Introduction à R

Pascal Bessonneau

05/2016

Table des matières

1 Les premiers pas avec RStudio

1.1 Présentation générale

RStudio est né il y a quelques années et est le compagnon indispensable de R depuis deux ans environ.

Son interface est beaucoup plus attrayante et rappelle beaucoup l'interface d'autres logiciels de statistiques (SAS par exemple).

En outre elle a de nombreux avantages : apporte un meilleur confort de programmation R, une meilleure interface Sweave/knitr, facilite la programmation mixte R/C++, etc.

RStudio est une société commerciale qui contribue largement au développement du logiciel libre R et paquets avec, dans son équipe, de grands noms comme Hadley Wickham.

Elle vit de nombreux commerciaux : ses logiciels ont tous une version gratuite et une version payante

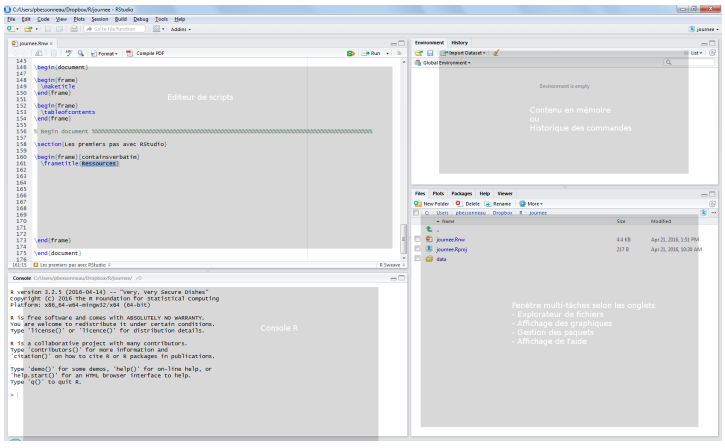
- RStudio Desktop : c'est une interface conviviale de développement R
- RStudio Server : c'est un serveur dont l'interface est identique à RStudio Desktop mais dont les commandes sont exécutées sur un serveur via un navigateur

1.2 Présentation générale de R-Studio

L'espace est divisé en quatre fenêtres :

- l'éditeur de scripts (en haut à gauche)
- le contenu de la mémoire ou l'historique des commandes (en haut à droite)
- la console R (en bas à gauche)
- une fenêtre contenant l'aide ou les graphiques ou l'explorateur de fichiers (en bas à droite)

1.3 Présentation générale



Comme vous l'avez remarqué dans chaque fenêtre, on passe d'une tâche à l'autre en utilisant **les onglets** : par exemple pour passer de l'explorateur de fichiers aux graphiques ou à l'aide (respectivement les onglets Files, Plots et Help).

Vous utiliserez essentiellement l'éditeur en haut à gauche.

Nous donnons également les raccourcis-claviers car ils peuvent être très pratiques.

1.4 Le fonctionnement

Tout en R est basé sur la ligne de commande (historiquement R se présente comme un shell).

Dans RStudio vous tapez les commandes dans l'éditeur puis vous les soumettez à R en utilisant le bouton *Run* en haut au milieu (ou en utilisant CTRL+ENTREE).

Par défaut *Run* ou CTRL+ENTREE soumettent la ligne sur laquelle se trouve le curseur ou une sélection d'une partie du script.

Quand vous créez un graphique ou que vous demandez de l'aide alors l'onglet correspondant se met en avant tout seul en bas à droite.

Dans R, tout est en mémoire vive (ou presque). En débutant tout les objets que vous manipulez ou que vous créez apparaissent dans la fenêtre en haut à droite : c'est (presque) tout le contenu de la mémoire.

Quand cet objet est une *data.frame* vous pouvez cliquer pour les visualiser (un peu comme dans un tableur). Attention, même s'il est possible d'éditer ces données c'est une opération à proscrire.

2 Les différentes fenêtres

2.1 L'éditeur (en haut à gauche)

Tout en R est basé sur la ligne de commande. Historiquement R se présente comme un shell.

Dans RStudio vous tapez les commandes dans l'éditeur puis vous les soumettez à R en utilisant le bouton *Run* en haut au milieu ou mieux en utilisant *CTRL+ENTREE*.

Par défaut *Run* ou *CTRL+ENTREE* soumettent la ligne sur laquelle se trouve le curseur ou une sélection d'une partie du script.

Par défaut l'éditeur est en mode *Script R*. Par conséquent il va essayer d'interpréter le contenu de la fenêtre pour vous aider en vous proposant les variables disponibles, les fonctions, etc.

La complétion se fait avec la touche *TAB*.

ATTENTION comme il essaie d'interpréter ce que vous écrivez il peut ralentir voire boguer quand vous prenez des notes dans l'éditeur.

Vous pouvez prendre des notes à condition de les mettre en commentaires. Pour cela commencer la ligne avec un #.

Une solution facile est de taper vos notes, de sélectionner le texte et de le mettre en commentaire en utilisant *CTRL+ALT+C* (ou dans le menu *Code*).

Les raccourcis-claviers les plus courants :

CTRL+ENTREE soumets la ligne ou la sélection à la console R

CTRL+ALT+C passer le contenu de script à commentaire ou l'inverse

CTRL+ALT+A pour indenter le code sélectionné comme un pro

CTRL+1 rend active la fenêtre éditeur

CTRL+2 rend active la fenêtre console

Un cheatsheet est disponible sur le site de RStudio.

2.2 La fenêtre de l'environnement global (en haut à droite)

Vous trouverez dans la fenêtre de l'évnirionnement globale la liste des objets que vous avez créés ou chargés en mémoire. Vous y trouvez ce qu'une commande *ls()* retourne.

Si ce sont des *data.frame* vous pouvez cliquer dessus pour ouvrir une vue "tableur". Seules les premières observations sont visibles.

2.3 La console R

Vous pouvez y taper directement les commandes qui seront interprétés par R.

Vous pouvez aussi voir dans cette fenêtre si il y a une erreur ou un message suite au code tapé ou au code soumis via l'éditeur. C'est dans cette fenêtre que vous retrouverez l'équivalent de la *log* sous SAS.

Vous avez accès à l'historique des commandes avec la flèche *en haut*.

2.4 La fenêtre multi-tâches (en bas à droite)

Vous y trouvez l'aide, les graphiques, la gestion des paquets selon l'onglet que vous choisissez. A essayer :

```
> ?rnorm
```

ou

```
> hist(rnorm(1000))
```

3 Les pas suivants...

3.1 Dans le futur

Apprenez à vous servir des projets... Ils permettent de travailler sur des projets différents en conservant l'interface (fenêtre, contenu mémoire,...) exactement comme vous l'avez laissé *lors du dernier lancement de RStudio*.

Outre la possibiité de produire des documents LaTeX, il est également possible de faire des documents HTML ou RTF incluant une belle présentation et le code R à l'intérieur...

Pour ceux qui sont intéressés RStudio est un outil de choix pour Sweave/knitr, le développement de paquets, le suivi de version avec git (ou svn), etc.

4 Qu'est ce que le langage R ?

4.1 Le type de langage

R est un langage interprété. C'est-à-dire qu'il va lire le code instructions par instructions et exécuté chacune d'elles à chaque fin de ligne.

La fin de ligne peut être au clavier la touche *entrée* ou la fin de ligne dans un fichier texte.

Il existe la possibilité de mettre plusieurs commandes sur la ligne avec le caractère " ;". Mais cela est déconseillé.

4.2 La casse

Le langage est casse-dépendant ce qui signifie que :

```
> A=5
> a=4
> A

## [1] 5

> a

## [1] 4
```

Cela est vrai pour les commandes, le nom des variables, ... mais également dans les comparaisons de chaîne de caractère.

Si on teste l'égalité de deux chaînes on obtient :

```
> "A"=="a"

## [1] FALSE

> "A"=="A"

## [1] TRUE
```

5 L'environnement de travail

5.1 La gestion de la mémoire (1)

Directement en tapant dans la console ou un programme généralement, les variables créées atterissent dans un "environnement" de base (*GlobalEnv*).

Sauf exception, les variables vont automatiquement être ajoutées à cet environnement.

Cet environnement comme tous les autres sont en mémoire vive contrairement à d'autres logiciels de statistiques.

De plus il n'y a pas de ramasse-miettes. Les variables ne sont supprimées que sur la demande de l'utilisateur/programmeur.

La place en mémoire vive est à la fois une force de R et une faiblesse.

Force car la vitesse d'exéuction en mémoire vive permet généralement des performances meilleures que dans d'autres logiciels.

Faiblesse car cela est problématique pour les gros volumes de données.

5.2 La gestion de la mémoire (2)

Il existe d'autres environnements que l'environnement de base. Par exemple quand on charge un paquet, les variables de ce paquet n'entre pas en conflit avec ses propres variables car les variables (et certaines fonctions) du paquet n'atterisse pas dans l'environnement global.

La création d'environnement est soit automatisé soit volontaire. Automatisé par exemple lorsqu'on crée une fonction, les variables de la fonction ne sont pas accessibles de l'extérieur.

Volontaire par exemple en appelant la fonction *new.env*.

5.3 Les opérateurs

Les opérateurs vont permettre l'assignation mais également les comparaisons et les opérations mathématiques.

```
> a <- 3 # Assignation
> a = 3 # Assignation (déconseillé)
> a == 3 # Comparaison

## [1] TRUE

> a != 3 # Différent de

## [1] FALSE

> a + 4  # somme

## [1] 7

> a | TRUE # Ou

## [1] TRUE
```

...

6 Opérateurs et variables

Comme tout langage les règles de précédence entre les arguments sont parfois complexes. Les parenthèses permettent de forcer la précédence.

```
> a <- 3+1
> (a <- 3) + 1

## [1] 4
```

6.1 Les variables

Les variables permettent de stocker les informations. Leur nom ne doit pas commencer par un nombre et ne pas contenir de caractères spéciaux tels que les blancs.

En vrai, c'est un peu plus compliqué. Il est possible d'avoir un nom de variable non standard en utilisant guillemets et/ou des guillemets inversés.

Vous pouvez utiliser des mots réservés par R. Cela ne soulève pas d'erreurs.

Par contre le mot réservé est "masqué". Il est possible de le faire mais cela est déconseillé. Notamment car la procédure de masquage/démasquage peut amener à des bugs subtils.

```
> min <- 3  # min est une fonction de base
> min       # on récupère bien le contenu de la variable

## [1] 3

> min(c(4,5,6)) # il y a "dé"masquage car R cherche une

## [1] 4

>               # fonction et non un nombre
> base::min(c(4,5,6))

## [1] 4
```

7 Les types de données

7.1 Typage

En informatique, toutes les informations sont stockées en binaire.

00111001 peut représenter aussi bien un nombre qu'un caractère.

Les types sont la nature des données que l'on peut stocker.

En R, il n'existe pas de typage fort, c'est-à-dire qu'une nouvelle variable est défini comme caractère, numérique en fonction de la première valeur qu'on mets dedans (la plupart du temps).

De plus une variable peut changer de type simplement en remplaçant le contenu par un autre.

Toutefois il existe des exceptions.

7.2 Les types de base

Les différents types sont les suivants :

integer ou entier en français. Il permettent de sauvegarder des nombres non décimaux

numeric ce type sert à stocker des nombres décimaux

complex ce type permet de stocker des nombres avec partie entière et partie imaginaire des complexes

character permet de stocker des chaines de caractères

logical permet de stocker des booléens (Vrai/Faux)

les dates , comme souvent les dates sont des objets délicats à manipuler. Les termes qui font référence à des type différents sont : *POSIXct*,*POSIXlt*, ...

factor les *factor* ce sont des modalités de variable ordinale ou qualitative

AsIs permet de stocker des données en utilisant l'opérateur *I()*. En pratique il se rencontre dans des fusions souvent c'est un signe que vous devriez vous inquiéter des résultats obtenus par la fusion

utilisateurs des packages (ou vous même) vous pouvez créer des types nouveaux

7.3 Les booléens

Les booléens sont des *TRUE* ou *FALSE* qui sont deux mots clefs de R. Ils peuvent être abrégés en *T* ou *F*.

Les booléens peuvent convertis en numériques naturellement ou explicitement.

Dans ce cas leurs valeurs sont 0 pour FALSE et 1 pour TRUE. Ce qui autorise de faire des sommes de booléen.

8 Compléments sur les types de base

8.1 Les *factor*

Les *factor* permettent de stocker des variables ayant peu de modalités. C'est par exemple le cas pour stocker : expérience A ou B, médicament Placebo ou Actif. L'étiquette est stockée sous forme de texte mais les facteurs sont aussi manipulables sous forme de nombres.

Cette dualité qui est très utile dans un laboratoire pour faire des ANOVA est un vrai problème quand on stocke des données complexes.

Si les facteurs sont indispensables pour faire certaines analyses comme les tests de Tukeyn certains auteurs font tout (ou presque) pour éviter leur utilisation comme H. Wickham.

8.2 La précision des nombres

Souvent, par exemple pour des entiers lors de sondages, il est nécessaire de s'inquiéter de la précision des types numériques. Ces limites peuvent dépendre de la machine utilisée.

Ces informations sont contenues dans l'objet *.Machine* que l'on peut taper directement ou on peut appeler seulement une partie.

Par exemple pour trouver les entiers maximums que l'on peut stocker sur la machine qui a compilé ce document :

```
> .Machine$integer.max

## [1] 2147483647
```

9 les objets de base

9.1 Vecteurs

L'objet le plus courant est de loin le vecteur. il s'agit d'un tableau à 1 dimension stockant un seul et unique type de données.

La création d'un vecteur est soit implicite soit explicite.

Implicite avec l'opérateur *c* :

```
> c(2,3,4,6,7)

## [1] 2 3 4 6 7
```

Soit explicite en appelant un créateur qui porte le nom du type que l'on souhaite stocké et sa longueur :

```
> numeric(4)

## [1] 0 0 0 0

> logical(4)

## [1] FALSE FALSE FALSE FALSE
```

9.2 Les fonctions associées aux types

Il existe deux familles de fonctions associées aux types :

is. les fonctions *is.* renvoie un booléen indiquant si la valeur appartient à un type donné

as. ces fonctions permettent de changer le type d'une valeur vers une autre (cast)

```
> is.integer(2L)

## [1] TRUE

> is.character(2.3)

## [1] FALSE

> as.character(2)

## [1] "2"
```

9.3 Matrices

Ensuite viennent les matrices qui sont des tableaux à deux dimensions :

```
> matrix( 0, ncol = 2, nrow = 2)

##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
```

Les matrices sont créés avec un premier argument qui contient les données et une (ou deux tailles, largeur et/ou longueur).


```
> matrix( 0, ncol = 2, nrow = 2)

##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
```

Les matrices sont créés avec un premier argument qui contient les données et une (ou deux tailles, largeur et/ou longueur).

Le type de données, *unique*, stocké pour les matrices est le type de données par le premier argument.

9.4 les *data.frame*

Les *data.frames* sont des tableaux comme les matrices mais qui permettent de stocker des types de données différentes dans chaque colonne.

```
> head(data.frame(lettre=LETTERS,numero=1:26))

##   lettre numero
## 1      A      1
## 2      B      2
## 3      C      3
## 4      D      4
## 5      E      5
## 6      F      6
```

Les *data.frames* sont de loin la structure la plus utilisée en faisant de la manipulation de données.

Mais ce type dérive en fait d'un autre type de base moins manipulable par les débutants : les listes.

10 les objets moins courants

10.1 les *list*

Les *list* sont la structure la plus pratique. Ce sont des vecteurs où chaque élement du vecteur peut être un objet R quelconque y compris une liste.

```
> a=list(1, LETTERS, matrix(0,2,2))
> str(a)
```

```
## List of 3
##  $ : num 1
##  $ : chr [1:26] "A" "B" "C" "D" ...
##  $ : num [1:2, 1:2] 0 0 0 0
```

Les *data.frames* sont en fait une liste avec comme condition que chaque élement soit un vecteur de même longueur (pour obtenir un tableau).

Ainsi les listes et les *data.frames* partagent beaucoup d'opérateurs en communs.

10.2 les *array*

les *array* sont des objets qui étendent les matrices à des tableaux à k-dimsions.

```
> array(1:4,c(2,2,2))

## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

10.3 les *list*

Les *data.frames* sont en fait une liste avec comme condition que chaque élement soit un vecteur de même longueur (pour obtenir un tableau).

Ainsi les listes et les *data.frames* partagent beaucoup d'opérateurs en communs.

Dans la programmation avancée, les *data.frames* sont souvent utilisées comme des listes.

10.4 les séries temporelles

Les séries temporelles sont des des élements pouvant stocker et avec des propriétés particulières des séries de date.

C'est le type utilisé pour tous les analyses en séries temporelles.

```
> ts(date())

## Time Series:
## Start = 1
## End = 1
## Frequency = 1
## [1] Mon May 16 12:06:03 2016
```

10.5 Les fonctions de transformation

Il existe deux familles de fonctions associées aux types :

is. les fonctions *is.* renvoie un booléen indiquant si la valeur appartient à un objet donné

as. ces fonctions permettent de changer un objet dans un autre type

```
> str(as.list(iris))

## List of 5
##  $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

> str(as.matrix(iris[,1:4]))

##  num [1:150, 1:4] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:4] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width"
```

11 Nommage des élements d'un objet

12 Les trois aspects de l'indexation

12.1 Dans cette partie...

... sont décrites les principales règles qui permettent de sélectionner une partie des objets les plus courants (le langage objet S4 est exclus).

12.2 Pour un vecteur

Pour un vecteur, l'opérateur d'extraction de valeurs sont des crochets. simples.

Comme on peut le voir sur les deux dernières lignes leur précédence n'est pas très forte mais plus grande que les opérateurs de calcul.

```
> LETTERS[1:10]

##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

> c(1,2,3,4)[1:2]

## [1] 1 2

> c(1,2,3,4)[-4]

## [1] 1 2 3

> c(1,2,3,4)[-4] * 2

## [1] 2 4 6
```

12.3 Les trois aspects de l'indexation

L'indexation sauf certaine exception peut se faire avec trois types de données sous R.

entiers avec les entiers, les chiffres positifs indiquent les positions des vecteurs qui seront extraits. Les chiffres négatifs indiquent les positions à exclure. Attention on ne peux pas mélanger chiffre positif et chiffres négatifs

12.4 les entiers

Avec les entiers, les chiffres positifs indiquent les positions des vecteurs qui seront extraits. Les chiffres négatifs indiquent les positions à exclure.

La longueur est quelconque (mais supérieur ou égal à 1). La longueur corresponds au nombre de d'éléments à extraire (ou à ne pas extraire).

Attention on ne peux pas mélanger chiffres positifs et chiffres négatifs

Attention Attention, amoureux du C et du perl, le zéro n'est jamais un indice valide en R.

La longueur du vecteur retourné est la longueur du vecteur de sélection pour les nombres d'entiers.

Pour les nombres négatifs, la longueur retournée est le total d'elements uniques du vecteurs moins la longueur du vecteur de sélection.

```
> LETTERS[c(1,2,3,4)]

## [1] "A" "B" "C" "D"

> LETTERS[c(2,4)]

## [1] "B" "D"

> LETTERS[c(2,4,2)]

## [1] "B" "D" "B"
```

```
> LETTERS[-c(1,2,3,4)]

##  [1] "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
## [11] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X"
## [21] "Y" "Z"

> LETTERS[-c(2,4)]

##  [1] "A" "C" "E" "F" "G" "H" "I" "J" "K" "L"
## [11] "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V"
## [21] "W" "X" "Y" "Z"

> LETTERS[-c(2,4,2)]

##  [1] "A" "C" "E" "F" "G" "H" "I" "J" "K" "L"
## [11] "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V"
## [21] "W" "X" "Y" "Z"
```

12.5 le vecteur logique

La grande différence avec les indices numériques est la longueur du vecteur.

Pour chaque position, si la valeur est *TRUE*, la valeur est retournée. Si c'est *FALSE*, la valeur n'est pas retourné.

Par conséquent la longueur du vecteur de sélection est la longueur du vecteur à séléctionner. Quant à la la longueur du vecteur de retour, c'est le nombre de *TRUE* (ie. la somme du vecteur logique).

Attention au recyclage ! Si le recyclage n'est pas possible R génère une erreur si la longueur des deux vecteurs ne coïncide pas.

```
> L <- LETTERS[1:6]
> L

## [1] "A" "B" "C" "D" "E" "F"

> L[c(T,F,T,F,T,T)]

## [1] "A" "C" "E" "F"

> L[c(T,F)] # recyclage

## [1] "A" "C" "E"
```

12.6 les noms

Les noms ont été évoqués brièvement... à peu près tout sous R peut porter un noms.

On peut donc utiliser un vecteur *character* avec le noms des élements pour les récupérer.

Les noms peuvent être utilisés en lieu et place des numéros. La longueur du vecteur de retour est alors le nombre de noms mis en arguments.

```
> (a <- 1:4)

## [1] 1 2 3 4

> (names(a) <- LETTERS[1:4])

## [1] "A" "B" "C" "D"

> a[c("A","D","A","C")]

## A D A C
## 1 4 1 3
```

12.7 Généralisation du système d'indexation

Une fois compris ce système d'indexation, le plus dur est fait car c'est sur ce système que se base pratiquement tout l'indexation des lignes et des colonnes d'une matrice, des élements d'une liste, ...

Quand l'objet est composé de lignes et de colonnes il suffit d'indexer et d'ajouter une "," pour indiquer à R sur quel dimensions on travaille.

12.8 *data.frames* et matrices

```
> str(iris[c(1,3,4),]) # à gauche -> lignes

## 'data.frame': 3 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.7 4.6
##  $ Sepal.Width : num  3.5 3.2 3.1
##  $ Petal.Length: num  1.4 1.3 1.5
##  $ Petal.Width : num  0.2 0.2 0.2
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1

> str(iris[,c(2,4,5)]) # à doirte -> colonnes

## 'data.frame': 150 obs. of  3 variables:
##  $ Sepal.Width: num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Width: num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species    : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

> str(iris[c(2,4,5),c(2,4,5)]) # les deux

## 'data.frame': 3 obs. of  3 variables:
##  $ Sepal.Width: num  3 3.1 3.6
##  $ Petal.Width: num  0.2 0.2 0.2
##  $ Species    : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1
```

12.9 Listes

Les listes sont proches des vecteurs. Les mêmes règles peuvent être appliquées.

Il y a toutefois une subtilité. Entre crochets simples, les listes renvoie une liste. Mais entre crochets doubles, un seul élement peut être renvoyé mais l'élement n'est pas de type liste mais du type contenu dans la liste à cette position.

C'est logique puisque vu l'hétérogénéité des élements pouvant être stockés dans une liste, R ne peut déterminer la meilleure stratégie pour rendre une série d'objets hétérogène.

Par contre quand un seul objet est renvoyé ce problème ne se pose pas.

