Verlag France – Pratique R – 2011

Chambers, J. M. (2008), Software for Data Analysis: Programming with R, Springer, ISBN 978-0-38775935-7.

Chang, W. (2013). R graphics cookbook. O'Reilly Media, Incorporated.

Dutang, C., V. Goulet et M. Pigeon. (2008), «actuar: An R package for actuarial science», Journal of Statistical Software, vol. 25, no 7. URL http://www.jstatsoft.org/v25/i07.

Farnsworth, G. V. (2008). Econometrics in R. Consulté sur http://cran.r-project.org/doc/contrib/Farnsworth-EconometricsInR.pdf

Gallic Ewen, (2015), Logiciel R et programmation, l'Université de Rennes 1.

Goulet Vincent,(2016), Introduction à la programmation en R, 5ième édition, ISBN 978-2-9811416-6-8, Dépôt légal – Bibliothèque et Archives nationales du Québec, 2016

Hornik, K. (2013), « The R FAQ », URL http://cran.r-project.org/doc/FAQ/R-FAQ.html.

Iacus, S. M., S. Urbanek, R. J. Goedman et B. D. Ripley. (2013), « R for Mac OS XFAQ », URL http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html.

Ihaka, R. et R. Gentleman. (1996), «R: A language for data analysis and graphics», Journal of Computational and Graphical Statistics, vol. 5, no 3, p. 299–314.

Lafaye de Micheaux, P., Drouilhet, R., & Liquet, B. (2011). Le logiciel R : Maîtriser le langage - effectuer des analyses statistiques. Springer.

Paradis, E. (2002). R pour les débutants. Consulté sur http://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf

Sanchez, G. (2013). Handling and processing strings in R. Berkeley : Trowchez Editions. Consulté sur http://gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf

Wickham, H. (2009). ggplot2 : Elegant graphics for data analysis. Springer.