```
> a <- list(1, LETTERS, matrix(0,2,2))
> a[c(T,F,T)]

## [[1]]
## [1] 1
##
## [[2]]
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
```

```
> a[[1]]

## [1] 1
```

13 Les différentres méthodes

13.1 Les méthodes

Tout d'abord R fournit des fonctions permettant de produire des graphiques simples grâce à quelques fonctions de base.

Ces graphiques dits "de base" sont assez simples à manipuler et à produire.

Après des méthodes plus avancés mais demandant beaucoup plus de dextérité sont disponibles dans les packages grid notamment. On parle de graphique "grid" ou "treillis".

Les graphiques grid ne seront pas évoqués. Par contre il existe deux packages réalisant de beaux graphiques simplement utilise en fait non les graphiques de base mais le type "grid" : *lattice* et *ggplot2*.

lattice est quelque peu passé de mode maintenant aussi il est préférable d'utiliser le paquet *ggplot2*. Sa belle esthétique est caractéristique pour un connaisseur.

14 Les graphiques de base

14.1 Les fonctions de base

hist pour faire un histogramme

barplot pour faire un diagramme en barre

boxplot pour faire des boites à moustaches

plot pour faire des "scatterplots"

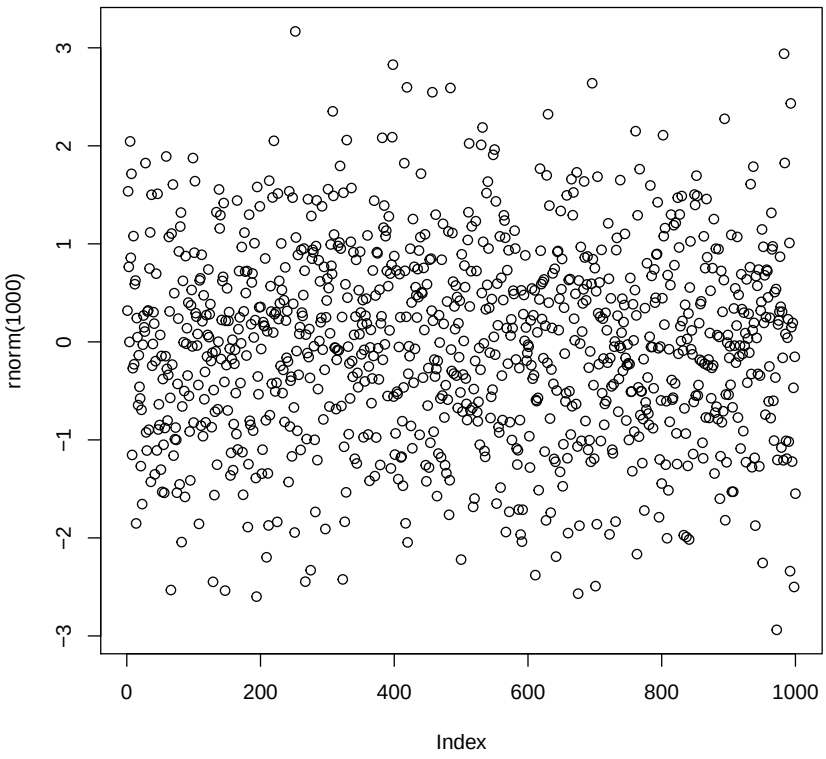
sunflowerplot pour faire des "scatterplots" quand les points se superposent

...

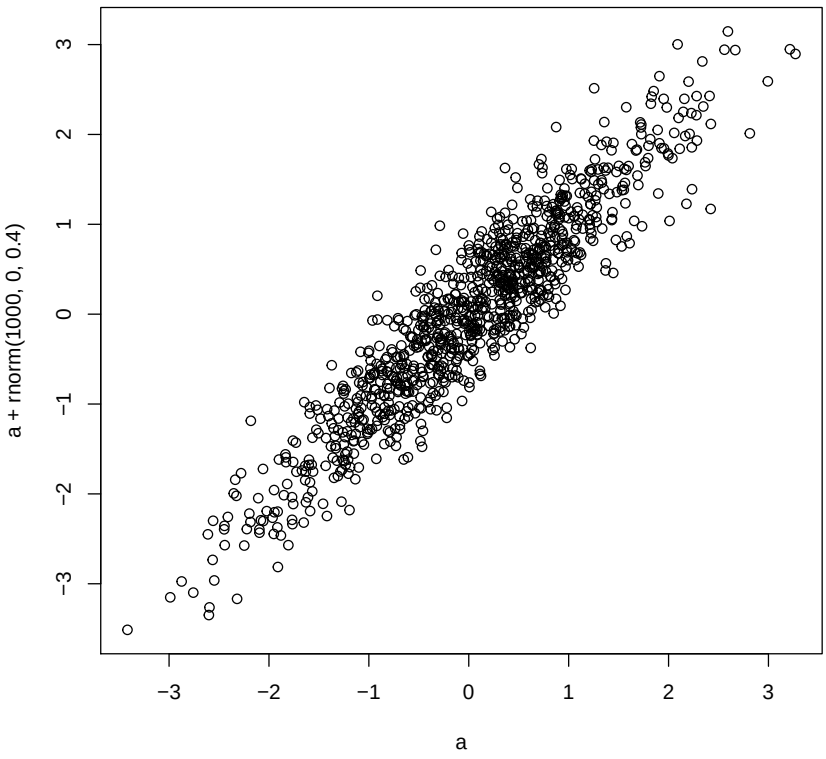
La fonction *plot* est polymorphe. C'est une fonction générique qui accepte de nombreux types d'objets en entrée et dont le résultat varie selon les arguments qui lui sont passés.

C'est une fonction générique de R.

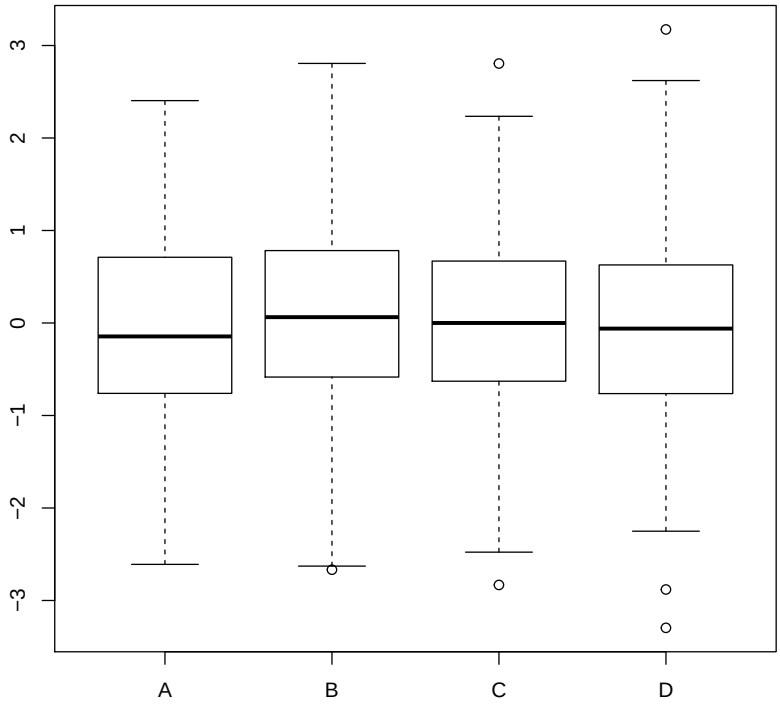
```
> plot(rnorm(1000))
```




```
> a=rnorm(1000)
> plot(a,a+rnorm(1000,0,0.4))
```




```
> plot(factor(sample(LETTERS[1:4],1000,replace=T)),
+              rnorm(1000)
+              )
```



De nombreux packages utilisent la fonction *plot* pour produire des graphiques en passant un argument propre au paquet.

C'est le cas par exemple pour une régression linéaire dans le paquet de base.

```
> a <- rnorm(1000)
> dt <- data.frame(
+   a=a,
+   b=a+rnorm(1000,0,0.4)
+ )
> rl <- lm(b ~ a)
> plot(rl,ask=F)
```

```
## Error in eval(expr, envir, enclos): objet 'b' introuvable
```

```
> class(rl)

## Error in eval(expr, envir, enclos): objet 'rl' introuvable
```

La fonction *plot* est une fonction générique. En fait la fonction spécialisé pour les objets de type *lm* est appelé en lieu et place de la fonction usuelle.

```
> methods(class=class(rl))

## Error in methods(class = class(rl)): objet 'rl' introuvable
```

Cette commande est très pratique pour connaître les fonctions implémentées pour ce type d'objet.

C'est en partie la raison pour laquelle on dit que R est un langage objet car c'est un langage dont certaines fonctions sont polymorphiques.

En fait on parle dans ce cas de modèle S3. Le modèle S4 qui est plus proche d'un vrai langage objet est peu utilisé pour l'instant.

Dans le cas des fonctions graphiques de base, les coordonnées sont calculés lors du première appel à la fonction.

Ensuite on peut rajouter des points ou des surfaces et les coordonnées sont exprimées dans les mêmes unités que celles des données.

Ce n'est pas le cas pour le paquet grid ce qui rend les manipulations plus complexes.

14.2 Les arguments les plus fréquents

Les fonctions de base accepte des arguments par défaut très souvent les mêmes :

xlim un vecteur contenant le minimum et le maximum pour l'axe des abscisses

ylim un vecteur contenant le minimum et le maximum pour l'axe des ordonnées

main le titre du graphique

xlab le nom de l'axe des abscisses

ylab le nom de l'axe des ordonnées

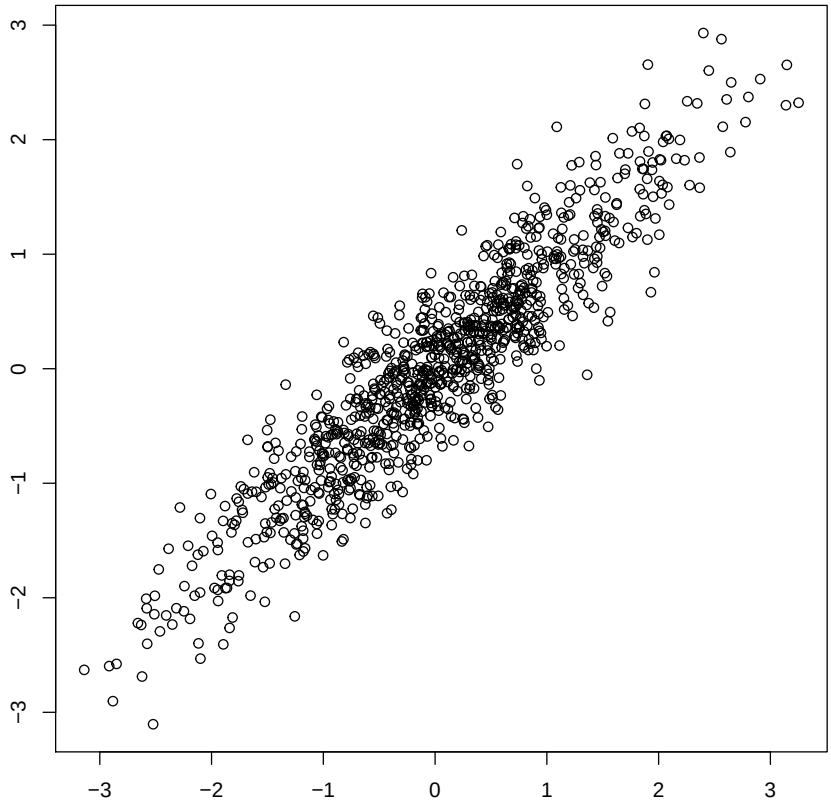
col la couleur des poiunts ou des surfaces

Le nombre de paramètres graphiques est impressionnant et il varie malheureusement un peu selon les fonctions.

on peut les visualiser avec la commande *par()*.

Par exemple on peut réduire les marges :

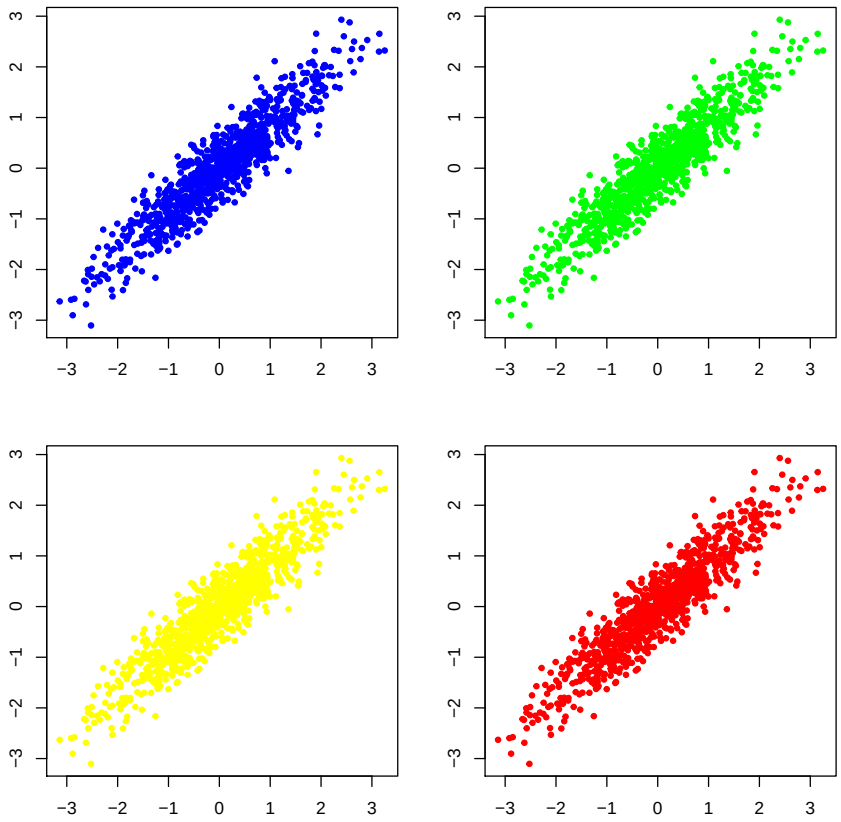
```
> par()$mar

## [1] 5.1 4.1 4.1 2.1

> par(mar=c(3.1,2.1,2.1,2.1))
> plot(dt$b,dt$a)
```

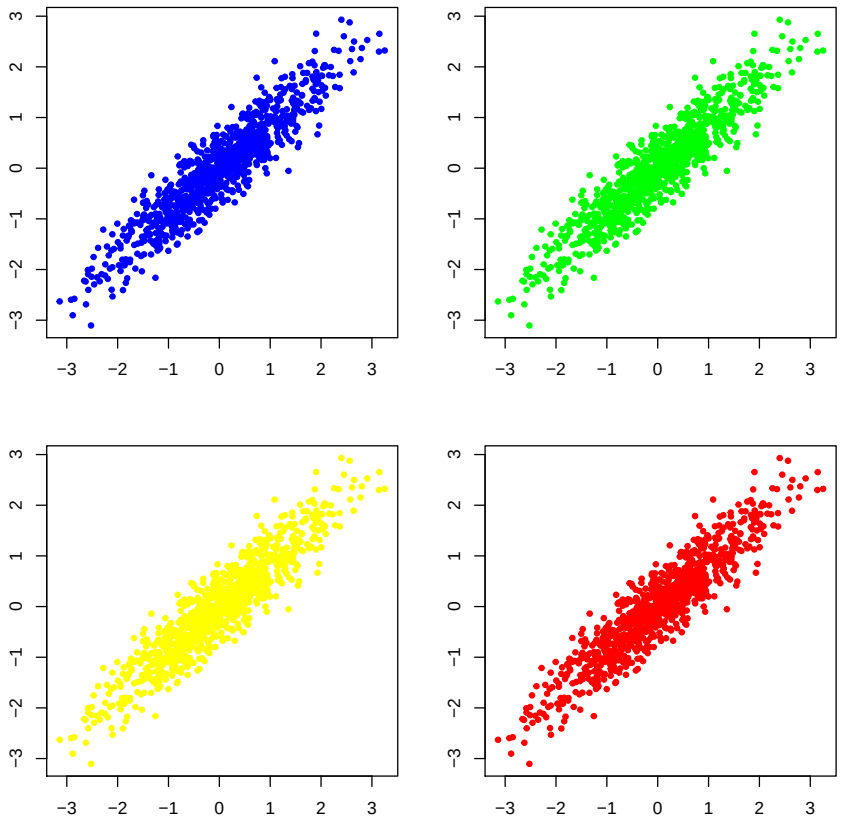


On peut également faire plusieurs graphiques sur la même page :

```
> par(mfrow=c(2,2))
> par(mar=c(3.1,2.1,2.1,2.1))
> plot(dt$b,dt$a,col="blue",pch=20)
> plot(dt$b,dt$a,col="green",pch=20)
> plot(dt$b,dt$a,col="yellow",pch=20)
> plot(dt$b,dt$a,col="red",pch=20)
```

On peut également faire plusieurs graphiques sur la même page :


−3 −2 −1 0 1 2 3
−3 −2 −1 0 1 2 3



14.3 Les fonctions de superposition

Il faut distinguer deux types de fonctions :

- les fonctions d'initialisation et de tracé
- les fonctions de superpositions sur un tracé

Un graphique R de base n'existe pas comme dit plus sans une échelle des X et une échelle des Y. Les fonctions comme *plot*, *barplot*, *hist*, ... initialise le graphique et font tout ce qu'il faut pour tracer un graphique.

Les fonctions de superposition permettent de réaliser des tracés mais en utilisant sur des graphiques existants.

Soit il s'agit de fonctions distinctes :

- *lines*
- *points*
- *text*
- ...

Les fonctions de superposition permettent de réaliser des tracés mais en utilisant sur des graphiques existants.

Soit il s'agit de fonctions d'initialisation mais avec un argument spécifique : généralement il s'agit de rajouter l'argument *add* :

```
> plot(x,y,add=T)
```

14.4 La notion de *device*

La sortie par défaut sur R est une fenêtre graphique. Par exemple dans RStudio l'onglet plot en bas à droite.

Mais on peut créer des graphiques dans des devices différents tels que des fichiers. Par exemple pour créer un fichier jpeg, il faut ouvrir un device *jpeg* qui va se substituer à la fenêtre graphique. Puis on va fermer le device pour finaliser l'export.

15 Les devices

```
> jpeg("graphiques/MonGraphique.jpeg")
> par(mfrow=c(2,2))
> par(mar=c(3.1,2.1,2.1,2.1))
> plot(dt$b,dt$a,col="blue",pch=20)
> plot(dt$b,dt$a,col="green",pch=20)
> plot(dt$b,dt$a,col="yellow",pch=20)
> plot(dt$b,dt$a,col="red",pch=20)
> dev.off()

## pdf
##   2
```

Les formats sont nombreux :

- png
- pdf
- svg
- ...

Chaque device porte le nom du type de fichier qu'il va créé.

15.1 La procédure avec les *device*

La procédure se fait avec les étapes suivantes :

1. vous créez le device avec par exemple *png("file")*
2. vous dessinez
3. vous refermez le device avec la commande *dev.off()*

Quelque devices comme le pdf permet de produire plusieurs graphiques sur plusieurs pages comme par exemple le format PDF.

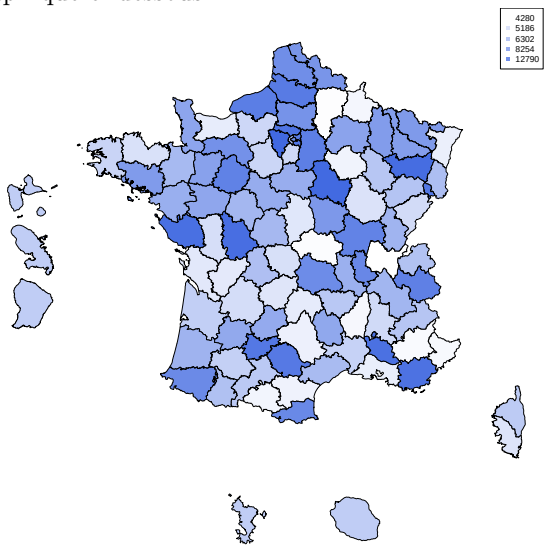
15.2 Les arguments des *device*

Les arguments varient, il suffit de regarder l'aide. Souvent on trouve l'argument dpi qui permet de donner la résolution du graphique.

Les arguments width et height donnent la taille dont l'unité dépend du device choisi.

15.3 Les options d'agencement avancée

L'option *layout* permet de définir des agencements avancés comme sur le graphique ci-dessous.



Il suffit de donner en entrée une matrice avec un numéro dans l'ordre de ce qui va être dessiner. Les fusions des cellules de la matrice donnent la taille de chaque zone.

Dans l'exemple, on représente un grand graphique central pour la France métropolitaine puis des petits carrés pour les DOMs.

Il suffit de donner en entrée une matrice avec un numéro dans l'ordre de ce qui va être dessiner. Les fusions des cellules de la matrice donnent la taille de chaque zone.

Dans l'exemple, on représente un grand graphique central pour la France métropolitaine puis des petits carrés pour les DOMs.

```
> xmetro <- 9
> xdom <- 1
> mm <- matrix(
+     c(
+       rep( 7,xdom), rep( 1,xmetro),
+       rep( 7,xdom), rep( 1,xmetro),
+       rep( 7,xdom), rep( 1,xmetro),
+       rep( 2,xdom), rep( 1,xmetro),
+       rep( 3,xdom), rep( 1,xmetro),
```

```
+       rep( 4,xdom), rep( 1,xmetro),
+       rep( 0,xdom), rep( 1,xmetro),
+       rep( 0,xdom), rep( 1,xmetro),
+       rep( 0,xdom), rep( 1,xmetro),
+       rep( 0,xdom), rep(0,3), rep(5,1), rep(0,1), rep(6,1), rep(0,xmetro-6)
+     ),
+     ncol=10,
+     nrow=10,
+     byrow=T
+ )
> mm
```

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

16 Répertoire de travail et parcours

16.1 La spécification des chemins sous R

Tous les répertoires doivent être indiqués avec la syntaxe *NIXs*, c'est-à-dire avec des slashs (/) en lieu et place des backslashs (\) sous *Windows*.

Pour rappel, il y a trois répertoires spéciaux à se souvenir :

. c'est le répertoire courant

.. c'est le répertoire parent du répertoire courant

~ c'est votre répertoire personnel

16.2 Le répertoire de travail de R

Le répertoire de travail de R est le point de référence pour accéder à vos fichiers.

Dans RStudio, le répertoire de travail ne change pas en manipulant l'explorateur de fichiers à droite. Il ne changera que si vous utilisez le bouton *More/Set as Working Directory*.

Le répertoire de travail de R dépend de la façon dont vous le lancez :

- avec la console graphique de R sous *Windows*, vous partez dans le répertoire X :\Program Files\R...
- avec RStudio, dans votre répertoire personnel
- depuis un terminal *Linux* dans le répertoire courant du shell
- ...

16.3 Les commandes utiles pour les répertoires

Les commandes à se souvenir sont les suivantes :

setwd pour *set working directory* qui permet de déterminer le répertoire courant

getwd pour *get working directory* qui renvoie dans un vecteur *character* le chemin courant

dir pour récupérer dans un vecteur *character* les fichiers du répertoire courant. Il faut spécifier *all=T* pour avoir les fichiers commençant par un point (même sous *Windows*).

Sur la console R, il y a des raccourcis dans les menus. Dans RStudio, vous avez la fenêtre *Files*.

16.4 Les chemins sous R

```
> setwd("~/Documents/R/FormationR")
```

```
> getwd()

## [1] "/home/pascal/Dropbox/R/FormationR"
```

```
> dir(pattern="tex")

##  [1] "01_RStudio-concordance.tex"
##  [2] "01_RStudio.synctex.gz"
##  [3] "01_RStudio.tex"
##  [4] "02_Le_langage-concordance.tex"
##  [5] "02_Le_langage.synctex.gz"
##  [6] "02_Le_langage.tex"
##  [7] "03_Les_types_de_donnees-concordance.tex"
##  [8] "03_Les_types_de_donnees.synctex.gz"
##  [9] "03_Les_types_de_donnees.tex"
## [10] "04_Les_types_objets-concordance.tex"
## [11] "04_Les_types_objets.synctex.gz"
## [12] "04_Les_types_objets.tex"
## [13] "05_Ligne_de_commande-concordance.tex"
## [14] "05_Ligne_de_commande.synctex.gz"
## [15] "05_Ligne_de_commande.tex"
## [16] "07_Graphiques-concordance.tex"
```

```
> dir(pattern="tex",all=T)

##  [1] "01_RStudio-concordance.tex"
##  [2] "01_RStudio.synctex.gz"
##  [3] "01_RStudio.tex"
##  [4] "02_Le_langage-concordance.tex"
##  [5] "02_Le_langage.synctex.gz"
##  [6] "02_Le_langage.tex"
##  [7] "03_Les_types_de_donnees-concordance.tex"
##  [8] "03_Les_types_de_donnees.synctex.gz"
##  [9] "03_Les_types_de_donnees.tex"
## [10] "04_Les_types_objets-concordance.tex"
## [11] "04_Les_types_objets.synctex.gz"
## [12] "04_Les_types_objets.tex"
## [13] "05_Ligne_de_commande-concordance.tex"
## [14] "05_Ligne_de_commande.synctex.gz"
## [15] "05_Ligne_de_commande.tex"
## [16] "07_Graphiques-concordance.tex"
```

17 Sauvegarde d'objets R

17.1 Sauvegarde de l'environnement

Il est appréciable de faire une sauvegarde complète de l'environnement avec lequel on travaille...

Dans ce cas, R permet de sauvegarder toutes les variables et fonctions en mémoire, seuls les paquets ne seront pas restaurés.

Il suffit d'utiliser la commande :

```
> save.image()
```

Dans le cas où il n'y a pas d'argument, le fichier s'appele automatiquement *.RData* et est sauvegardé dans le répertoire courant.

Lorsque R est lancé depuis un répertoire contenant un fichier dont le nom est *.RData*, il est automatiquement chargé.

La sauvegarde de l'environnement dans un fichier *.RData* vous est proposé quand vous quittez R, RStudio ou la console R.

Pour éviter ce comportement, pour un script par exemple, vous pouvez taper :

```
> q("no")
```

Lorsqu'on spécifie un argument, cela doit être une chaine texte qui indique le nom (et le répertoire éventuellement) du fichier.

Le format est un format *.Rdata* qui est le format propre de R. Il a la particularité d'être compressé (GNU/ZIP, *gzip*). Les fichiers produits sont donc légers.

17.2 Sauvegarde d'objets

Les objets complexes, *list*, *data.frame*, *array*, ... peuvent être sauvegardés au format *RData*.

Pour sauvegarder un(des) objet(s), il suffit d'utiliser la commande :

```
> save(iris,file="data/iris.RData")
> save(iris,mtcars,file="data/misc.RData")
```

Avec, en premier, l'objet (ou les objets) puis *file* et le nom du fichier.

17.3 Restauration d'objets

Il suffit pour recharger un environnement ou un objet (c'est-à-dire un objet de type *.RData*) d'utiliser la commande *load* :

```
> load("data/iris.RData")
```

La commande restaure en mémoire l'objet sous le nom que vous avez utilisé pour le sauvegarder (dans l'exemple c'est *iris*).

Le chargement en lui-même est « silencieux »... C'est-à-dire que R ne précise pas le(s) objet(s) qui est(sont) chargé(s) par la commande load.

Comme vu précédemment, l'objet sauvegardé prends en mémoire le nom qu'on lui a donné lors de la commande *save*.

Il est évident que, 6 mois après, il peut être difficile de se rappeler du nom de l'objet sauvegardé...

Comme souvent dans R, il faut changer le contexte d'évaluation pour obtenir le nom de l'objet :

```
> (load("data/iris.RData"))

## [1] "iris"
```

Pour des traitements automatisés, il est ainsi possible de récupérer le nom des objets sauvegardés.

```
> recharger <- load("data/iris.RData")
> recharger

## [1] "iris"
```

18 Fichiers texte

18.1 *read.table*

Sauf cas particulier, on peut charger (presque) n'importe quel type de fichier texte délimité avec la commande *read.table*.

La fonction *read.table* prend comme argument le nom du fichier texte.

Par défaut ce fichier texte doit avoir comme séparateurs des blancs entre les champs et pour séparateur entre la partie entière et la partie décimale un point.

La première ligne n'est pas considérée comme une entête mais comme des données.

Chaque colonne détectée correspondra à une variable qui sera restituée dans un *data.frame*.

La fonction *read.table* est une fonction de haut niveau. Par haut niveau, cela signifie que le plus gros du travail est épargné à l'utilisateur.

En effet la fonction va détecter elle-même le type de chaque variable/colonne.

18.2 les options de *read.table*

Qu'en-est-il des autres formats de fichier avec séparateurs ?

Il suffit de se rappeler les trois arguments suivants :

header si *TRUE* alors la première ligne est utilisée pour définir les noms de variable

sep c'est un vecteur caractère qui permet de définir quel(s) est(sont) le(s) séparateur(s) de colonnes

dec c'est un vecteur caractère qui définit le caractère utilisé pour séparer la partie entière de la partie décimale

A partir de ces réglages, on peut charger n'importe quel fichier délimité.

Par exemple, le format CSV français d'Excel s'écrit :

```
> iris <- read.table("data/iris.csv",
+ header=T,sep=";",dec=",")
```

Pour les formats de fichiers courants, il existe des alias de la fonction *read.table* : elles sont équivalentes à *read.table* mais ont des valeurs par défaut différentes.

Les version numérotées *2* correspondent aux formats français.

Table 1 – Alias de read.table

Arguments	read.delim	read.delim2	read.csv	read.csv2
header	T	T	T	T
sep	\t	\t	,	;
dec	.	,	.	,

Parmi les autres options utiles de *read.table*, on pourra noter les options suivantes :

stringsAsFactors par défaut *TRUE*, les variables de type *character*, si *TRUE*, seront transformées en variable de type *factor* automatiquement

encoding l'encodage du fichier qui peut par exemple être « latin1 » ou « UTF-8 »

na.strings un vecteur *character* indiquant la(es) valeur(s) à considérer comme une(des) valeur(s) manquante(s)

row.names le nom de la colonne ou le numéro de la colonne qui sera utilisée pour le nom des observations

nrows indique le nombre de lignes à lire

skip nombre de lignes à ignorer en début de fichier

Il est à noter deux options pour forcer certain aspect du chargement :

col.names vecteur *character* permettant de nommer les variables.

as.is vecteur *character* indiquant le type de chaque colonne

L'argument *as.is* permet notamment de spécifier un type *character* pour des codes « 001 », « 010 » qui seraient traités comme des nombres par R.

Dans ce cas, il peut être intéressant d'importer le fichier (ou une partie avec l'option *nrows*), récupérer le type de chaque variable dans un vecteur, et modifier seulement le type de quelques variables.

Sans utiliser les fonctions *apply*...

1. on récupère le nom des colonnes
2. on crée un vecteur avec NA pour l'auto-détection de type et *character* pour la variable *Species*
3. on charge le fichier avec ce vecteur comme argument de *ColClasses*

```
> iris <- read.csv2( "data/iris.csv", nrow = 1)
> types <- rep( NA, ncol(iris) )
> names(types) <- colnames(iris)
> types["Species"] <- "character"
> iris <- read.csv2( "data/iris.csv", colClasses=types )
> (types <- sapply(as.list(iris),class))
```

```
## Sepal.Length  Sepal.Width Petal.Length
##    "numeric"    "numeric"    "numeric"
##  Petal.Width      Species
##    "numeric"  "character"
```

Un exemple plus souple avec les fonctions *apply* :
Pour éviter de spécifier le type de chaque colonne :

1. on lit une première fois le fichier (en partie)
2. on récupère et on modifie le type de chaque colonne
3. on lit le fichier avec le type de colonne définitif

```
> iris <- read.csv2( "data/iris.csv", nrow = 10 )
> (types <- sapply(as.list(iris),class))

## Sepal.Length  Sepal.Width Petal.Length
##    "numeric"    "numeric"    "numeric"
##  Petal.Width      Species
##    "numeric"     "factor"

> types["Species"] <- "character"
> iris <- read.csv2( "data/iris.csv", colClasses=types )
> (types <- sapply(as.list(iris),class))

## Sepal.Length  Sepal.Width Petal.Length
##    "numeric"    "numeric"    "numeric"
##  Petal.Width      Species
##    "numeric"  "character"
```

18.3 l'aternative *readr*

Le paquet *readr* est un paquet de Hadley Wichkam (très connu).

Il fonctionne sur le principe d'automate finis par conséquent il est plus robuste aux erreurs dans les fichiers.

Par exemple il permet d'importer des fichiers contenant des verbatim avec des sauts de lignes si le champ est encadré par des quotes.

Le paquet a aussi comme particularité de disposer d'une fonction qui essaie de "deviner" le format du CSV (français/anglophone) permettant ainsi de simplifier l'importation.

Parmi les inconvénients, il y a le fait qu'il essaie de deviner le format (c'est un défaut et une qualité) et qu'il n'y a pas d'option spécifiques pour indiquer l'*encoding* du fichier.

Le diable se cache dans les détails, les fonctions de *readr* utilise la même syntaxe excepté qu'il faut remplacer les points par des "_".

```
> require(readr)
> iris <- read_csv2("Support R/data/Iris.csv")
```

Le diable se cache dans les détails, les fonctions de *readr* utilise la même syntaxe excepté qu'il faut remplacer les points par des "_".

```
> require(readr)
> iris <- read_csv2("Support R/data/Iris.csv")
```

readr permet de lire également des lignes de textes avec read_lines, des fichiers log, format fixe, etc. et ce plus rapidement que les fonctions de base.

18.4 l'aternative *data.table*

Avec le paquet *data.table* qui permet de manipuler des *data.frame* plus rapidement si celle-ci sont manipulées avec des variables "index" est livré une version de *read.csv*.

Attention le type de retour est *data.table* et non *data.frame*.

18.5 Exporter avec *write.table*

L'export d'un fichier texte délimité se fait avec la fonction *write.table*.

Comme la fonction *read.table*, elle a le même type d'alias.

Elle prend pour premier argument la *data.frame* à exporter puis l'argument *file* qui est un vecteur *character* indiquant le nom du fichier.

Les arguments essentiels sont les mêmes que pour *read.table* : *sep*, *dec* et *header*.

```
> write.csv2(iris,"data/iris.csv")
```

18.6 les options de *write.table*

Contrairement aux alias de *read.table*, les options des alias ne sont pas modifiables. Pour modifier le comportement pour les champs *sep*, *dec* et *header*, il faut passer par la fonction d'origine *write.table*.

Les options intéressantes sont notamment :

row.names par défaut *T*, les identifiants de ligne sont exportés dans une première colonne

append pour ajouter à un fichier existant si TRUE

na la valeur à utiliser pour les valeurs manquantes

col.names pour spécifier les noms des colonnes éventuellement

18.7 Chargement de données au format fixe

La fonction *read.fwf* est l'équivalent de read.table pour les anciens formats de fichiers texte sans séparateur de colonne mais à position de colonne fixe.

On indique la taille de chaque colonne séquentiellement.

Cette fonction a un comportement un peu surprenant. Quand *header=T*, les noms de colonnes sont récupérés, ils doivent être séparés par un séparateur de champs.

En fait cela est logique si on considère que les noms peuvent être plus longs que la largeur attribuée aux données.

```
AMC  Concord2229304099
AMC  Pacer  1733504749
AMC  Spirit 2226403799
BuickCentury2032504816
BuickElectra1540807827
```

Ce fichier est importé avec la commande :

```
> a=read.fwf("data/fixed.txt",width=c(5, 7, 2, 4, 4))
> colnames(a) <- c("model","make","mph","weight","price")
```

La fonction ci-dessous permet de récupérer automatiquement les noms s'ils utilisent la disposition des données.

```
> read.fwf2 <- function ( file, width, ... ) {

+    l = scan( file, what="character", nlines=1, sep="\n" )

+    col.names <- substr(
+         rep(l,length(width)) ,
+         cumsum(c(1,width[-length(width)])),
+         c(cumsum(width))
+    )
+    return(
+       read.fwf( file, width=width, col.names=col.names, skip = 1, ... )
+    )
+ }
```

On peut également importer ce type de fichier avec la commande *read.fortran* qui reprend la syntaxe de ce langage de 1977.

La syntaxe n'est pas développée ici. SAS utilise dans certains cas une syntaxe proche (pour les fonctions *put* et *input*).

18.8 Chargement de données texte avec les fonctions de bas niveau

Comme indiqué précédemment, les fonctions *read.table* sont des fonctions de haut niveau et nécessite peu de sueur pour l'utilisateur.

Mais elles reposent sur l'existence d'une fonction de bas niveau, *scan*, qui permet de lire n'importe quel format de fichier au prix d'un peu d'efforts.

18.9 Chargement de données texte/binaire avec les fonctions de bas niveau

R fournit aussi des fonctions de type C pour la lecture :

readChar pour la lecture caractère par caractère d'un buffer texte

readBin pour la lecture d'un fichier binaire

readLines pour la lecture ligne par ligne d'un fichier

18.10 Sauvegarde de données texte avec les fonctions de bas niveau

Comme indiqué précédemment, les fonctions *write.table* sont des fonctions de haut niveau et nécessite peu de travail pour l'utilisateur.

Mais elles reposent sur l'existence d'une fonction de bas niveau, *cat*, qui permet de créer n'importe quel type de fichier.

18.11 Sauvegarde de données texte/binaire avec les fonctions de bas niveau

Les fonctions de type C pour l'écriture sont :

cat pour l'enregistrement d'un buffer texte

writeLines pour l'enregistrement ligne par ligne d'un fichier

19 Autres Fichiers statistiques

19.1 le paquet *foreign*

Le paquet *foreign* permet de charger de nombreux formats externes.

Il fonctionne comme *read.table*, avec en lieu et place de *table*, le type de fichier.

Les formats supportés sont : Minitab, S, SAS, SPSS, Stata, Systat, dBase,...

19.2 SPSS avec *foreign*

Par exemple pour un fichier SPSS :

```
> iris.spss <- read.spss("data/iris.sav")

## re-encoding from CP1252
```

```
> class(iris.spss)

## [1] "list"
```

```
> str(iris.spss)

## List of 5
##  $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : chr [1:150] "setosa    " "setosa    " "setosa    " "setosa    " ...
##  - attr(*, "label.table")=List of 5
##   ..$ Sepal.Length: NULL
##   ..$ Sepal.Width : NULL
```

```
##   ..$ Petal.Length: NULL
##   ..$ Petal.Width : NULL
##   ..$ Species     : NULL
##  - attr(*, "codepage")= int 1252
##  - attr(*, "variable.labels")= Named chr(0)
##   ..- attr(*, "names")= chr(0)
```

Les fonctions de ce paquet sont relativement avancées et permettent, pour la plupart des formats de fichiers, de récupérer les attributs des variables et autres données additionnelles.

En conséquence, l'objet retourné n'est pas toujours une *data.frame*.
Dans le cas de SPSS, pour revenir à une *data.frame*, la commande est simple :

```
> str(as.data.frame(iris.spss))

## 'data.frame': 150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa    ","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Mais on peut perdre éventuellement les attributs.
Les attributs sont accessibles par la fonction *attributes*.

19.3 SAS avec *foreign*

Le code SAS pour exporter une table sous ce format est le suivant :

```
libname xportout xport 'iris.xpt';
data xportout.iris;
   set iris;
run;
```

Mais les limitations sont nombreuses et cela reste moins pratique que d'utiliser un format texte.

19.4 SAS avec *haven*

Contrairement au paquet *foreign*, le paquet *haven*, permet d'importer des fichiers sas au format *sas7bdat*, c'est-à-dire le format "normal" de SAS.

Il est assez efficace mais il ne supporte pas toutes les fonctionnalités de SAS. Il est possible que l'importation échoue car des fonctionnalités avancées sont utilisés dans la table SAS.

Il n'existe pas de documentation claire sur ce qui est supporté et ce qui ne l'est pas ce qui rend les choses problématiques.

```
> require(haven)
> eleves <- read_sas("eleves.sas7bdat")
```

Une documentation spécifique existe quand il y a des dates dans les fichiers.

19.5 Stata 13

La nouvelle version de Stata (supérieure ou égale à la version 13) inaugure un nouveau format de fichiers.

Il n'est pas supporté par *foreign* par contre le paquet *readstata13* permet l'importation de ces fichiers.

19.6 JSON et XML

R permet l'importation des fichiers XML et JSON. Le premier est une syntaxe commune sur les applications orientées web old school tandis que le deuxième est typique des nouvelles applications web utilisant javascript lourdement comme Node.js, JQuery, D3.js, ...

Par exemple sur de nombreux sites d'open data on trouve maintenant des exports en JSON pour faciliter la vie de ceux qui font des visualisations de ces données.

19.7 jsonlite

Le paquet le plus simple pour importer des données JSON est *jsonlite*.

On utilise l'argument simplifyDataFrame pour essayer de réduire les données à une *data.frame* dans la mesure du possible. Sinon les sorties sont du type *list* qui est la structure de données la plus proche du type JSON sous R.

Par exemple si on l'utilise sur les données des arbres remarquables collectées sur le site open data de la ville de Paris...

```
> require(jsonlite)

## Loading required package: jsonlite

> arbres <- fromJSON("data/arbresremarquablesparis2011.json",
+                    simplifyDataFrame = T)
```

Si on regarde de plus près, le retour est relativement complexe car les coordonnées géographiques notamment, font que l'objet est une liste. Les identifiants des arbres sont des vecteurs et les descriptifs des arbres sont dans une *data.frame* nommée*fields*. Les coordonnées sont dans une *list*.

En général, sur les formats un peu complexes, les traitements demandent un peu de programmation.

19.8 XML

Le paquet le plus utile et le plus rapide pour l'XML est le paquet *XML*. Il a fait l'objet d'un livre XML and Web Technologies for Data Sciences with R. Il n'est pas nécessaire d'acheter le livre pour se servir du paquet mais ce livre mais il est peu être utile si on travaille beaucoup avec les fichiers XML.

Le paquet contient une fonction *xmlToDataFrame* qui est généralement ce que l'on veut. Après il parfoit nécessaire de préciser sur quels noeuds on travaille avec la fonction *xmlNodes*.

Le paquet permet également de réaliser des requêtes *XSLT* éventuellement pour formater et/ou localiser les noeuds.

20 Bases de données

20.1 Les possibilités de R avec les bases de données

R est capable de transférer des données depuis une base de données vers R et inversement.

Il est en outre capable de lancer des commandes sur la base de données (suppression de tables, consultation des tables et des schémas).

Ces fonctionnalités sont basés, comme la plupart de langages modernes, sur le modèle DBI (Database Interface).

20.2 Le fonctionnement de la DBI

Le principe du module *DBI* est de rationnaliser les échanges avec les bases de données.

Ainsi on accède, sauf commande particulière, de la même façon depuis R à une base de données qu'il s'agisse de MySQL, PostgreSQL, Oracle ou SQLite.

La seule limite est de disposer d'un *driver* correspondant à la base de données auquelle on veut accéder.

1. Le module de R correspondant à la base de données est appelé par l'utilisateur
2. Le module *DBI* est chargé implicitement
3. L'utilisateur se connecte à la base de données
4. l'utilisateur effectue les requêtes qu'il souhaite
5. L'utilisateur, à la fin de l'utilisation, clôt la connexion

Il est à noter que l'utilisateur à la fin de la connexion doit récupérer l'objet créé. Cet objet complexe représente une connexion entre R et la base de données. Par la suite, cet objet est utilisé pour toutes les opérations sur la base.

Rien n'interdit (sauf l'administrateur de la bd) d'avoir plusieurs connexions actives à la base de données avec des paramètres utilisateur identiques ou différents.

Le danger est de ne pas penser à détruire l'objet *connexion* et de laisser une connexion ouverte sur la base de données.

Sauf exception, la création d'une connexion nécessite des droits utilisateur adéquats et une déclaration d'usage auprès de l'administrateur de la base de données.

Cela permet notamment de choisir le pilote le plus adapté pour l'attaque de la base de données par R.

Et à l'administrateur d'être vigilant pendant la période d'apprentissage.

La différence avec certains logiciels statistiques est qu'il est nécessaire de connaître un peu le langage SQL pour accéder aux données.

En effet les requêtes sont formulées en SQL.

20.3 Un exemple avec SQLite

SQLite est une base de données de « poche ». En effet ce n'est pas un serveur et donc être installé sur son poste.

Son usage ici est purement illustratif et pédagogique.

Comme dit précédemment, la *DBI* fait que les commandes ci-dessous sont (presque) les mêmes que pour un serveur MySQL ou Oracle.

20.4 Connexion à la base de données

Dans un premier temps, il faut se connecter à la base de données.

```
> require(RSQLite)
> driver <- dbDriver("SQLite")
> con <- dbConnect( driver, dbname = "data/eslc_stu.sqlite")
```

Le driver correspond au nom de la base de données. Il est passé en argument à la *DBI*. A noter que SQLite ne nécessite pas d'utilisateur et de mots de passe.

Dans le cas d'un serveur de base de données, la syntaxe est légèrement différente. La commande de l'utilisateur, l'adresse du serveur ainsi que les identifiants.

```
> require(RMySQL)
> driver <- dbDriver("MySQL")
> conn <- dbConnect(
+    driver,
+    host="127.0.0.1",
+    username="user",
+    password = "passwd",
+    dbname = "eslc"
+ )
```

20.5 Requêtes sur une table

Dans ce cas, on veut rapatrier des informations de la base vers une data.frame.

Le pilote se charge automatiquement des conversions nécessaires entre les noms de variables (caractères interdits pour les noms de variables pour R ou pour la base de données) et le type des variables (par exemple les dates sont converties).

20.6 Requête SELECT sur une table

Le principe est toujours le même :

1. on « forme » la requête
2. la requête est exécutée

Une première requête très simple...

```
> requete <- dbSendQuery( con, "SELECT * from stu")
> stu <- fetch(requete, n = 5 )
> dbClearResult(requete)
```

L'argument $n = 5$ de *fetch* indique que l'on ne veut rapatrier que les 5 premières lignes de la requête.

La commande *dbClearResult(requete)* est optionnelle pour certaines base de données.

Une seconde requête très simple avec une sélection... La requête peut être très complexe. Elle doit respecter la syntaxe SQL supportée par le serveur.

```
> requete <- dbSendQuery(
+    con,
+    "SELECT * from stu WHERE country_id='FR'"
+ )
> stu <- fetch(requete, n = 5 )
> dbClearResult(requete)

## [1] TRUE
```

Au lieu d'utiliser les guillemets simples, on peux utiliser \".

20.7 Bonnes pratiques avec les serveurs...

Avant d'aller plus loin, il est important de rappeler certains points :

- la méthode d'authentification, les identifiants et l'usage doit être approuvé par l'administrateur de la base de données.
- le paquet *DBI* camoufle une bonne partie de la complexité des échanges. La commande *dbClearResults* illustre par exemple le fait que la requête est mise en cache par le serveur. Il est primordial de suivre quelques règles simples lors des requêtes...
- Même si c'est optionnel, vider dès que possible le cache coté serveur avec *dbClearResults*
- Ecrire tous les scripts avec *fetch(con, n=limits)*. La variable *limits* sera fixée au début pendant le déboguage à quelques lignes. Puis pour la valeur sera mise -1 quand le script sera stable. Mieux, limiter dans le code SQL

le nombre de lignes récupérées (option *limit*, *fetch*, *rownum*,... selon le serveur).
- Sauf cas d'espèce, vous ne devez avoir qu' **une seule déclaration de connexion** : un seul objet *dbConnect*. Si vous en avez plusieurs, vous êtes autant d'utilisateurs sur le serveur que de connexions...
- Respecter toutes les précautions d'usage que vous employez habituellement en travaillant en SQL

Dans tous les cas, il est important de lire les recommandations indiquées dans la notice du pilote et de travailler de concert avec les administrateurs de la base de données.

20.8 Rapatrier une table...

Plutôt qu'une requête, vous pouvez rapatrier toute la table...

```
> stu <- dbReadTable( con, "NomDeLaTable" )
> stu <- dbReadTable( con, "NomDeLaTable", row.names=student_id )
```

20.9 Créer une table...

Si vous avez les droits, vous pouvez créer une table :

```
> dbWriteTable( con, "stu2", stu )
```

20.10 Les commandes non standardisées

Les commandes permettant de faire des requêtes de type *UPDATE*, *INSERT* utilise le *mapping*. La requête est écrite en SQL avec une syntaxe particulière pour les champs dont les données proviendront de R.

Lors de l'exécution, la commande met en relation chaque identificateur dans la requête avec une variable d'une *data.frame*.

Les commandes permettant de visualiser les tables de la base de données, la structure des tables, ... sont des commandes propres à chaque base de données.

De ce fait, ils ne sont pas standards.

Pour *RSQLite*, les seules commandes (dont le noms sont assez transparents) sont *dbListTables*, *sqliteCopyDatabase*.

21 Microsoft Office

21.1 Le paquet *XLConnect*

Le paquet *XLConnect* est relativement simple d'utilisation et permet d'importer et d'exporter des fichiers Excel au format 2003 ou 2007/2010/2013.

Le paquet est basé sur une bibliothèque Java de la fondation Apache. Office n'est pas nécessaire mais « Java » l'est : soit openjdk, soit le Java d'Oracle.

Le fait de pouvoir produire des fichiers Office sous NIXs par exemple est très appréciable.

Outre l'import et export simple, le paquet permet de modifier la présentation du tableur (couleurs,styles,...).

La documentation de *XLConnect* est particulièrement complète.

Le système ressemble un peu à l'accès à une base de données. Il y a deux étapes :

- on ouvre une connection sur un fichier Excel (existant ou nouveau)
- on manipule le fichier en utilisant la connexion créée

21.2 Lecture de fichier Excel

```
> wb = loadWorkbook("eslc.xlsx", create = T)
> data = readWorksheet(wb, sheet = "Eleves")
```

Selon le suffixe *xls* ou emphxslx, il s'adapte automatiquement à la version d'Excel 2003 ou 2010/../2013.

21.3 Sauvegarde dans un fichier Excel

```
> classeur = loadWorkbook("res_eslc.xlsx", create = T)
> createSheet(classeur, name = "eleves")
> writeWorksheet(classeur, eslc_eleves, sheet = "eleves")
> saveWorkbook(wb)
```

21.4 En plus de la base...

Il faut savoir que contrairement aux fonctions de R, les *rownames* ne sont pas exportés. Un argument permet d'utiliser une colonne existante pour les *rownames* mais les *rownames* eux-mêmes.

L'écriture est définitive quand la fonction *saveWorkbook* est appelée et l'écriture se fait en ajoutant au fichier existant.

Ainsi si un *data.frame* plus petit est exporté dans une feuille déjà pleine, il restera les données pré-existantes.

Lors de l'écriture pour éviter ces problèmes, le moyen le plus direct est de supprimer la feuille existante.

```
> classeur = loadWorkbook("res_eslc.xlsx", create = T)
> try({removeSheet(wb, sheet = "eleves")},silent=T)
> createSheet(classeur, name = "eleves")
> writeWorksheet(classeur, eslc_eleves, sheet = "eleves")
> saveWorkbook(wb)
```

Il y a une autre possibilité en vidant la feuille existante. Plus doux... Avec la fonction *clearSheet*.

Mais là également pour des traitements automatisés il faudra souvent utiliser des *try* pour éviter les erreurs lors de l'exécution.

21.5 Les arguments supplémentaires

Les arguments *startRow*, *startCol* permettent à l'importation et à l'exportation de se concentrer sur une zone de la feuille. Votre collègue pourra laisser son titre en première ligne avec les données en troisième ligne.

Vous pouvez également personnaliser le style des cellules : cela passe par la création d'un style de cellules, d'ajouter des paramètres de style et enfin l'appliquer à des cellules définies.

21.6 La mise en forme

```
> csHeader = createCellStyle(wb, name = "header")
> setFillPattern(csHeader,
+                fill = XLC$FILL.SOLID_FOREGROUND)
> setFillForegroundColor(csHeader,
+                        color = XLC$COLOR.GREY_25_PERCENT)
> setCellStyle(wb, sheet = sheet, row = 1,
+              col = seq(length.out = ncol(curr)),
+              cellstyle = csHeader)
```

Ce code est pompé sur la vignette car l'auteur n'a pas beaucoup pratiqué ces mises en forme. Remarquer qu'on retrouve un peu l'esprit Microsoft avec des constantes ...pas pratiques... pour appliquer un style (ex : *XLC$FILL.SOLID_FOREGROUND*). Elles sont contenues dans une grande liste *XLC*.

L'autre point non abordé est le fait que le paquet repose (en fait) beaucoup sur la définition de régions de cellules (ou noms sous Excel). Elle permet de contrôler l'importation, l'exportation, la mise en forme sur des portions de la feuille identifiées par des noms.

Pour cela voir les fonctions : *createName*, *writeNamedRegion*, ...

Et pour info vous pouvez insérer des graphiques (statiques !) au format png sur une feuille.

21.7 *readxl*, l'alternative à XLConnect

Le paquet *readxl* est un paquet qui permet la lecture de fichiers Excel sans la présence d'un moteur Java.

Il lit les fichiers des versions 2003 à 2013.

Il est très rapide mais a moins d'options que le paquet XLConnect et est donc très simple à utiliser.

```
> require(readxl)
> classeur = read_excel("res_eslc.xlsx")
```

Avec l'argument *row*, il est possible de définir à partir de quelle ligne, la lecture se fait. On peut également définir le type de chaque colonne : character, date, numeric, ...

Ainsi l'utilisation est très proche de read_csv du même auteur.

21.8 Les sorties pour le reporting

Pour rester dans les paquets pour Microsoft Office, si vous en avez besoin il existe le paquet *ReporteRs* qui permet l'exportation de résultats et/ou tableaux pour Word et Powerpoint. Le site est très bien fait et a de nombreux tutoriaux.

ReporteRs site

Mais ces paquets restent en retrait car ils sont destinés à la production de rapports plutôt qu'à l'esprit *litterate programming* de Sweave qui est remplacé maintenant par *knitr* ou *RMarkdown*.

Ils permettent de réaliser des "cahiers d'analyse" et/ou des rapports en permettant des documents dynamiques tout au long d'une analyse.

21.9 Shiny, D3.js, ...

Des paquets et le serveur Shiny permettent enfin d'exporter des documents totalement dynamiques pour le web.

Shiny est produit par la société qui produit RStudio. Le principe est de créer des documents web interactifs.

Pour avoir une idée : gallerie Shiny, paquet pour D3.js.

C'est pour créer des "datavisualistion" comme on peux voir dans le NYT par exemple.

Ces infographies sont plus simples à réaliser qu'il n'y paraît.

Il y a un MOOC sur Coursera encore gratuit pour l'instant qui permet de s'y mettre.

Et c'est très pédagogiques : Developping Data Products

et un sur le *Reproductible Research*https ://www.coursera.org/course/repdata

22 NoSQL et big data

22.1 Base NoSQL

Des paquets pour les bases NoSQL et pour le big data sont disponibles sur le CRAN :

- HadoopStreaming
- hive
- RcppRedis
- RCassandra
- ...

23 Recherche directe sur une fonction

23.1 Recherche directe sur une fonction

La recherche se fait à l'aide de la syntaxe *?fonction* ou bien *help(fonction)*.

La syntaxe de l'aide obtenue est théoriquement assez constante d'un paquet à un autre car elle est basée sur un format standard :.*Rd*. Ce format ressemble à un fichier LaTeX.

Dans l'aide, on trouvera les parties suivantes :

Début le nom de la fonction et le nom du paquet auquelle elle appartient

Description une description sommaire de la fonction

Usage la façon d'appeler la fonction

Arguments les arguments possibles pour la fonction

Details les détails sur le fonctionnement de la fonction

Value le type d'objet et/ou les valeurs retournées

Note des remarques générales

References les références bibliographiques

See also des liens vers des fonctions connexes

Examples des exemples fonctionnels de la fonction

La partie la plus difficile est la partie *Usage*. En effet c'est dans cette partie qu'on va lire la façon d'appeler la fonction selon le type de l'objet passé.

23.2 La partie Usage de t.test

```
Usage

t.test(x, ...)

## Default S3 method:
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)

## S3 method for class 'formula'
t.test(formula, data, subset, na.action, ...)
```

23.3 Recherche directe sur une fonction

La première description indique l'appel minimal.

Dans le second cas, la syntaxe indiquée correspond à l'appel de la fonction lorsque l'objet est tel que précisé dans l'aide.

Le troisième cas indique le comportement de la fonction si l'objet est type *formula*.

23.4 La partie *Usage* de *t.test*

C'est dans cette partie qu'on retrouve les arguments possibles de la fonction. Parfois, ils ne sont pas tous listés.

Les arguments sans valeurs par défaut sont des arguments obligatoires.

Les arguments avec une valeur par exemple *paired = FALSE* sont des arguments facultatifs car ils prennent comme valeurs par défaut la valeur indiquée.

Les arguments sont positionnels, c'est-à-dire qu'on peut les passer à la fonction dans l'ordre où ils sont cités dans la rubrique *Usage*.

Toutefois on peut aussi utiliser un mécanisme d'appel par nom : dans ce cas on passe le nom de l'argument suivi de sa valeur.

Dans les spécifications d'un paquet R, il est indiqué que les exemples doivent être fonctionnels (sauf si *Not run* est précisé en commentaire).

D'ailleurs on peut les exécuter en tapant *example(fonction)*

Toutes ces syntaxes suivantes sont équivalentes :

```
x <- rnorm(1000)
t.test(x,conf.level=0.8)
t.test(conf=0.8,x=x)
```

Il est à noter que l'aide de R est parfois cryptique dans les paquets de base. C'est notamment le cas de la fonction *plot*. La fonction a tellement de possibilités que l'aide ne fournit que les éléments de base.

23.5 *S3* et fonctions

R est un langage pseudo-objet pour le *S3*. C'est-à-dire que certaines fonctions vont se comporter différemment selon le type d'objet qu'on lui passe.

Cela repose sur l'évaluation de la classe de l'objet au niveau de l'appel ou de l'affichage...

Pour connaître la classe d'un objet il faut taper :

```
class(iris)

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

Pour connaître les fonctions génériques utilisables avec un type d'objet donné, il suffit de taper la commande :

```
methods(class="data.frame")

##  [1] aggregate     anyDuplicated as.data.frame
##  [4] as.list       as.matrix     by
##  [7] cbind         coerce        [<-
## [10] [             [[<-          [[
## [13] $<-           $             dim
## [16] dimnames<-    dimnames      droplevels
## [19] duplicated    edit          format
## [22] formula       head          initialize
## [25] is.na         Math          merge
## [28] na.exclude    na.omit       Ops
```

```
## [31] plot          print         prompt
## [34] rbind         row.names<-   row.names
## [37] rowsum        show          slotsFromS3
## [40] split<-       split         stack
## [43] str           subset        summary
## [46] Summary       tail          t
## [49] transform     unique        unstack
## [52] within
## see '?methods' for accessing help and source code
```

24 La recherche sur une fonctionnalité

24.1 La recherche par mots clefs via la console

R permet dans la console de chercher les fonctions associées à un mot-clef à l'aide la commande *help.search* ou *? ?*.

Cette fonction ne cherche que les occurences des mots dans les champs *noms* et *description* et seulement pour les paquets **installés**.

Pour réponse, R renvoie les noms des fonctions suivis des paquets correspondants et enfin d'un description brève des fonctions.

24.2 La recherche par mots clefs sur internet

Sur la page des packages du CRAN, on peut déjà chercher et trouver beaucoup d'informations.

L'autre méthode est d'utiliser le moteur de recherche RSeek qui ne référencie que les pages sur R. Le moteur cherche dans les newsgroups de R, les documents, ...

Outre les newsgroups, des sites de type Quora, dédiés à R ou aux statistiques, existent et permettent de poser des questions. On citera par exemple :

StatExchange ou Cross Validated orienté statistiques

StackOverFlow orienté programmation

25 Les documents sur R

25.1 La recherche de tutoriels et de documents

Un gros volume de documentation est disponible sur la page du projet dans la partie *Documentation*.

On trouvera également un journal sur le site de R.

Enfin beaucoup de paquets sont décrits dans le Journal of statistical software.

26 Les RUGs

26.1 Les groupes d'utilisateurs

Les R Users Groups ou *RUGs* sont très actifs... Deux sont présents dans la région parisienne :

Semin-R INED, MNHN, ...

FLtauR INSEE, ...

De nombreux documents, issus des conférences notamment, sont disponibles sur leurs sites. Le groupe *Semin-R* a une mailing-list pour poser des questions ainsi que les utilisateurs de R en sciences sociales R-soc.

Le groupe *Semin-R* a une mailing-list pour poser des questions ainsi que les utilisateurs de R en sciences sociales R-soc.

Les réponses à ces groupes seront « plus douces » si vous posez une question qui a déjà été posé (que sur les mailing-list officielles de R).

A noter que de nombreux paquets ont une mailing-lists ou un groupe Google pour obtenir de l'aide sur un paquet précis : knitr, FactoMineR, ggplot2, ...

26.2 Les conférences

Une réunion des utilisateurs français est organisé maintenant chaque année. Cette année (2015) à Grenoble sur les jours de la formation.

Les conférences **UseR !** sont mondiales. En 2015, début juillet en Aalborg qui comme chacun sait est au Danemark.

27 Statistiques descriptives : variables quantitatives

27.1 Les statistiques de base

Pour avoir un aperçu à partir des indicateurs les plus courants, on peut utiliser la fonction *summary*.

Cette fonction donne un résumé de la variable adaptée au type de la variable.

```
> summary(iris)

##   Sepal.Length    Sepal.Width
##  Min.   :4.300   Min.   :2.000
##  1st Qu.:5.100   1st Qu.:2.800
##  Median :5.800   Median :3.000
##  Mean   :5.843   Mean   :3.057
##  3rd Qu.:6.400   3rd Qu.:3.300
##  Max.   :7.900   Max.   :4.400
##   Petal.Length    Petal.Width
##  Min.   :1.000   Min.   :0.100
##  1st Qu.:1.600   1st Qu.:0.300
##  Median :4.350   Median :1.300
##  Mean   :3.758   Mean   :1.199
##  3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

Si on veut extraire ces informations, on peut utiliser la fonction *sink*.

Cette fonction permet de rediriger les résultats affichés à l'écran vers un fichier.

```
sink("resultats.txt")
summary(iris)
sink()
```

Mais des fonctions sont disponibles pour chaque indicateur :

mean donne la moyenne. L'argument *trim* permet d'écarter une certaine proportion des valeurs extrêmes dans le calcul de la moyenne.

min donne le minimum

max donne le maximum

range donne le minimum et le maxium dans un vecteur de longueur 2

sd donne l'écart-type

median donne la médiane

quantile donne les quantiles c(0,0.25,0.5,0.75,1) mais qui sont modifiables par l'option *probs*

On peut ainsi calculer pour une variable quantitative quelques statistiques :

```
> mean(iris$Sepal.Length)

## [1] 5.843333

> sd(iris$Sepal.Length)

## [1] 0.8280661

> quantile(iris$Sepal.Length)

##   0%  25%  50%  75% 100%
##  4.3  5.1  5.8  6.4  7.9
```

```
> r <- c(
+   mean(iris$Sepal.Length),
+   sd(iris$Sepal.Length),
+   quantile(iris$Sepal.Length)
+   )
> names(r) <- c("Moy.","EC","Min","1erQ","Méd.","3emeQ","Max")
> r

##      Moy.        EC       Min      1erQ
## 5.8433333 0.8280661 4.3000000 5.1000000
##      Méd.     3emeQ       Max
## 5.8000000 6.4000000 7.9000000
```

```
> r <- rbind( r, c(
+   mean(iris$Sepal.Length),
+   sd(iris$Sepal.Length),
+   quantile(iris$Sepal.Length)
+   )
+ )
> rownames(r) <- c( "Sepal.Length", "Sepal.Width" )
> r

##                  Moy.        EC Min 1erQ
## Sepal.Length 5.843333 0.8280661 4.3  5.1
## Sepal.Width  5.843333 0.8280661 4.3  5.1
##              Méd. 3emeQ Max
## Sepal.Length  5.8   6.4 7.9
## Sepal.Width   5.8   6.4 7.9

> write.csv2(r,"data/Resultats.txt")
```

Il existe une connexion de fichier particulière : "clipboard".

Dans ce cas, le résultat n'est pas écrit dans un fichier mais dans le presse-papiers par exemple pour le copier-coller dans une application tierce.

```
> write.csv2(r,"clipboard")
```

Pour les CSVs, cela ne donne pas grand chose, mais le paquet *questionr* contient la fonction *clipcopy* qui permet d'exporter un objet au format HTML (via le paquet *R2HTML*).

```
> clipcopy(r)
```

L'objet peut alors peut alors être directement coller dans un tableur.
Attention, toutes ces fonctions renvoient *NA* si une valeur ou plus est manquante (ie. égale à *NA*).

```
> min(c(3,NA,5))

## [1] NA
```

Pour ne pas avoir *NA* comme réponse, il faut utiliser un argument supplémentaire *na.rm=T*

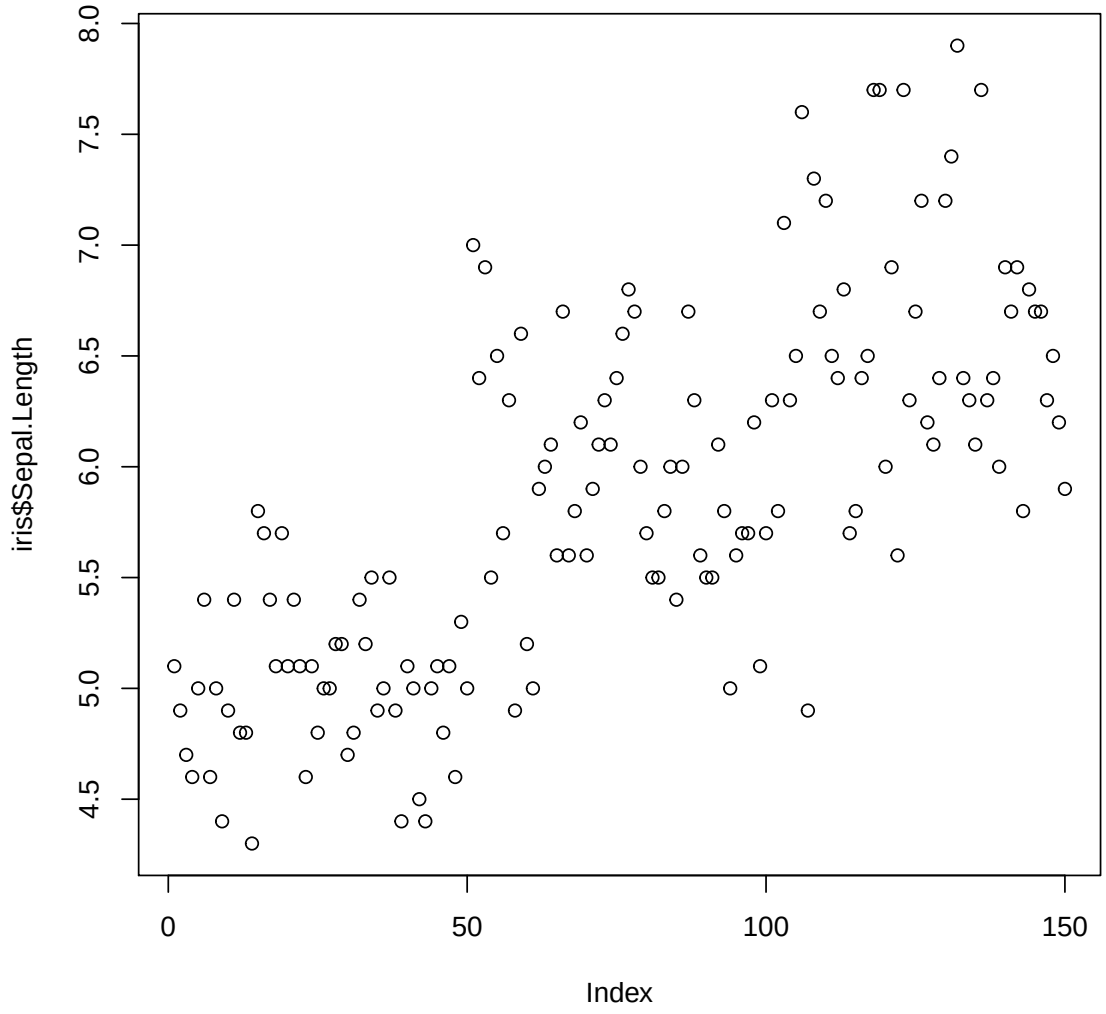
```
> min(c(3,NA,5),na.rm=T)

## [1] 3
```

27.2 Plot

Pour obtenir une représentation graphique de la variable, il suffit de taper :

```
plot(iris$Sepal.Length)
```

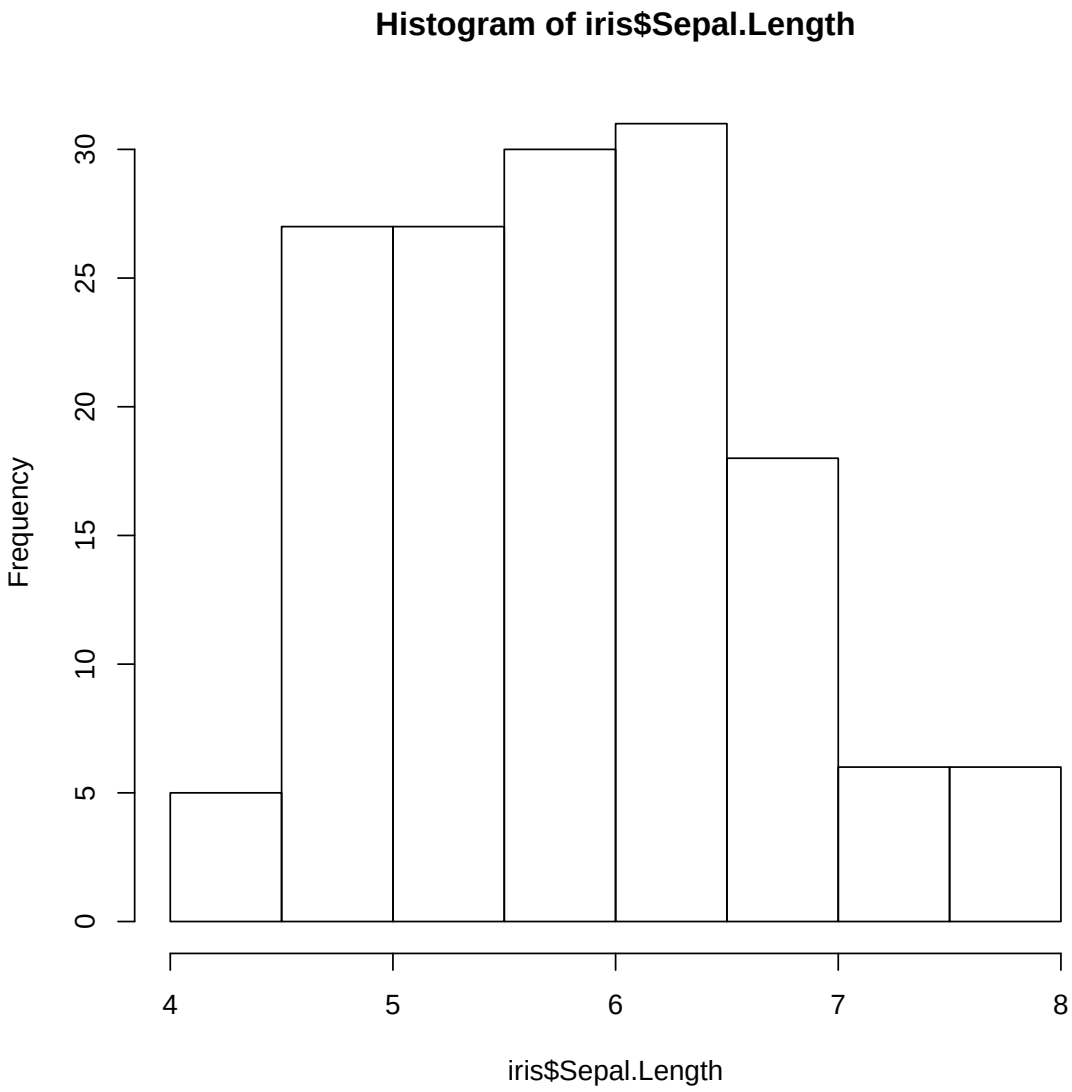
Dans ce cas les valeurs de la variable sont en ordonnées et les abscisses sont les numéros d'observation.



27.3 Histogramme

Pour obtenir un histogramme, il suffit de taper :

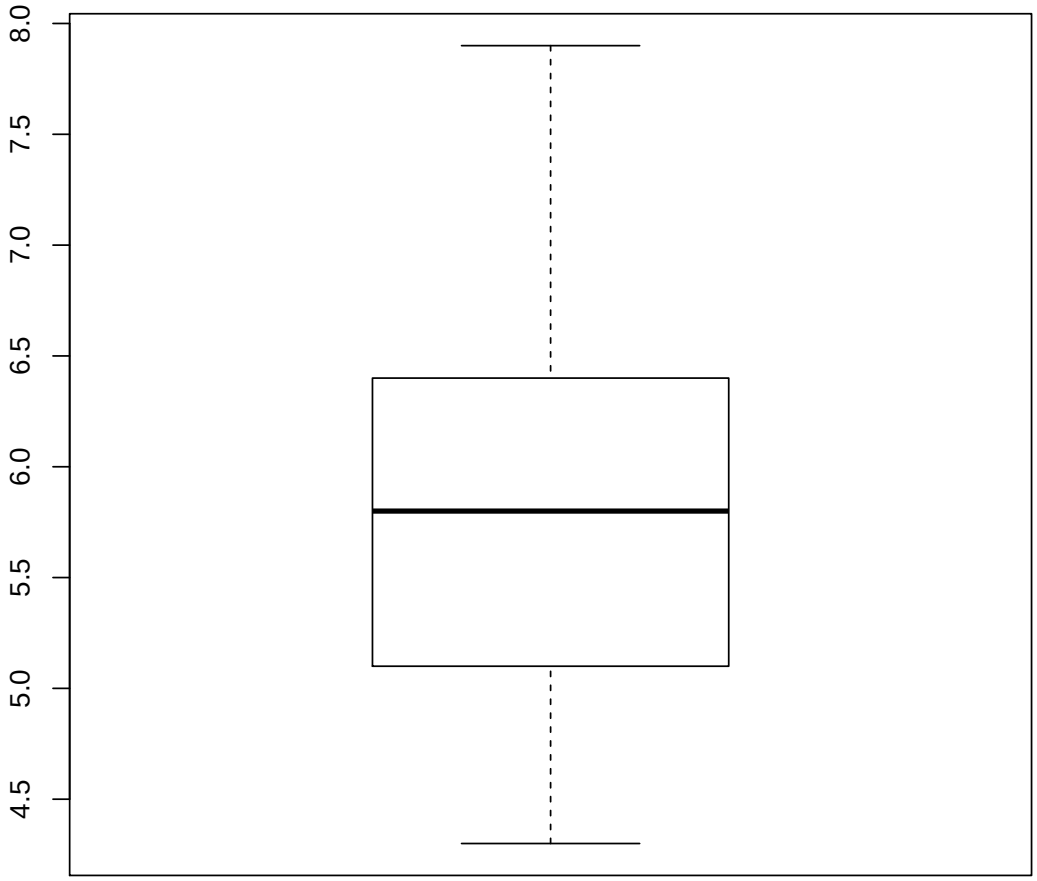
```
hist(iris$Sepal.Length)
```



27.4 Boxplots

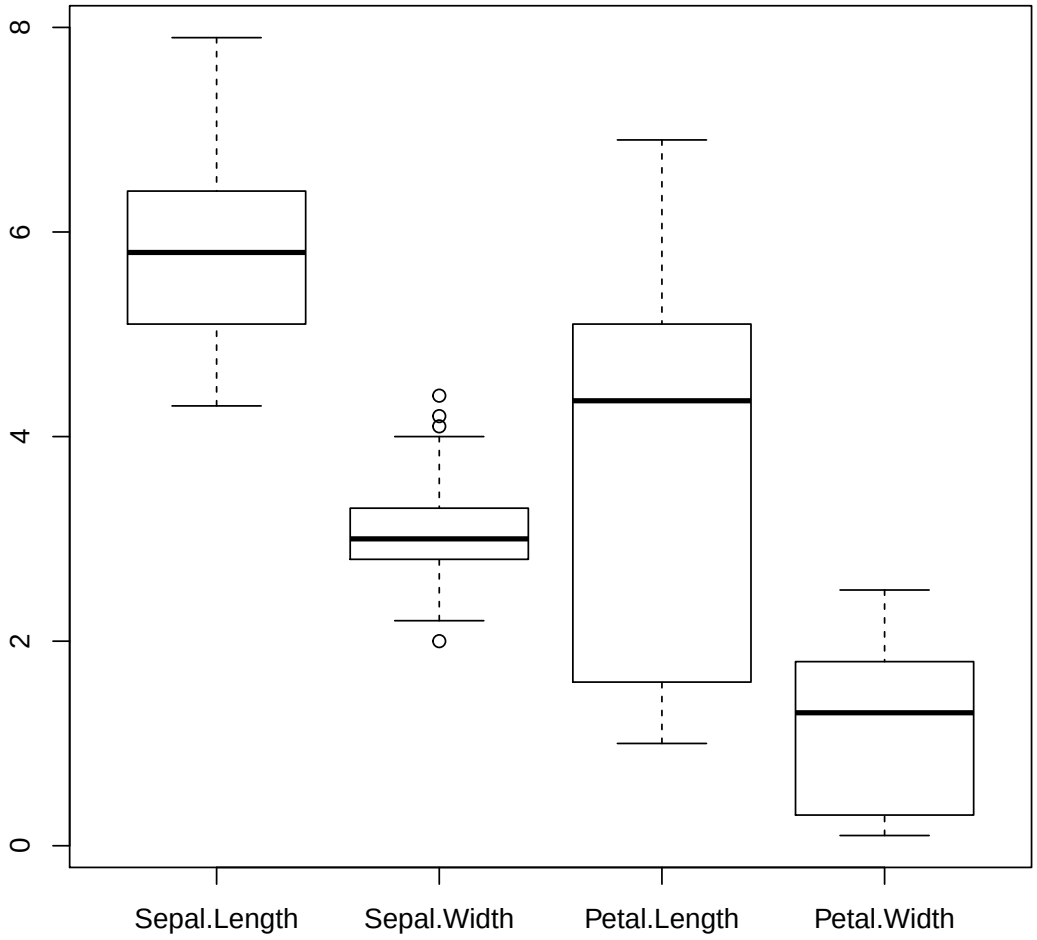
Pour obtenir un boxplot, il suffit de taper :

```
boxplot(iris$Sepal.Length)
```



Pour obtenir des boxplots de plusieurs variables d'une même *data.frame*, il suffit de taper :

```
boxplot(iris[,1:4])
```



27.5 Corrélations

Pour analyser la covariance ou la corrélation de deux variables (ou plus), il suffit d'appeler la fonction *cov* ou *cor* avec les deux variables.

```
> cor(cg$score,cg$plaisir)

## [1] NA
```

Les méthodes disponibles pour le calcul de la corrélation sont celles de Pearson, Kendall et Spearman.

La gestion des valeurs manquantes est particulière dans le cas des corrélations.

En effet on peut avoir plusieurs cas de figures, pour reprendre l'aide de R.

Dans la majorité des cas on utilisera le *use="pairwise.complete.obs"* qui permet d'obtenir une valeur de corrélation en présence de valeurs manquantes. Dans ce cas, la corrélation est calculée pour tous les couples de valeurs ne contenant pas de *NA*.

Les alternatives sont de rendre une erreur en présence de *NA* ou de supprimer toutes les observations contenant une valeur manquante.

```
> cor(cg$score,cg$plaisir,use="pairwise.complete.obs")

## [1] 0.1492273
```

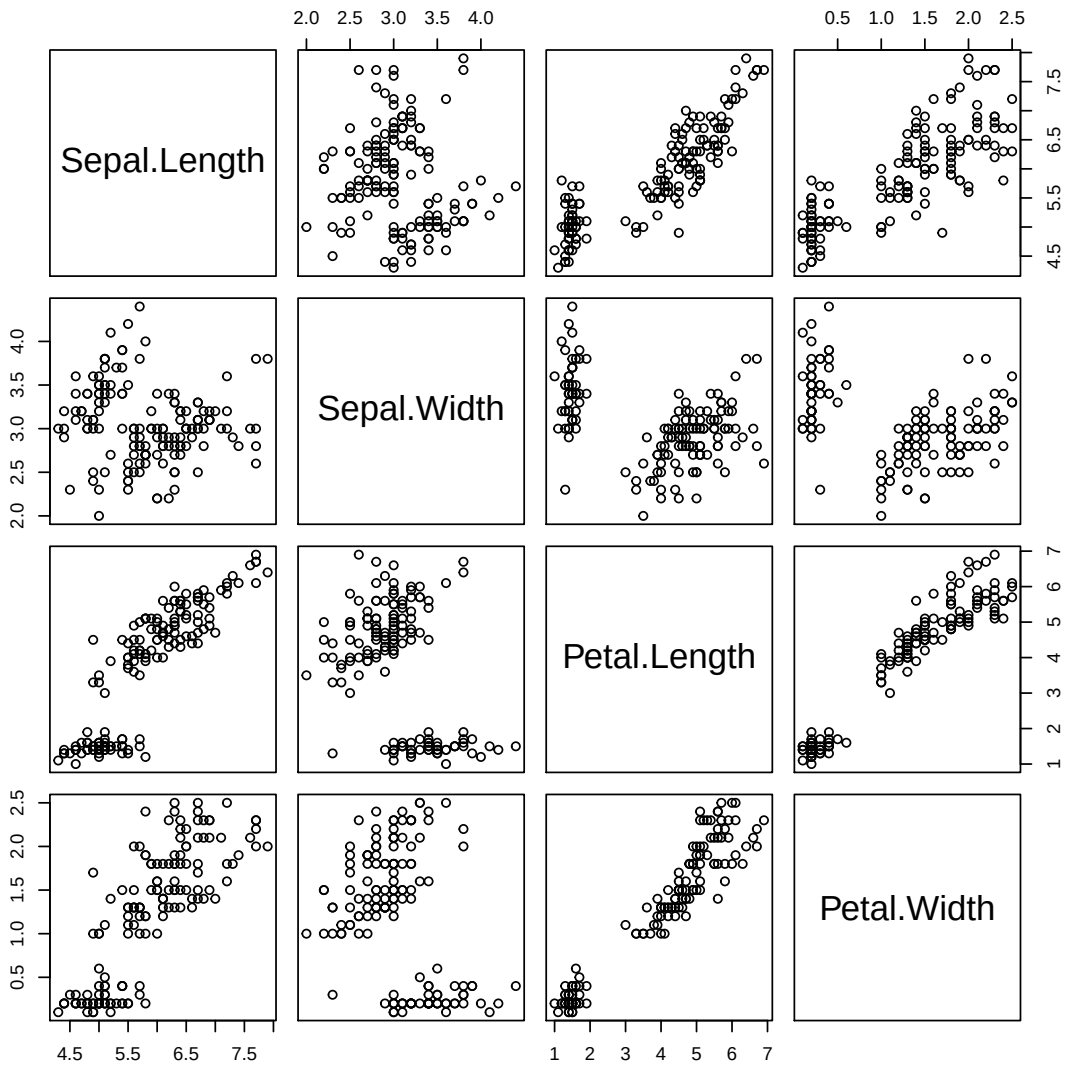
En fait, une matrice de corrélations peut être créée en utilisant la propriété de *cor* de se comporter différemment face à un objet de type *data.frame*.

```
> cor(cg[,8:12],use="pairwise.complete.obs")

##                 score    plaisir performance
## score       1.0000000 0.14922727  0.46193763
## plaisir     0.1492273 1.00000000  0.31990943
## performance 0.4619376 0.31990943  1.00000000
## estime_soi  0.1203776 0.12368912  0.42288147
## anx_sociale 0.1236460 0.08192793  0.04367632
##             estime_soi anx_sociale
## score        0.1203776  0.12364597
## plaisir      0.1236891  0.08192793
## performance  0.4228815  0.04367632
## estime_soi   1.0000000 -0.17216954
## anx_sociale -0.1721695  1.00000000
```

27.6 Corrélations - Graphiques

On peut également passer une *data.frame* à *plot* pour avoir une table des corrélations graphiques.

```
plot(iris[,1:4])
```



28 Exemple de test statistique

28.1 Test de corrélation

Le but est ici de vous illustrer un test pour vous montrer l'utilisation des objets.

```
> cor.test(cg$score,cg$plaisir,use="pairwise.complete.obs")

##
##  Pearson's product-moment correlation
##
## data:  cg$score and cg$plaisir
## t = 22.511, df = 22250, p-value <
## 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.1363555 0.1620487
## sample estimates:
##       cor
## 0.1492273
```

Une sortie des résultats apparait mais comme la plupart des fonctions, la fonction *cor.test* renvoie aussi un objet silencieusement. Comme il n'y a pas d'affectation R devine qu'on veut l'imprimer. Mais si on change la syntaxe :

```
> montest <- cor.test(cg$score,cg$plaisir,use="pairwise.complete.obs")
```

On peut accéder aux valeurs en utilisant directement la fonction *str* qui permet de connaître la structure des objets.

```
> str(montest)

## List of 9
##  $ statistic  : Named num 22.5
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named int 22250
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 5.56e-111
##  $ estimate   : Named num 0.149
##   ..- attr(*, "names")= chr "cor"
##  $ null.value : Named num 0
##   ..- attr(*, "names")= chr "correlation"
##  $ alternative: chr "two.sided"
##  $ method     : chr "Pearson's product-moment correlation"
##  $ data.name  : chr "cg$score and cg$plaisir"
##  $ conf.int   : atomic [1:2] 0.136 0.162
##   ..- attr(*, "conf.level")= num 0.95
##  - attr(*, "class")= chr "htest"
```

Par exemple, pour récupérer la corrélation et son intervalle de confiance :

```
> montest$estimate

##       cor
## 0.1492273

> montest$conf.int

## [1] 0.1363555 0.1620487
## attr(,"conf.level")
## [1] 0.95
```

28.2 Autres tests

Les tests disponibles sous R sont nombreux dans les packages de base et incalculables si on tient compte des paquets : Student, Wilcoxon, χ^2, Bartlett, Fisher, ...

Tous, ou presque, fonctionnent sur le même principe.

29 Statistiques descriptives : variables qualitatives

29.1 Les données patient

Ce sont des données recoltées par une pharmacienne sur dossiers dans deux hopitaux.

Les données portent sur le traitement de la douleur chez l'enfant. Le but est de déterminer la qualité de la prise en charge de la douleur dans ces hopitaux.

Pour cela les évaluations de la douleur se font sur une échelle de 0 à 100.

Les pathologies (CIM2) concernées sont codées de 1 à 4, de plus en plus grave (appendicite à arthrodèse).

Variable	Description
UID	Identifiant
Hopital	Hopital (A ou B)
Sexe	Sexe du patient
Poids	Poids du patient
Vitaux	Nombres de prises des signes vitaux
CIM2	Pathologie (de 1 à 4, de plus en plus grave)
age	Âge du patient
dureeopmin	Durée de l'opération en minutes
postopj	Nombres de jours post-opératoires
scoliose	Présence d'une scoliose
drepano	Enfant souffrant d'une drépanocytose/sphérocytose
ACP	Pompe à morphine administrée
peridurale	Injection péridurale
periACP	Pompe à morphine administrée en péridurale
nbttt	Nombre de traitements contre la douleur
totalechelle	Total des échelles de la la douleur
nbechelle	Nombre de prises d'échelles de douleur

29.2 Tableaux de contingence

Il suffit d'utiliser la fonction *table*.

```
> patient <- read.csv2("data/patient.csv")
> table(patient$sexe)

##
##  Feminin Masculin
##      101       75
```

Ces tableaux peuvent être à 2 niveaux :

```
> table(patient$sexe,patient$CIM2)

##
##             1  2  3  4
##   Feminin  18 24 18 41
##   Masculin 26 22 18  9
```

Comme les colonnes ne sont pas nommées cela peut être difficile de s'y retrouver si les variables ont les mêmes modalités.

Il suffit dans ce cas là de modifier la syntaxe :

```
> table(Sexe=patient$sexe,CIM2=patient$CIM2)

##           CIM2
## Sexe        1  2  3  4
##   Feminin  18 24 18 41
##   Masculin 26 22 18  9
```

29.3 Exportation de tableaux de contingence

Pour l'exportation des tableaux, elle peut se faire avec :

- *xtable*
- la fonction clipcopy de *questionr*
- sous forme de CSV

Dans ce dernier cas, l'aspect de la table exportée sera donc différent car il y a une conversion vers une *data.frame*.

```
> as.data.frame(table(Sexe=patient$sexe,CIM2=patient$CIM2))

##       Sexe CIM2 Freq
## 1  Feminin    1   18
## 2 Masculin    1   26
## 3  Feminin    2   24
## 4 Masculin    2   22
## 5  Feminin    3   18
## 6 Masculin    3   18
## 7  Feminin    4   41
## 8 Masculin    4    9
```

29.4 Exportation de tableaux de contingence (ancienne version)

Il y a une astuce pour contourner le problème. La classe *table* a la même structure interne qu'une *matrix*. Alors on berne R.

```
> tableau <- (table(Sexe=patient$sexe,CIM2=patient$CIM2))
> class(tableau)

## [1] "table"
```

```
> class(tableau) <- "matrix"
> class(tableau)

## [1] "matrix"
```

29.5 Exportation de tableaux de contingence (version correcte)

Depuis la première version du cours, il y a maintenant un contournement dans R :

```
> tableau <- (table(Sexe=patient$sexe,CIM2=patient$CIM2))
> as.data.frame.matrix(tableau)

##           1  2  3  4
## Feminin  18 24 18 41
## Masculin 26 22 18  9
```

29.6 Marges sur les tableaux

On peut ajouter les marges avec la fonction addmargins :

```
> addmargins(tableau,1)

##           CIM2
## Sexe        1  2  3  4
##   Feminin  18 24 18 41
##   Masculin 26 22 18  9
##   Sum      44 46 36 50
```

```
> addmargins(tableau,2)

##           CIM2
## Sexe         1   2   3   4 Sum
##   Feminin   18  24  18  41 101
##   Masculin  26  22  18   9  75
```

```
> addmargins(tableau,1:2)

##           CIM2
## Sexe         1   2   3   4 Sum
##   Feminin   18  24  18  41 101
##   Masculin  26  22  18   9  75
##   Sum       44  46  36  50 176
```

29.7 Tableau de proportion

Cette fois c'est la fonction *prop.table* qu'on appelle en utilisant une table déjà créée :

```
> prop.table(tableau,1)

##           CIM2
## Sexe              1         2         3
##   Feminin  0.1782178 0.2376238 0.1782178
##   Masculin 0.3466667 0.2933333 0.2400000
##           CIM2
## Sexe              4
##   Feminin  0.4059406
##   Masculin 0.1200000
```

29.8 Proportions dans les tableaux

```
> prop.table(tableau,2)

##           CIM2
## Sexe              1         2         3
##   Feminin  0.4090909 0.5217391 0.5000000
##   Masculin 0.5909091 0.4782609 0.5000000
##           CIM2
## Sexe              4
##   Feminin  0.8200000
##   Masculin 0.1800000
```

On peut combiner les deux...

```
> addmargins(prop.table(tableau),1:2)

##           CIM2
## Sexe                1          2          3
##   Feminin  0.10227273 0.13636364 0.10227273
##   Masculin 0.14772727 0.12500000 0.10227273
##   Sum      0.25000000 0.26136364 0.20454545
##           CIM2
## Sexe                4        Sum
##   Feminin  0.23295455 0.57386364
##   Masculin 0.05113636 0.42613636
##   Sum      0.28409091 1.00000000
```

29.9 Autres fonctions

Les autres fonctions dans les tableaux de base sont *ftable*, *tabulate*, *xtabs*, ... Chacun a sa petite spécificité.

Dans les packages, il existe pléthore de fonctions pour calculer des tableaux croisés. Notamment des versions qui permettent d'ajuster la présentation comme dans *questionr*.

30 Concaténation de données (merge)

30.1 A ne pas faire (ou avec prudence)

R permet de concaténer des lignes (*rbind*), des colonnes (*cbind*), des *data.frames* (mêmes fonctions) ensemble. Toutefois il convient d'utiliser avec sagesse cette fonctionnalité.

Si en sciences expérimentales faire une fusion de table avec une simple concaténation est très pratique, cette opération n'est pas raisonnable sur des tables plus complexes et surtout sur des tables contenant des identifiants qui permettent de réaliser une fusion plutôt qu'une concaténation.

En tout cas dès que *cbind* est utilisé il faut vérifier :

- que les deux tableaux ont la même taille
- chaque ligne identifie une observation
- que les observations sont strictement dans le même ordre dans les deux tableaux

En tout cas dès que *rbind* est utilisé il faut vérifier :

- que le nombre de colonnes sont identiques
- que le type de chaque colonnes sont identiques

rbind est un peu plus sûr car R généralement refuse d'opérer en cas de différence de noms et/ou de types de variables dans les deux tableaux de données.

rbind s'avère quand même pratique si on souhaite travailler par exemple sur une base public et une privé... et réassembler le tout à la fin du traitement.

C'est typiquement le cas par exemple quand on utilise *split*.

30.2 Fusion avec une seule variable

Ce cas est en fait beaucoup plus fréquent qu'il n'y parait. On veux par exemple ajouter une variable avec une couleur pour les graphiques, le nombre d'élèves dans l'établissement, ...

Et ce type de fusion se fait avec un vecteur.

```
> couleurs <- c( "red", "green", "blue" )
> names(couleurs) <- levels(iris$Species)
> iris$couleur <- couleurs[as.character(iris$Species)]
> with( iris, table(couleur,Species) )

##        Species
## couleur setosa versicolor virginica
##   blue       0          0        50
##   green      0         50         0
##   red       50          0         0
```

Un exemple numérique, si on veut ajouter la longueur moyenne par espèce pour les orchidées :

```
> longueur_par_spe <- tapply( iris$Sepal.Length, iris$Species, mean )
> iris$Sepal.Length.Moy <- longueur_par_spe[as.character(iris$Species)]
> with( iris, table(Sepal.Length.Moy,Species) )
```

```
##                 Species
## Sepal.Length.Moy setosa versicolor virginica
##            5.006     50          0         0
##            5.936      0         50         0
##            6.588      0          0        50
```

30.3 Fusions avec merge

La fonction *merge* dans R permet de fusionner des tables avec un identifiant (clef) commun entre les tables.

La fusion peut être réalisée en utilisant des variables *factor* mais il est préférable de les transformer variable *character* avant la fusion.

Les fusions possibles sont des fusions de 1 à 1 ou de 1 à n.

x, y	les 2 *data.frames* que l'on veut fusionner
by	si la variable porte le même nom dans les deux *data.frame*, il suffit de préciser le nom de la variable précédé de *by*
by.x, by.y	dans ce cas on spécifie le nom de la colonne pour x (la première *data.frame* et pour y (la deuxième).

TABLE 2 – Les principaux arguments de *merge*

Voilà l'essentiel de la fonction.

Il faut noter qu'on a la possibilité de fusionner les tables non pas en utilisant le nom d'une variable de la *data.frame* mais les *row.names*. Dans ce cas, l'argument que l'on passe à *by* est *'row.names'*.

```
> res <- merge( eleves, scores, by="id" )
> dim(res)

## [1] 5000    7
```

Dans le cas de l'utilisation des rownames :

```
> rownames(eleves) <- eleves$id
> rownames(scores) <- scores$id
>
> res <- merge( eleves, scores, by="row.names" )
> dim(res)

## [1] 5000    9
```

Après la fusion, la fonction utile est *dim* qui donne le nombre de lignes et de colonnes :

```
> dim(eleves);dim(scores);dim(res)

## [1] 5000    6
## [1] 5000    2
## [1] 5000    9
```

all	Si vrai alors toutes les lignes des deux *data.frame* seront conservées dans la *data.frame* finale.
all.x	Si *TRUE* alors toutes les lignes de la *data.frame* x seront conservées dans la *data.frame* finale. Les lignes de y ne trouvant pas de correspondance seront éliminées.
all.y	Si *TRUE* alors toutes les lignes de la *data.frame* x seront conservées dans la *data.frame* finale. Les lignes de y ne trouvant pas de correspondance seront éliminées.

TABLE 3 – Le type de jointure

```
> colnames(res)

## [1] "Row.names" "id.x"      "sexe"
## [4] "age3e"     "retard"    "secteur"
## [7] "acad"      "id.y"      "score"
```

La fonction *merge* effectue une jointure naturelle. C'est-à-dire que seules les lignes présentes dans x et dans y seront présentes dans la *data.frame* finale.

Pour changer ce comportement, il existe trois arguments *all*

Jointure naturelle

```
> conatif <- read.csv2( "data/evaluation-conatif.csv" )
> conatif$id <- as.character( conatif$id )
> dim(conatif)

## [1] 4987    8

> ec <- merge( eleves, conatif, by="id" )
> dim(ec)

## [1] 4987   13
```

Fusion à gauche

```
> ec <- merge( eleves, conatif, by="id", all.x = T )
> dim(ec)

## [1] 5000   13
```

Si des colonnes de x et de y portent le même nom, les colonnes provenant de x seront suffixés avec x. Et pour y, la colonne sera suffixés par y.

Il est possible de spécifier des suffixes personnalisés plutôt que ces suffixes par défaut avec l'argument *suffixes*.

Il attends un vecteur *character* de longueur 2 comme par exemple...

```
> res <- merge( eleves, scores, by="row.names",
+               suffixes=c(".eleves",".scores" )
+               )
> dim(res)
```

```
## [1] 5000    9

> colnames(res)

## [1] "Row.names" "id.eleves" "sexe"
## [4] "age3e"     "retard"    "secteur"
## [7] "acad"      "id.scores" "score"
```

Pour trouver les lignes qui n'ont pas été importées...

La syntaxe et très simple et fait appel à l'opérateur *%in%*.

Ici on cherche les lignes, de *eleves* pour lesquelles il n'y a pas de données pour la partie conative.

```
> res <- merge( eleves, conatif, by="id", all.x=T )
> (perdus <- res$id[ !(res$id %in% conatif$id) ])

##  [1] "e014161" "e03592"  "e041612" "e044139"
##  [5] "e1123"   "e121165" "e151894" "e162289"
##  [9] "e184897" "e213974" "e242770" "e243719"
## [13] "e251862"
```

Il est possible de spécifier un vecteur de noms de variables pour l'argument *by*.

Mais les identifiants composite ne sont pas conseillés (dans l'absolu).

31 Un mot sur les fonctions...

31.1 R langage fonctionnel

R est un langage fonctionnel. Si cela signifie que "tout est fonction dans R", cela signifie également qu'il faut privilégier le traitement des vecteurs au détriment des boucles.

Au début cela peut paraître contre-intuitif mais cela permet souvent de gagner en vitesse d'exécution, en possibilité de rendre le calcul parallèle et en lisibilité (si si...).

Par exemple, sur un vecteur, il doit vous paraître évident que :

```
> x <- 1:4
> x*4

## [1]  4  8 12 16

> # n'est autre que l'équivalent implicite de
> vreponse <- c()
> for ( ii in 1:length(x) ) vreponse <- c( vreponse, x[ii]*4 )
> vreponse

## [1]  4  8 12 16
```

L'utilisation et la production de statistiques va en grande partie utilisé ce principe illustré ici par un vecteur mais qui est utilisé dans les fonctions de type apply sur des objets plus complexes.

Ici on utilise l'opérateur de multiplication qui est une fonction parmi d'autres.

```
> "*"(3,4)

## [1] 12
```

Cela oblige à savoir utiliser les fonctions sous R. La définition se fait avec la syntaxe suivante :

```
> mafonction <- function ( arg1, arg2, arg3=F, ... ) {
+   # code
+ }
```

Mais souvent, dans les opérations de manipulations de données, des fonctions *anonymes* seront utilisées.

C'est-à-dire directement des fonctions : sans nom, jetables.

Cela ressemble à ça par exemple :

```
> apply(iris[1:4],2,function(x){
+   c(mean(x),sd(x))
+ })

##      Sepal.Length Sepal.Width Petal.Length
## [1,]    5.8433333   3.0573333     3.758000
## [2,]    0.8280661   0.4358663     1.765298
##      Petal.Width
## [1,]   1.1993333
## [2,]   0.7622377
```

32 Aggrégation de données

32.1 Considérations sur les aggrégations

Contrairement à d'autres logiciels, R peut paraître strict voire pénible lors des aggrégations. En fait, la pratique de R permet de réaliser que R impose cette syntaxe notamment pour éviter de réaliser des regroupements n'ayant pas de sens.

Une exemple simple, cette requête SQL peut tout à fait renvoyer un résultat valide :

```
SELECT *, uai FROM base_eleves GROUP BY uai ;
```

Hors le sexe de l'élève, présent dans la ligne élève, va devenir une variable vide de sens. En effet elle a un sens au niveau individuel mais pas au niveau d'un établissement.

R va rendre difficile ce type d'aggrégation.

L'aggrégation ne sera possible que si on obtient un vecteur cohérent avant aggrégation.

32.2 Pourquoi aggréger ?

Il existe de nombreuses façons d'aggréger des données sous R. L'utilisation de chacune dépend des goûts de chacun et surtout de la finalité de l'aggrégation.

Par exemple, l'aggrégation peut servir à...

- créer un enregistrement pour constituer un unité plus grande que celle d'origine (ex : passer élève à établissement)
- créer des statistiques pour des unités plus importantes (ex : établissement, pays, ...)
- ...

32.3 Aggrégations statistiques

Beaucoup de statistiques peuvent réalisées avec certaines fonctions de R qui appartiennet à la famille *apply*.

Par exemple, *tapply* permet de réaliser des regroupements en fonction d'une ou plusieurs variables en calculant des statistiques sur une variable.

```
> res <- with( xtfme, tapply( vali_f, num_etab, mean, na.rm=T ) )
> res[1:5]

##  0010529V  0010560D  0011110B  0011238R  0011289W
## 0.9500000 0.9444444 0.6521739 0.8709677 0.8750000
```

Dans le cas précédent, on demande la moyenne (vecteur de longueur 1) et un variable de regroupement. Mais *tapply* permet de faire des choses plus complexes. Dans ce cas, il y a un résultat par croisement de modalité. Ce qui donne un tableau.

```
> with( xtfme, tapply(
+   vali_f,
+   list( strate=strate, sexe=sexe ),
+   mean,
+   na.rm=T
+   )
+ )

##       sexe
## strate         1         2
##      1 0.8695652 0.9317269
##      2 0.7598647 0.8289183
##      3 0.6610360 0.7807487
##      4 0.8760246 0.9536935
```

C'est la limite (ou la puissance) de tapply. On peut ainsi s'amuser à obtenir des tableaux à k dimensions pour k variables de regroupement.

Inversement on peut être limité par le nombre de valeurs renvoyées par la fonction de calcul.

```
> with( xtfme, tapply(
+   vali_f,
```

```
+   list(strate=strate),
+   function(x,na.rm=T) {
+     c( mean(x,na.rm=na.rm), sd(x,na.rm=na.rm) )
+     }
+   )
+ )
```

```
## $`1`
## [1] 0.9003984 0.2995427
##
## $`2`
## [1] 0.7947574 0.4039915
##
## $`3`
## [1] 0.7224355 0.4479202
##
## $`4`
## [1] 0.9134360 0.2812698
```

Deux illustrations pour calculer le nombre d'élèves dans chaque strate :

```
> with(xtfme, tapply( rep(1,length(strate)), strate, sum) )

##    1    2    3    4
## 2008 1793 1823 1883

> with(xtfme, tapply( num_etab, strate, length) )

##    1    2    3    4
## 2008 1793 1823 1883
```

et pour les poids...

```
> with(xtfme, tapply( poids, strate, sum) )

##      1      2      3      4
## 5220.8  537.9  182.3  941.5
```

La fonction *aggregate* permet des choses similaires ou un peu plus complexes.

```
> with( xtfme, aggregate(
+   cbind(vali_f, vali_m),
+   list(strate=strate),
+   mean
+   )
+ )

##   strate    vali_f    vali_m
## 1      1 0.9003984 0.9213147
## 2      2 0.7947574 0.8315672
## 3      3 0.7224355 0.7761931
## 4      4 0.9134360 0.9373340
```

```
> with( xtfme, aggregate(
+   cbind(vali_f, vali_m),
+   list(strate=strate,sexe=sexe),
+   mean
+   )
+ )

##   strate sexe    vali_f    vali_m
## 1      1    1 0.8695652 0.9268775
## 2      2    1 0.7598647 0.8410372
## 3      3    1 0.6610360 0.7815315
## 4      4    1 0.8760246 0.9344262
## 5      1    2 0.9317269 0.9156627
## 6      2    2 0.8289183 0.8222958
## 7      3    2 0.7807487 0.7711230
## 8      4    2 0.9536935 0.9404631
```

Les indications de variables peuvent se faire en formule.

```
> aggregate(
+   vali_m ~ strate + sexe,
+   data=xtfme, mean
+   )

##   strate sexe    vali_m
## 1      1    1 0.9268775
## 2      2    1 0.8410372
## 3      3    1 0.7815315
## 4      4    1 0.9344262
## 5      1    2 0.9156627
## 6      2    2 0.8222958
## 7      3    2 0.7711230
## 8      4    2 0.9404631
```

Ou comme dans l'aide...

```
> data(iris)
> aggregate( . ~ Species, data=iris, mean, na.rm=T  )

##      Species Sepal.Length Sepal.Width
## 1     setosa        5.006       3.428
## 2 versicolor        5.936       2.770
## 3  virginica        6.588       2.974
##   Petal.Length Petal.Width
## 1        1.462       0.246
## 2        4.260       1.326
## 3        5.552       2.026
```

32.4 plyr

Un paquet de Hadley Wickham, *plyr*, permet de réaliser ce type d'opérations assez facilement.

Le paquet *plyr* permet de traiter des *array*, des vecteurs, des *data.frames*.

Il offre des fonctions génériques permettant de créer, transformer ou faire des calculs sur des *data.frames*.

Outre ce côté générique, il offre quelques avantages sur les fonctions de base.

Par exemple les statistiques sur une variable deviennent :

```
> ddply( xtfme, .(strate), summarize,
+        moy_f=mean(vali_f), sd_f=sd(vali_f),
+        moy_m=mean(vali_m), sd_m=sd(vali_m)
+        )

##   strate     moy_f      sd_f     moy_m      sd_m
## 1      1 0.9003984 0.2995427 0.9213147 0.2693140
## 2      2 0.7947574 0.4039915 0.8315672 0.3743546
## 3      3 0.7224355 0.4479202 0.7761931 0.4169085
## 4      4 0.9134360 0.2812698 0.9373340 0.2424255
```

On récupère une *data.frame* ...

Idem...

```
> head(
+   ddply( xtfme, .(strate,sexe), summarize,
+        moy_f=mean(vali_f), sd_f=sd(vali_f),
+        moy_m=mean(vali_m), sd_m=sd(vali_m)
+        ), 4
+ )

##   strate sexe     moy_f      sd_f     moy_m
## 1      1    1 0.8695652 0.3369477 0.9268775
## 2      1    2 0.9317269 0.2523407 0.9156627
## 3      2    1 0.7598647 0.4274065 0.8410372
## 4      2    2 0.8289183 0.3767883 0.8222958
##        sd_m
## 1 0.2604662
## 2 0.2780327
## 3 0.3658477
## 4 0.3824747
```

etc...

32.5 aggrégation personnalisée

Une fonction d'aggrégation complètement personnalisée par exemple... avec les fonctions classiques

```
> agg <- lapply( split(xtfme,xtfme$num_etab), function(x) {
+       data.frame(
+         uai=unique(x$num_etab), vali_f=mean(x$vali_f),
+         vali_m=mean(x$vali_m), poids=sum(x$poids),
+         prop_garcons=mean(ifelse(x$sexe==1,0,1))
+       )
+     }
+   )
> head( do.call( rbind, agg), 3 )
```

```
##              uai    vali_f    vali_m poids
## 0010529V 0010529V 0.9500000 0.9500000  52.0
## 0010560D 0010560D 0.9444444 0.9444444  46.8
## 0011110B 0011110B 0.6521739 0.8260870   2.3
##          prop_garcons
## 0010529V    0.4500000
## 0010560D    0.5000000
## 0011110B    0.5217391
```

```
##              uai    vali_f    vali_m poids
## 0010529V 0010529V 0.9500000 0.9500000  52.0
## 0010560D 0010560D 0.9444444 0.9444444  46.8
## 0011110B 0011110B 0.6521739 0.8260870   2.3
##          prop_garcons
## 0010529V    0.4500000
## 0010560D    0.5000000
## 0011110B    0.5217391
```

Quelque chose de discutable d'un point de vue méthodologique mais possible...

```
> elev <- merge(scores,eleves,by="id")
> elev$sexe <- as.character(elev$sexe)
> elev <- elev[elev$sexe!="M",]
>
> coef <- function(score,age3e,n) { coef(lm( score ~ age3e ))[n] }
>
> res <- ddply(
+   elev, .( , secteur ), summarize,
+   coef1_age3e = coef1(score,age3e,1),
+   coef2_age3e = coef2(score,age3e,2)
+ )
```

33 Transposition

33.1 Transposition de matrices

La transposition simple d'une matrice ou d'une data.frame se fait avec la fonction t :

```
> (a = matrix( 1:16, nrow=4, ncol=4 ))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

```
> t(a)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

33.2 reshape2

Encore un paquet d'Hadley Wickham...
Il permet de faire un peu se qu'on veut au niveau des transpositions.
Les deux fonctions centrales sont *cast* et *melt*.

33.3 *melt*

Cette fonction permet de passer d'un tableau large à un tableau long...

```
> names(airquality) <- tolower(names(airquality))
> head(airquality)

##   ozone solar.r wind temp month day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

Cette fonction permet de passer d'un tableau large à un tableau long...

```
> head(
+   melt( airquality,
+         id=c("month", "day"),
+         measure.vars=c("ozone"),
+         na.rm=TRUE
+         )
+   )

##   month day variable value
## 1     5   1    ozone    41
## 2     5   2    ozone    36
## 3     5   3    ozone    12
## 4     5   4    ozone    18
## 6     5   6    ozone    28
## 7     5   7    ozone    23
```

On peux utiliser plusieurs variables comme variables de mesure.

```
> head(
+   z <- melt( airquality,
+         id=c("month", "day"),
+         measure.vars=c("wind","ozone"),
+         na.rm=TRUE
+         ), 3
+   )

##   month day variable value
## 1     5   1     wind   7.4
## 2     5   2     wind   8.0
## 3     5   3     wind  12.6

> table(z$variable)
```

```
##
##  wind ozone
##   153   116
```

33.4 *cast*

A l'inverse pour passer d'un tableau long à un tableau large...

```
> head(
+   dcast( z, month + day ~ variable )
+   )

##   month day wind ozone
## 1     5   1  7.4    41
## 2     5   2  8.0    36
## 3     5   3 12.6    12
## 4     5   4 11.5    18
## 5     5   5 14.3    NA
## 6     5   6 14.9    28
```

La fonction *(a/d)cast* peut également être utilisé pour réaliser des statistiques...

```
> head(
+   dcast( z, month ~ variable, mean, na.rm=T )
+   )

##   month      wind    ozone
## 1     5 11.622581 23.61538
## 2     6 10.266667 29.44444
## 3     7  8.941935 59.11538
## 4     8  8.793548 59.96154
## 5     9 10.180000 31.44828
```

33.5 dplyr, data.table, ...

Des paquets supplménetaires ont fait leur apparition ces dernières années.

Ils changent notablemment la syntaxe de R. Par exemple avec *dplyr* (ou à la *magrittr*), la séquence des commandes est inversée.

Trois formes d'écriture de la même opération avec le paquet *dplyr* (source : une des vignettes du paquet).

Version *dplyr* façon *plyr* :

```
> a1 <- group_by(flights, year, month, day)
> a2 <- select(a1, arr_delay, dep_delay)
> a3 <- summarise(a2,
+   arr = mean(arr_delay, na.rm = TRUE),
+   dep = mean(dep_delay, na.rm = TRUE))
> a4 <- filter(a3, arr > 30 | dep > 30)
```

Version *dplyr* sans variable intermédiaire :

```
> filter(
+   summarise(
+     select(
+       group_by(flights, year, month, day),
+       arr_delay, dep_delay
+     ),
+     arr = mean(arr_delay, na.rm = TRUE),
+     dep = mean(dep_delay, na.rm = TRUE)
+   ),
+   arr > 30 | dep > 30
+ )
```

Version *dplyr* (à la *magrittr*) :

```
> flights %>%
+   group_by(year, month, day) %>%
+   select(arr_delay, dep_delay) %>%
+   summarise(
+     arr = mean(arr_delay, na.rm = TRUE),
+     dep = mean(dep_delay, na.rm = TRUE)
+   ) %>%
+   filter(arr > 30 | dep > 30)
```

L'exemple plus haut deviendrait :

```
> agg2 <- xtfme %>% group_by(num_etab) %>%
+   summarize(vali_f=mean(vali_f),
+         vali_m=mean(vali_m), poids=sum(poids),
+         prop_garcons=mean(ifelse(sexe==1,0,1))
+   )

## Error in function_list[[i]](value): impossible de trouver la fonction "group_by"

> head( agg2 )

## Error in head(agg2): objet 'agg2' introuvable
```

L'écriture est pour certains plus intuitives (*dplyr*,*magrittr*. Le problème est que les objets manipulés ne sont pas tout à fait des objets standards de R (*dplyr* et *data.table*).

Si certaines tâches (manipulation de données en base de données, manipulation de données dans R, ...) sont beaucoup facilités et/ou accélérées par ces paquets, il semble plus raisonnable de comprendre comment fonctione le langage R avant de passer à ces outils.

Mais rapidement il faudra les maîtriser car :

1. ils sont pratiques
2. beaucoup de paquets et de codes circulent avec cette syntaxe
3. probablement l'avenir de R

L'autre avantage rest que dplyr permet de faire les opérations indiquées ci-dessus dans une base de données en traduisant le code R en code SQL et récupère ensuite les résultats de la requête.

Vous avez un exemple sur cette vignette.

34 Les fonctions de base

34.1 Les fonctions utilitaires

paste concatene des chaines (vectoriel)

nchar indique le nombre de caractères de la chaîne (vectoriel)

substr retourne une partie de la chaine (vectoriel)

toupper passe la casse en majuscules

tolower passe la casse en minusucles

chartr transforme des caractères en d'autres

34.2 Les fonctions utilitaires (*nchar*)

```
> nchar("12345")

## [1] 5

> nchar(c("121243","SDFSDGFG","GDFG","ERTERYTYU"))

## [1] 6 8 4 9
```

34.3 Les fonctions utilitaires (*substr*)

La fonction *substr* contrairement à d'autres langages extrait la chaîne en utilisant des positions dans la chaîne (et non une longueur).

Le premier argument est la chaîne et les deux arguments sont la position du début de la chaîne à extraire et le second (optionnel selon les alias, voir ?substr) la position de fin de la chaine à extraire.

```
> substr("abcd",2,3)

## [1] "bc"

> chaine <- c("121243","SDFSDGFG","GDFG","ERTERYTYU")
> substr(chaine,2,4)

## [1] "212" "DFS" "DFG" "RTE"

> substr(chaine,2,nchar(chaine)-2)

## [1] "212"    "DFSDG"  "D"      "RTERYT"
```

Comble du bonheur cette fonction peut être à droite ou à gauche d'une affectation...

```
> chaine <- c("121243","SDFSDGFG","GDFG","ERTERYTYU")
> substr(chaine,2,nchar(chaine)-2)
```

```
## [1] "212"    "DFSDG"  "D"      "RTERYT"

> substr(chaine,2,nchar(chaine)-2) <- "Mickey"
> chaine

## [1] "1Mic43"    "SMickeFG"  "GMFG"
## [4] "EMickeyYU"
```

34.4 Les fonctions utilitaires (*toupper*,*tolower*)

Les fonctions *toupper* et *tolower* ont un nom assez transparent...

```
> tolower(c("A","à","B", LETTERS[10:15]))

## [1] "a" "à" "b" "j" "k" "l" "m" "n" "o"

> toupper(c("A","à","B", letters[10:15]))

## [1] "A" "À" "B" "J" "K" "L" "M" "N" "O"
```

Il est à remarquer que R fait une conversion propre et les accents sont conservés lors du changement de casse.

34.5 La fonction *paste*

C'est la fonction de concaténation de R. Elle permet de concaténer et ce sur un vecteur.

```
> paste("Items",1:4)

## [1] "Items 1" "Items 2" "Items 3" "Items 4"

> paste("Items",1:4,sep="|")

## [1] "Items|1" "Items|2" "Items|3" "Items|4"

> paste0("Items",1:4)

## [1] "Items1" "Items2" "Items3" "Items4"
```

Elle est très plastique. L'alias *paste0* correspond à un séparateur vide.

L'argument collapse permet de concaténer un vecteur en un vecteur de longueur 1.

```
> paste( paste("Items",1:4), collapse="," )

## [1] "Items 1,Items 2,Items 3,Items 4"

> paste( paste("Items",1:4), sep=" ", collapse="," )

## [1] "Items 1,Items 2,Items 3,Items 4"
```

34.6 La fonction *sprintf*

Elle utilise la syntaxe de *sprintf* du langage C ANSI. Son utilisation permet de définir le format en sortie par exemple pour les chiffres (blancs ou zéros devant, etc...).

```
> sprintf( "Il y a %02i items dans la base", 1 )

## [1] "Il y a 01 items dans la base"

> sprintf( "%02i %02i %02i %02i", 1, 3, 15, 09 )

## [1] "01 03 15 09"

> sprintf( "%08.5f", 1.2351 )

## [1] "01.23510"
```

34.7 La fonction *iconv*

Comment se débarasser des caractères ? La solution la plus simple serait de transformer un à un tous les lettres accentuées avec *chartr* par exemple. Possible. Mais une solution existe avec une outil puissant venu des outils GNU.

La fonction en question s'appelle *iconv*.

Son utilisation première est de transformer un vecteur *character* dans un format, par exemple Unicode, en format Latin1 par exemple.

Cela s'écrit...

```
> chaine_convertie <- iconv( chaine_brute, from="UTF8", to="Latin1" )
```

Cela permet d'utiliser R pour transformer aisément un fichier en Unicode en Latin1 par exemple.

```
> r <- scan("Fiches_Fonctions_Character.Rnw",what=character(),sep="\n")
> r <- iconv( r, from="UTF8", to="Latin1//TRANSLIT" )
> write( r, file = "Fiches.txt", sep = "\n")
```

34.8 La fonction *iconv* ou comment convertir les accents

Mais en passant d'un type de codage à un autre certains caractères peuvent disparaître car ils ne sont pas disponibles dans le type de caractères final.

C'est par exemple le cas pour les caractères accentués en ASCII, ils n'existent pas.

```
> accents <- c("à", "è", "ì", "ò", "ù" , "À", "È", "Ì", "Ò", "Ù" )
> iconv( accents, to="ASCII" )

##  [1] NA NA NA NA NA NA NA NA NA NA
```

On voit que des valeurs manquantes apparaissent.

Mais cette fonction a *en mémoire* certaines conversions possibles. Il suffit d'un petit ajout...

```
> accents <- c("à", "è", "ì", "ò", "ù" , "À", "È", "Ì", "Ò", "Ù" )
> iconv( accents, to="ASCII//TRANSLIT" )

##  [1] "a" "e" "i" "o" "u" "A" "E" "I" "O" "U"
```

Mais peut être trop puissant... Que faire avec ces vieux logiciels qui ne supporte pas les lettres majuscules accentuées ?

```
> accents <- c("à", "è", "ì", "ò", "ù" , "À", "È", "Ì", "Ò", "Ù" )
>
> accents[accents==toupper(accents)] <- iconv(
+   accents[accents==toupper(accents)],
+   to="ASCII//TRANSLIT"
+   )
>
> accents

##  [1] "à" "è" "ì" "ò" "ù" "A" "E" "I" "O" "U"
```

Mais vous aviez deviné...

35 Les fonctions avancées

35.1 Introduction

Ces fonctions, toujours dans les fonctions de base, ont la particularité d'utiliser une recherche de motif. Ces fonctions sont :

- strsplit
- grep & al.
- sub & al.

Ces fonctions utilisent par défaut un type de motif *dynamique*, les expressions régulières.

Les expressions régulières sont une arme redoutable pour traiter du texte.

Expliquer les expressions régulières dépassent le cadre de ce cours. De nombreuses ressources sont en ligne et l'auteur recommande un livre *Mastering regular expressions* de Jeffrey Friedl aux éditions O'Reilly. La troisième édition est recommandée mais est disponible seulement en anglais. La seconde est quant à elle disponible en français.

Les quelques diapos qui suivent illustrent l'utilisation qui peuvent en être fait.

Supprimer les blancs...

```
> a = "     lkdfsjkfhsjkfh  kflgfhdgkjfdghdkjgghd     sdfhsdjkfhsjkfhf   "
> (a <- gsub("\\s+"," ",a))

## [1] " lkdfsjkfhsjkfh kflgfhdgkjfdghdkjgghd sdfhsdjkfhsjkfhf "
```

```
> (a <- gsub("^(\\s+)|(\\s+)$","",a))

## [1] "lkdfsjkfhsjkfh kflgfhdgkjfdghdkjgghd sdfhsdjkfhsjkfhf"
```

Vérification d'UAI.

```
Monsieur,

Les collèges 0030035X, 1030021G et 0030021G ne souhaitent
pas participer à l'enquête que vous organisez.
De plus le matériel d'évaluation n'est pas arrivé dans les
collèges 0021826Z et 0020467X

Cordialement,
```

	uai	dep	num	hash	OK
1	0030035X	003	0035	X	TRUE
2	1030021G	103	0021	G	FALSE
3	0030021G	003	0021	G	TRUE
4	0021826Z	002	1826	Z	TRUE
5	0020467X	002	0467	X	TRUE

35.2 Fonctions avancées (*strsplit*)

Si vous ne maitrisez pas les expressions régulières, ces fonctions acceptent comme un argument *fixed* qui indique que le motif que vous recherchez n'est pas une expression régulière.

Mais attention dans ce cas les motif sont casse dépendant. "A" et "a" ne sont pas reconnus tous les deux si vous utilisez le motif "a". Pour éviter ces désagrements, le package *stringr* vous offre plus de facilité. Il est présenté ensuite.

Elle permet de découper du texte avec un séparateur défini.

```
> liste <- c( "Croatia|Czech Republic|Denmark|Estonia" )
> strsplit( liste, "|", fixed=T )

## [[1]]
## [1] "Croatia"        "Czech Republic"
## [3] "Denmark"        "Estonia"

> strsplit( liste, "|", fixed=T )[[1]]

## [1] "Croatia"        "Czech Republic"
## [3] "Denmark"        "Estonia"
```

Attention elle renvoie une liste car chaque élement du vecteur est analysé et découpé et renvoyé dans un objet d'une liste.

```
> liste <- c(
+   "Croatia|Czech Republic|Denmark|Estonia",
+   "Finland|France|Germany|Greece"
+ )
> strsplit( liste, "|", fixed=T )

## [[1]]
## [1] "Croatia"        "Czech Republic"
## [3] "Denmark"        "Estonia"
##
## [[2]]
## [1] "Finland" "France"  "Germany" "Greece"
```

35.3 Fonctions avancées (*grep*)

grep est la fonction qui permet de renvoyer les numéros des élements qui correspondent à un pattern particulier. Vous l'utiliserez beaucoup par exemple pour sélectionner des variables.

```
> (sepal_var <- grep("Sepal",colnames(iris),fixed=T))

## [1] 1 2

> colnames(iris)[sepal_var]

## [1] "Sepal.Length" "Sepal.Width"

> (sepal_var <- grep("(?i)sepal",colnames(iris)))

## [1] 1 2

> colnames(iris)[sepal_var]

## [1] "Sepal.Length" "Sepal.Width"
```

L'option value est intéressante car elle renvoie les occurences qui correspondent au lieu de renvoyer un vecteur logique.

```
> grep("[0-9]+",c(
+   "12354", "fgdgd", "45648", "dfsdf456", "sdfsdf456456")
+   ,value=T
+ )

## [1] "12354"        "45648"
## [3] "dfsdf456"     "sdfsdf456456"
```

L'option invert est intéressante car elle renvoie les occurences qui ne correspondent pas sous la forme d'un vecteur logique ou dans l'exemple les valeurs elles-mêmes.

```
> grep("[0-9]+",c(
+   "12354", "fgdgd", "45648", "dfsdf456", "sdfsdf456456")
+   ,invert=T,value=T
+ )

## [1] "fgdgd"
```

35.4 Fonctions avancées (*sub*)

sub et *gsub* sont des opérateurs de remplacement de texte. Le premier se limite à une occurence dans chaque élement, le second généralise à n occurrences.

```
> gsub( "Sepal", "Sepale", colnames(iris) )

## [1] "Sepale.Length" "Sepale.Width"
## [3] "Petal.Length"  "Petal.Width"
## [5] "Species"
```

35.5 Fonctions avancées (motif approximatif)

La recherche de motifs floue (avec correspondance inexacte) n'est pas en reste.

Les fonctions agrep, charmatch & al. permettent de faire ce genre de recherches.

```
> agrep( "Sedal", colnames(iris), fixed=TRUE )

## [1] 1 2

> colnames(iris)[agrep( "Sedal", colnames(iris) )]

## [1] "Sepal.Length" "Sepal.Width"
```

De nombreuses options permettent d'obtenir le degré de similitude voulu.

35.6 Fonctions avancées

Les fonctions avancées basées exclusivement sur les expressions régulières sont : *regexpr*, *gregexpr*, *regexec*.

La fonction *glob2rx* peut aider les débutants en convertissant des motifs génériques en expressions régulières.

```
> glob2rx( "*Longueur" )

## [1] "^.*Longueur$"
```

36 stringr

36.1 Le paquet *stringr*

Ce paquet comme d'autres évoqués lors de la formation sont réalisé par Hadley Wickham. La syntaxe n'est pas sans rappeler celles de ses autres paquets : *ggplot2*, *reshape2*, *plyr*, ...

Ce paquet permet de simplifier l'utilisation des fonctions de base de R en ce qui concerne les chaines de caractères.

Ce paquet est construit lui-même sur un autre paquet *stringi*.

stringi utilise un bibliothèque (des fonctions) standard pour la manipulation des chaînes de caractère en Unicode.

Elle est disponible pour de nombreux langages : Java, Ruby, Perl, ...

36.2 Enlever les blancs

Ainsi pour enlever les blancs...

```
> str_trim( c( "  A", "b   ", "   C "), side="both")

## [1] "A" "b" "C"
```

L'argument *side* permet de spécifier à droite, gauche ou les deux.

36.3 Reconnaissance de motif

La fonction *grep* est simplifiée. La recherche précédente devient :

```
> (sepal_var <- str_detect( colnames(iris),
+                           fixed("sepal", ignore_case=T)))

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

Il existe d'autres fonctions qui copient *paste*, *nbchar*, ...

Et d'autres un peu plus originales...

Par exemple, *str_ count* permet de compter le nombre d'occurences, *str_ extract* permet d'extraire les motifs (pour les expressions régulières) et *str_ dup* pour trouver les doublons dans un vecteur.

37 Les fonctions

37.1 Les fonctions

Les fonctions sont un type d'objets R à part entière. Ainsi il existe comme pour les autres types d'objets une fonction *is* correspondante :

```
is.function( { function ( x ) {
 x^2
} } )

## [1] TRUE
```

Ce qui peut être perturbant pour les débutants est l'utilisation que vous avez pu voir de fonctions anonymes : les fonctions sont utilisées directement par exemple dans une fonction *apply*.

Mais les fonctions peuvent être également stockées pour être réutilisées plusieurs fois.

```
my.square <- function ( x ) {
 return(x^2)
}
my.square(3)

## [1] 9
```

Par défaut, si la dernière ligne renvoie une valeur, cette valeur est retournée par la fonction. Néanmoins pour rendre le code plus lisible et surtout plus robuste, il convient d'utiliser la fonction *return* qui prend **un seul** argument qui est renvoyé comme valeur de retour de la fonction.

Les fonctions en R ne renvoie qu'un seul objet. Par conséquent, il est souvent nécessaire de renvoyer des *lists* ou des *data.frames* pour récupérer l'ensemble du matériel créé au sein de la fonction.

Il existe une autre fonction similaire à *return* : *invisible*. Elle est utilisée abondamment dans R notamment par les commandes graphiques (ou *t.test* par exemple).

Elle permet de ne renvoyer une valeur que lorsque l'appel de la fonction est dans un contexte d'évaluation.

```
my.square <- function ( x ) {
 invisible(x^2)
}
my.square(3)
(my.square(3))

## [1] 9
```

37.2 Portée des variables dans une fonction

Dans R, les fonctions héritent de l'environnement père : c'est-à-dire que les objets disponibles dans l'environnement d'appel de la fonction le sont aussi au

sein de la fonction.

Mais les objets passés à la fonction sont des copies. Par conséquent, en R, toutes les modifications faîtes sur les objets au sein d'une fonction sont perdus. De plus si un objet est créé avec un nom existant dans l'environnement père, le nom de cet objet fait désormais référence à l'objet créé au sein de la fonction (et non à l'objet de même nom dans l'environnement père).

Pour les personnes disposant d'un bagage informatique solide, R utilise des passages par valeurs (et non par références) et utilise un procédé d'évaluation dit *lazy*...

Pour simplifier, tout objet n'est évalué que si l'évaluation est effectivement nécessaire dans le code. Il en va de même pour les objets copiés.

Ce phénomène est bien expliqué dans les manuels de R et dans les ouvrages avancés sur R.

On peut donc accéder à une valeur définie hors de la fonction.

```
z <- 2
 my.square <- function ( x ) {
 return(z*x^2)
}
my.square(3)

## [1] 18
```

A l'intérieur de la fonction, l'objet peut être modifié mais les changements resteront locaux et seront perdus à la fermeture de la fonction.

```
z <- 2
 my.square <- function ( x ) {
 z <- 4
 return(z*x^2)
}
my.square(3)

## [1] 36

z

## [1] 2
```

37.3 Environnement

Les variables créées dans la fonction sont détruites après la fin de l'exécution.

37.4 Les arguments d'une fonction

Les arguments peuvent être soit obligatoires soit optionnels.

Les arguments obligatoires ne prennent pas de valeur par défaut. C'est le cas pour le x de la fonction présentée précédemment dans ce document.

Les arguments sont avant tout positionnels. Mais pas seulement. Voyons la syntaxe de l'aide de la fonction *t.test*...

37.5 Les arguments

```
t.test(x, ...)

## Default S3 method:
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)
```

La première ligne indique que la fonction n'attends qu'un paramètre obligatoire x. On retrouve cette information dans la partie qui est réservée à l'appel par défaut de la fonction : il n'y a pas de valeurs par défaut pour x.

Par contre, tous les autres arguments ont des valeurs par défaut ce qui indique qu'ils sont optionnels.

On pourrait par exemple comparer la moyenne de deux vecteurs en appelant la fonction :

```
t.test(rnorm(1000),y=rnorm(1000,2))

##
##  Welch Two Sample t-test
##
## data:  rnorm(1000) and rnorm(1000, 2)
## t = -44.342, df = 1997.3, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.071372 -1.895908
## sample estimates:
##  mean of x  mean of y
## -0.0105896  1.9730503
```

Mais les arguments étant en premier lieu positionnels, cet appel suffit :

```
t.test(rnorm(1000),rnorm(1000,2))

##
##  Welch Two Sample t-test
##
## data:  rnorm(1000) and rnorm(1000, 2)
## t = -42.992, df = 1997.8, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.062978 -1.882977
## sample estimates:
##   mean of x   mean of y
## 0.004807897 1.977785590
```

Les arguments peuvent être passés de façon positionnels mais alourdirait le code. Aussi, on peux plus simplement préciser un couple *nom/valeur par défaut*.

```r
t.test(rnorm(1000),rnorm(1000,2),var.equal=TRUE)

##
##  Two Sample t-test
##
## data:  rnorm(1000) and rnorm(1000, 2)
## t = -44.111, df = 1998, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.089847 -1.911932
## sample estimates:
##   mean of x   mean of y
## 0.006032255 2.006921564
```

En temps normal lorsqu'un nom de paramètre incorrect est utilisé, R lève une exception.

Toutefois, lors de la création de la fonction, on peut utiliser un argument spécial : "...".

L'utilisation de cet argument indique à R que des arguments supplémentaires peuvent être passés à la fonction.

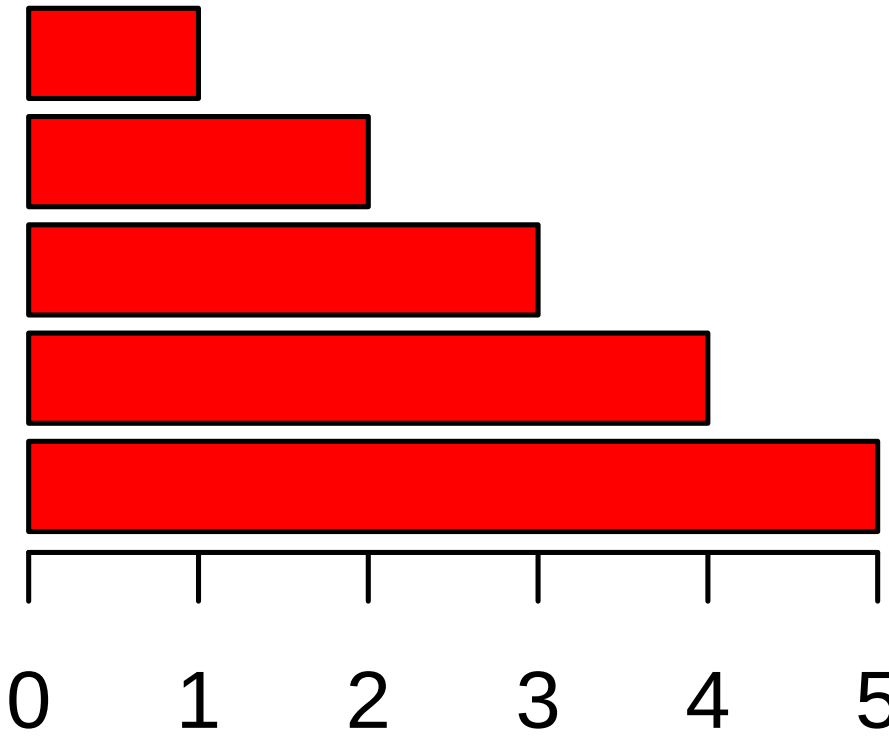
R ne lèvera pas d'exception si la correspondance entre le nom des arguments d'appel et le nom des arguments définis n'est pas correct.

Par contre il conserve les arguments supplémentaires et peut les passer à une autre fonction appelée au sein de la première fonction.

C'est extrèmement pratique pour *surcharger* une fonction existante. Le plus souvent pour des fonctions graphiques qui ont de nombreux paramètres.

Par exemple, pour créer des *barplot* différents des barplots par défaut...

```r
my.barplot <- function( x, horiz=T, ... ) {
  barplot( x, horiz=horiz, ... )
}
#my.barplot( c(5,4,3,2,1), col="red" )
```


0
1
2
3
4
5

37.6 Changement dans l'environnement père...

En fait il existe une possibilité pour changer la valeur d'une variable dans l'environnement père.

C'est pratique pour modifier une *data.frame* encombrante par exemple.

```
i <- 1
a <- function (x) { i <- 2 }
i

## [1] 1

i <- 1
a <- function (x) { i <<- 2 }
a(7);i;

## [1] 2
```

L'inconvénient est que cela rend la fonction dépendante de l'environnement père et du nom des variables dans celui-ci.

Son utilisation est donc à limiter sauf cas particuliers.

38 Les structures de contrôle

38.1 Les boucles

Les boucles sont à éviter car lentes à exécuter. Il faut leur préférer les fonctions de type *apply*. La syntaxe d'une boucle est la suivante...

```
for ( mavar in sequence ) {
       ... code R...
}
```

la variable *mavar* prend à chaque itération un élement de *sequence* dans l'ordre. Les itérations peuvent se faire sur un type quelconque comme des entiers (usuels) mais également un vecteur de *character* par exemple. Ou bien un vecteur de fonctions...

38.2 Les tests

Les tests ont la structure suivante :

```
if ( valeur ) {
       ... code R...
}
```

ou

```
if ( valeur ) {
 ... code R...
} else {
... code R...
}
```

La condition est executé si la valeur est *TRUE*, *T* ou différent de 0.

Attention, le vecteur booléen doit être de longueur 1. A l'intérieur d'un test, R attend *T* ou *F* et pas *c(T,F,T)*.

Les fonctions à connaître sont donc *any* qui renvoie vrai si au moins un élement est vrai dans le vecteur passé en argument.

Et la fonction *all* qui renvoie vrai si toutes les valeurs du vecteur passé en argument sont vrai.

Des opérations sur les booléens disponibles :

- qui *renvoient* des vecteurs de longueur plus grande que 1
  `&` : et
  `|` : ou
- qui *renvoient* des vecteurs de longueur 1
  `&&` : et
  `||` : ou

Il y a une fonction à connaître car très rapide et très simple :

```
ifelse( mavar, valeur_si_vrai, valeur_si_faux )
```

Par exemple :

```
ifelse( rnorm(10) > 0, 1, -1 )

##  [1]  1 -1  1  1 -1 -1 -1  1 -1 -1
```

38.3 Stopper l'exécution

La fonction *stop* permet d'arrêter un script et d'indiquer une erreur.

```
if ( class != "numeric" ) stop("Non numerique")
```

39 Les fonctions apply

39.1 Les différentes fonctions

Dans la famille *apply*, on a en fait :

```
lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
replicate(n, expr, simplify = TRUE)
```

Par exemple, nous voulons par exemple récupérer les quantiles de toutes les variables numériques. Pour cela, nous utilisons la fonction *apply*.

```
(r <- apply(iris[,1:4],2,quantile))

##      Sepal.Length Sepal.Width Petal.Length
## 0%            4.3         2.0         1.00
## 25%           5.1         2.8         1.60
## 50%           5.8         3.0         4.35
## 75%           6.4         3.3         5.10
## 100%          7.9         4.4         6.90
##      Petal.Width
## 0%           0.1
## 25%          0.3
## 50%          1.3
## 75%          1.8
## 100%         2.5
```

La fonction *apply* permet d'appliquer une fonction sur une *data.frame* dans le sens :

- des lignes, ligne par ligne, avec l'indice 1
- des colonnes, colonne par colonne, avec l'indice 2
- cellule par cellule avec l'indice *1 :2* (ou *c(1,2)*)

39.2 Les fonctions apply

Donc pour l'exemple précédent, calculer les quantiles, on demande à R de passer chaque colonne à la fonction quantile.

La fonction quantile rend un vecteur et R se "débrouille" tout seul avec les vecteurs résultats : il les aggrège sous forme de matrice.

39.3 Les différentes fonctions

Par exemple sapply, prends comme argument une *list* et renvoie quelque chose de simplifié quand elle le peut.

Par exemple pour retrouver les colonnes numeric d'une *data.frame*...

```
sapply( iris, is.numeric )

## Sepal.Length  Sepal.Width Petal.Length
##         TRUE         TRUE         TRUE
##  Petal.Width      Species
##         TRUE        FALSE
```

Pourquoi ça marche ?

Parce que *data.frame* peut être convertie en *list* puis la fonction est appliquée à chaque élement de la *list*.

```
str(as.list(iris))
```

```
## List of 5
##  $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

L'avantage de *sapply* est qu'elle renvoie un objet simplifié par rapport à *lapply*.

vapply est identique avec un contrôle sur le type d'objet renvoyé.

replicate est une fonction extrêmement utile. Un des gros avantages de R est qu'il permet très aisement de simuler des données.

replicate est une des fonctions qui permet de le faire en répétant une boucle tout en générant des nombres aléatoires.

39.4 les autres fonctions apply

mapply se distingue car elle peut prendre plusieurs arguments.

vapply est utilisé sur les vecteurs et permet la vérification du type en sortie.

...

39.5 Les différentes fonctions

```
set.seed(42)
system.time(
res1 <- replicate( 10000, function() { return(mean(rnorm(1000))) } )
)

##    user  system elapsed
##    0.02    0.00    0.07

system.time({
res2 <- numeric(10000)
for ( ii in 1:10000 ) { res2[ii] <- mean(rnorm(1000)) }
})

##    user  system elapsed
##    1.80    0.00    3.29
```

Ce qu'il ne faut surtout pas faire :

```
system.time({
res2 <- c()
for ( ii in 1:10000 ) { res2 <- c( res2, mean(rnorm(1000)) ) }
})

##    user  system elapsed
##    2.28    0.00    2.31
```

39.6 Un exemple, le bootstrap...

```
n <- 1000
set.seed(42)
b <- replicate( n, mean( sample( patient$totalechelle,
                                 length(patient$totalechelle),
                                 replace = T ), na.rm=T ) )
mean((b-mean(b))^2)

## [1] 2055.7
```

39.7 Les boucles

```
for ( ii in 1:4 ) { print(ii) }

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

```
for ( ww in LETTERS[1:4] ) { print(ww) }

## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
```

En vrai, une boucle pourrait servir à ça :

```
a <- numeric(4)
for ( ii in 1:4 ) { a[ii] <- mean(rnorm(1000)) }
a

## [1]  0.02653023 -0.01359218 -0.06310900
## [4] -0.02976024
```

Ce qui s'écrit plus simplement et surtout beaucoup plus efficacement :

```
a <- vapply(1:4,function(x) mean(rnorm(x)),numeric(1))
a

## [1]  0.78987050 -0.06258372 -0.03924717
## [4]  0.73992320
```

En vrai, une boucle pourrait servir à ça :

```
vars <- colnames(iris)[sapply(iris,is.numeric)]
for ( ii in vars ) { iris[ii] <- scale(iris[ii]) }
```

Ce qui s'écrit plus simplement et surtout beaucoup plus efficacement :

```
vars <- colnames(iris)[sapply(iris,is.numeric)]
iris[,vars] <- apply(iris[,vars],2,scale)
```

Une utilisation justifiée des boucles.

```
for ( ww in c( function(x) {x^1}, function(x) {x^2}, function(x) {x^3} ) ) { print(ww(2)) }

## [1] 2
## [1] 4
## [1] 8
```

En fait, non

```
power <- function(n,x) {x^n}
sapply(as.list(1:3),power,x=2)

## [1] 2 4 8
```

39.8 Split...

La fonction *split* permet de découper une *data.frame* en fonction des modalités d'une variable et de récupérer une *list* en sortie avec pour chaque modalité la partie correspondante de la *data.frame*.

```
str(split(iris,factor(iris$Species)))

## List of 3
##  $ setosa    :'data.frame': 50 obs. of  5 variables:
##   ..$ Sepal.Length: num [1:50] -0.898 -1.139 -1.381 -1.501 -1.018 ...
##   ..$ Sepal.Width : num [1:50] 1.0156 -0.1315 0.3273 0.0979 1.245 ...
##   ..$ Petal.Length: num [1:50] -1.34 -1.34 -1.39 -1.28 -1.34 ...
##   ..$ Petal.Width : num [1:50] -1.31 -1.31 -1.31 -1.31 -1.31 ...
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ versicolor:'data.frame': 50 obs. of  5 variables:
##   ..$ Sepal.Length: num [1:50] 1.397 0.672 1.276 -0.415 0.793 ...
##   ..$ Sepal.Width : num [1:50] 0.3273 0.3273 0.0979 -1.7375 -0.5904 ...
##   ..$ Petal.Length: num [1:50] 0.534 0.42 0.647 0.137 0.477 ...
##   ..$ Petal.Width : num [1:50] 0.263 0.394 0.394 0.132 0.394 ...
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2 2 2 2 2 2 2 2 2 2 ...
##  $ virginica :'data.frame': 50 obs. of  5 variables:
##   ..$ Sepal.Length: num [1:50] 0.5515 -0.0523 1.5176 0.5515 0.793 ...
##   ..$ Sepal.Width : num [1:50] 0.557 -0.82 -0.132 -0.361 -0.132 ...
##   ..$ Petal.Length: num [1:50] 1.27 0.76 1.21 1.04 1.16 ...
##   ..$ Petal.Width : num [1:50] 1.706 0.919 1.182 0.788 1.313 ...
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 3 3 3 3 3 3 3 3 3 3 ...
```

39.9 do.call

do.call est une fonction assez complexe. Elle permet notamment de définir l'environnement dans lequel exécuté une commande R.

Toutefois elle a une utilisation simple à connaître. Elle permet en une ligne d'aggréger des résultats provenant d'une commande lapply.

```
stats <- function (x) { c(
  quantile( x$Sepal.Length,probs=c(0,0.25,0.5,0.75,1)),
  mean(x$Sepal.Length),
  sd(x$Sepal.Length) )
}
res <- lapply( split(iris,iris$Species), stats )
str(res)

## List of 3
##  $ setosa    : Named num [1:7] -1.8638 -1.26 -1.0184 -0.7769 -0.0523 ...
##   ..- attr(*, "names")= chr [1:7] "0%" "25%" "50%" "75%" ...
##  $ versicolor: Named num [1:7] -1.1392 -0.2939 0.0684 0.5515 1.3968 ...
##   ..- attr(*, "names")= chr [1:7] "0%" "25%" "50%" "75%" ...
##  $ virginica : Named num [1:7] -1.139 0.461 0.793 1.276 2.484 ...
##   ..- attr(*, "names")= chr [1:7] "0%" "25%" "50%" "75%" ...
```

```
do.call( rbind, res )

##                  0%        25%         50%
## setosa     -1.86378 -1.2599638 -1.01843718
## versicolor -1.13920 -0.2938574  0.06843254
## virginica  -1.13920  0.4609133  0.79301235
##                   75%        100%
## setosa     -0.7769106 -0.05233076 -1.0111914
## versicolor  0.5514857  1.39682886  0.1119073
## virginica   1.2760656  2.48369858  0.8992841
##
## setosa     0.4256782
## versicolor 0.6233453
## virginica  0.7679092
```

Si l'exemple peut être réalisé par exemple avec plyr, il est bonne illustration de *do.call*.

Plutôt qu'une matrice, si les résultats sont de types différents, on peut écrire dans certains cas :

```
do.call( data.frame, res )
```

39.10 lapply

La fonction *lapply* est une fonction dont l'utilisation doit croître avec l'expérience. Elle est centrale dans R et s'annonce de plus en plus indispensable car elle est à la base des fonctions de vectorisation des calculs dans R.

Par exemple, un jackknife, est très facile à réaliser avec une fonction *lapply*.

```
mm <- mean(iris[,"Sepal.Length"])
res <- sapply( as.list(1:nrow(iris)),
               function (x) {
                 (mean(iris[-x,"Sepal.Length"])-mm)^2
} )
vv <- sqrt(sum(as.numeric(res))/(nrow(iris)*(nrow(iris)-1)))
```

```
paste( "[", qt(0.025,nrow(iris)-1)*vv+mm,
       ":", qt(0.975,nrow(iris)-1)*vv+mm, "]" )

## [1] "[ -0.00108282416339017 : 0.00108282416339017 ]"
```

39.11 Calculs parallèles

La vectorisation est pour l'instant assez peu documenté. Il existe l'ouvrage de McCallum (2012) et quelques ressources dans les blogs sur R.

Sous les systèmes de type GNU/Linux, la vectorisation sur une même machine est d'une simplicité évangélique. Il suffit de charger le paquet *parallel* et de spécifier le nombre de processeurs à utiliser et d'utiliser la fonction *mclapply*.

Ce qui donne pratiquement le même code que précedemment pour un jackknife...

```
mm <- mean(iris[,"Sepal.Length"])
res <- mclapply( as.list(1:nrow(iris)), function (x)
  (mean(iris[-x,"Sepal.Length"])-mm)^2,
  mc.cores=4
)

vv <- sqrt(sum(as.numeric(res))/(nrow(iris)*(nrow(iris)-1)))

paste( "[", qt(0.025,nrow(iris))*vv+mm, ":",
       qt(0.975,nrow(iris))*vv+mm, "]" )
```

Avec ce mécanisme, 10 processus R vont être lancés en parallèle sur la machine. La mémoire nécessaire à chaque processus doit être disponible. Ce qui revient à demander à la machine 4 fois la mémoire nécessaire à l'éxecution du processus.

Le système utilise la commande *fork* du système d'exploitation. Par conséquent, chaque processus récupère l'environnement (variables) et paquets de la session courante. Pratique.

Dans le cas de simulation, il est nécessaire de bien lire l'aide du package pour obtenir selon ses besoins des seeds parallèles ou asynchrone.

Dans le cas de machine Windows, cette méthode ne fonctionne pas en raison du fonctionnement de Windows (quelque soit sa version).

Aussi dans ce cas et pour faire du calcul parallèle en gérant plus finement les ressources matériels et plusieurs ordinateurs quelque soit leur système d'exploitation, il est nécessaire de passer plutôt par l'utilisation des framework SNOW et MPI par exemple.

L'utilisation est plus délicate car l'utilisateur doit notamment indiquer quelles variables, quels paquets, ... doivent être injectés dans les processus avant le lancement du calcul.

Une vue entière est dédiée au problème des calculs lourds...

High-Performance and Parallel Computing with R

40 L'automatisation des scripts

40.1 Lancement d'un script automatiquement

Pour lancer un script automatiquement, on peut le faire dans un fichier *batch*, c'est-à-dire un petit executable qui se termine en *.bat* sous *Windows*.

Il est conseillé de mettre le chemin de R dans le PATH *Windows* pour ne pas avoir à taper le chemin complet d'accès à R.

On peut ainsi appeler un script :

```
R -f Monscript.R
```

Mais R a une commande spécialement conçues pour réaliser des opérations depuis des fichiers exécutables...

```
R CMD BATCH Monscript.R
```

Un fichier *.Rout* est généré automatiquement et contient tout ce qui est apparu dans la console.

40.2 *source*

La fonction *source* permet d'exécuter le contenu d'un script depuis un autre script.

Cela permet par exemple de stocker des fonctions génériques puis de les rappeler en suite sans faire de paquets...

```
source("MesFonctions.R")
monbarplot(iris$Species)
```

40.3 Les règles de rédaction des scripts

R est un langage de programmation...

Pour la relecture et la lisibilité du code penser à commenter et à indenter !

Il est souvent plus simple d'utiliser un éditeur de texte tel que *emacs* ou *notepad++* pour profiter de la coloration syntaxique puis de copier-coller dans la console R.

ou *RStudio*.

41 Les statistiques descriptives complexes

41.1 Les fonctions de base pondérées

R a la mauvaise réputation de ne pas tenir compte des poids de sondage.

Si cela est (en partie) vrai pour les fonctions de base, de nombreux paquets permettent désormais de tenir compte des poids de sondage :

questionr il regroupe des fonctions originales provenant d'autres paquets. Il permet notamment de faire les statistiques de base sur des données pondérées

FactoMineR un paquet d'ADD très avancé qui utilise les poids de sondage

nmle pour les modèles mixtes

survey analyse factorielle et modèles linéaires généralisés

lavaan.survey pour les modèles structuraux

...

Différentes solutions existent aussi pour traiter des données d'enquêtes internationales (PISA, ESLC, ...) qui prennent en compte notamment les *plausible values*.

Deux paquets notamment sont entièrement dédiés aux sondages :

survey permet des calculs complexes de précision (penser à surveymeans sous amphétamine). Il permet notamment de manipuler les données de sondages internationaux tels que PIRLS, PISA, IALS, ... qui utilise des *replication weights*. Un package complémentaire permet de faire des calculs sur des scores de performance (*svyPVpack*).

sampling un paquet pour réaliser essentiellement des tirages d'échantillon et de calage (aka CUBE et CALMAR). Il est écrit par l'équipe de Y. Tillé.

Dans *questionr*, il suffit d'utiliser les fonctions :

- wtd.means
- wtd.var
- wtd.table

```
> ddply( xtfme, .(strate), summarize,
+   moy_f=wtd.mean(vali_f,poids), sd_f=sqrt(wtd.var(vali_f,poids)),
+   moy_m=wtd.mean(vali_m,poids), sd_m=sqrt(wtd.var(vali_m,poids))
+ )
```

```
##   strate     moy_f      sd_f     moy_m
## 1      1 0.9003984 0.2994967 0.9213147
## 2      2 0.7947574 0.4042547 0.8315672
## 3      3 0.7224355 0.4490306 0.7761931
## 4      4 0.9134360 0.2813445 0.9373340
##        sd_m
## 1 0.2692727
## 2 0.3745985
## 3 0.4179420
## 4 0.2424900
```

Pour les fonctions de base, c'est un peu compliqué, car on doit garder la variable de poids en plus de la variable sur les calculs sont réalisées dans les regroupements.

Cela nécessite une certaine pratique. En réalité, il faut se tourner rapidement vers le paquet *survey* pour une estimation précise de la variance et l'analyse plus complexe.

41.2 Les croisements de variables, ...

Ces cas sont bien décrits dans les diapos concernant la manipulation de données et les fonctions *apply & al.*.

42 Formules

42.1 Les formules

Les *formula* sont un type spécial d'objet sous R.

Elles se reconnaissent par l'utilisation du ~.

```
y ~ a*x+b
```

est l'équivalent de $y = ax + b$.

42.2 le test de Student

Un exemple simple d'utilisation d'une formule est le test de Student.

Dans ce cas on veut savoir si la variable totalechelle est fonction du sexe (variable qualitative).

On peut l'écrire sous forme d'une formule :

```
> t <- t.test( totalechelle ~ sexe, data = p )
```

```
> t

##
##  Welch Two Sample t-test
##
## data:  totalechelle by sexe
## t = 4.8147, df = 134.66, p-value =
## 3.906e-06
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  228.9551 548.1834
## sample estimates:
##  mean in group Feminin
##               595.7426
## mean in group Masculin
##               207.1733
```

43 Régression linéaire

43.1 La régression linéaire

Pour la regression linéaire, la fonction porte le nom de *lm*.
Elle utilise les formules.

```
> (rl <- lm( totalechelle ~ dureeopmin, data = p ))

##
## Call:
## lm(formula = totalechelle ~ dureeopmin, data = p)
##
## Coefficients:
## (Intercept)   dureeopmin
##       96.29         1.67
```

```
> summary(rl)

##
## Call:
## lm(formula = totalechelle ~ dureeopmin, data = p)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -894.8 -282.8 -129.6   76.9 3521.5
##
## Coefficients:
##             Estimate Std. Error t value
## (Intercept)  96.2924    76.2405   1.263
## dureeopmin    1.6705     0.3131   5.335
##             Pr(>|t|)
## (Intercept)    0.208
## dureeopmin  2.94e-07 ***
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
##
## Residual standard error: 577.8 on 174 degrees of freedom
## Multiple R-squared:  0.1406,Adjusted R-squared:  0.1357
## F-statistic: 28.47 on 1 and 174 DF,  p-value: 2.937e-07
```

Les fonctions disponibles pour les objets lm sont plus nombreuses...

```
> methods(class=class(rl))

##  [1] add1           addterm
##  [3] alias          anova
##  [5] boxcox         case.names
##  [7] coerce         confint
##  [9] cooks.distance deviance
## [11] dfbeta         dfbetas
## [13] drop1          dropterm
## [15] dummy.coef     effects
## [17] extractAIC     family
```

```
## [19] formula        hatvalues
## [21] influence      initialize
## [23] kappa          labels
## [25] logLik         logtrans
## [27] model.frame    model.matrix
## [29] nobs           plot
## [31] predict        print
## [33] proj           qr
## [35] residuals      rstandard
## [37] rstudent       show
## [39] simulate       slotsFromS3
## [41] summary        variable.names
## [43] vcov           xtable
## see '?methods' for accessing help and source code
```

Notamment on peut se passer d'accéder à l'objet directement grâce à des 'accesseurs' qui permettent de renvoyer les données les plus intéressantes comme par exemple les résidus (*residuals*), les valeurs ajustées (*fitted*), ...

43.2 La régression linéaire multiple

Dans le cas de la régression linéaire multiple, il suffit d'ajouter des variables explicatives.

Dans ce cas il y a deux possibilités : soit le $+$, soit le $*$.

La différence est que le $+$ n'indique pas d'interaction tandis que le $*$ indique une interaction.

```
> (rlm <- lm( totalechelle ~ dureeopmin + age + nbttt + nbechelle, data = p ))

##
## Call:
## lm(formula = totalechelle ~ dureeopmin + age + nbttt + nbechelle,
##     data = p)
##
## Coefficients:
## (Intercept)   dureeopmin          age
##   -271.3861      -0.8769       9.3682
##       nbttt    nbechelle
##     59.2139      31.0276
```

```
> summary(rlm)

##
## Call:
## lm(formula = totalechelle ~ dureeopmin + age + nbttt + nbechelle,
##     data = p)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1181.79  -143.18    -9.61   141.26  1183.06
##
## Coefficients:
##              Estimate Std. Error t value
```

```
## (Intercept) -271.3861    58.4513  -4.643
## dureeopmin    -0.8769     0.2179  -4.025
## age            9.3682     4.6729   2.005
## nbttt         59.2139    10.8305   5.467
## nbechelle     31.0276     1.7423  17.808
##              Pr(>|t|)
## (Intercept) 6.82e-06 ***
## dureeopmin  8.55e-05 ***
## age           0.0466 *
## nbttt       1.60e-07 ***
## nbechelle    < 2e-16 ***
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
##
## Residual standard error: 297.1 on 171 degrees of freedom
## Multiple R-squared:  0.7767,Adjusted R-squared:  0.7715
## F-statistic: 148.7 on 4 and 171 DF,  p-value: < 2.2e-16
```

43.3 L'ANOVA

Cette fois-ci on réalise une ANOVA en tenant d'une interaction entre les deux variables :

```
> (paov <- aov( totalechelle ~ sexe * CIM2, data = p ))

## Call:
##    aov(formula = totalechelle ~ sexe * CIM2, data = p)
##
## Terms:
##                       sexe     CIM2 sexe:CIM2
## Sum of Squares   6498405  7490556   1953146
## Deg. of Freedom        1        1         1
##                 Residuals
## Sum of Squares   51657018
## Deg. of Freedom       172
##
## Residual standard error: 548.0251
## Estimated effects may be unbalanced
```

```
> summary(paov)

##              Df   Sum Sq Mean Sq F value
## sexe          1  6498405 6498405  21.637
## CIM2          1  7490556 7490556  24.941
## sexe:CIM2     1  1953146 1953146   6.503
## Residuals   172 51657018  300332
##               Pr(>F)
## sexe        6.54e-06 ***
## CIM2        1.44e-06 ***
## sexe:CIM2     0.0116 *
## Residuals
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
```

43.4 Les formules avancées en régression linéaire

Il est possible de modifier la formule pour faire des modèles plus complexes. Il n'est pas nécessaire pour cela de transformer les variables au préalable. Il faut pour cela utiliser l'argument *I*.

```
> (rlm1 <- lm( totalechelle ~ I(log(dureeopmin)) + age + nbttt +
+               I(log(nbechelle)), data = p ))

##
## Call:
## lm(formula = totalechelle ~ I(log(dureeopmin)) + age + nbttt +
##     I(log(nbechelle)), data = p)
##
## Coefficients:
##        (Intercept)  I(log(dureeopmin))
##            -293.86             -105.59
##                age               nbttt
##              15.73               82.84
##  I(log(nbechelle))
##             300.44
```

Autre exemple...

```
> (rlm2 <- lm( totalechelle ~ I(log(dureeopmin)) + age + nbttt +
+               I(nbttt^2) + I(log(nbechelle)), data = p ))

##
## Call:
## lm(formula = totalechelle ~ I(log(dureeopmin)) + age + nbttt +
##     I(nbttt^2) + I(log(nbechelle)), data = p)
##
## Coefficients:
##        (Intercept)  I(log(dureeopmin))
##           -258.433             -95.273
##                age               nbttt
##             15.881              44.397
##         I(nbttt^2)   I(log(nbechelle))
##              2.613             306.786
```

43.5 Comparer des modèles avec Stargazer

```
> stargazer(rlm1,rlm2,
+            column.labels = c("Modèle","Modèle"),
+            align=T)
```

	Dependent variable :	
	totalechelle	
	Modèle	Modèle
	(1)	(2)
I(log(dureeopmin))	-105.590^{*}	-95.273^{*}
	(54.147)	(55.257)
age	15.730^{**}	15.881^{**}
	(6.531)	(6.535)
nbttt	82.843^{***}	44.397
	(14.943)	(43.402)
I(nbttt^2)		2.613
		(2.769)
I(log(nbechelle))	300.445^{***}	306.786^{***}
	(35.785)	(36.421)
Constant	-293.862	-258.433
	(232.917)	(235.998)
Observations	176	176
R^2	0.549	0.552
Adjusted R^2	0.539	0.539
Residual Std. Error	422.039 (df = 171)	422.174 (df = 170)
F Statistic	52.130^{***} (df = 4 ; 171)	41.856^{***} (df = 5 ; 170)
Note :		*p<0.1 ; **p<0.05 ; ***p<0.01

43.6 Modèles linéaires généralisés

Cette fois, il s'agit de la glm. Son fonctionnement est similaire.

Pour une régression logistique, la syntaxe est relativement simple.

On va examiner pour la table patient si on peut identifier l'hopital des patients :

```
> patient$hopital <- ifelse( patient$Hopital == "A", 0, 1 )
```

```
> glm.model <- glm(hopital ~ vitaux + dureeopmin + nbttt +
+                    nbechelle,
+                  data = patient,
+                  family=binomial())
```

	Dependent variable :
	hopital
vitaux	-0.033^{**}
	(0.016)
dureeopmin	0.003^{*}
	(0.002)
nbttt	0.379^{***}
	(0.103)
nbechelle	-0.050^{***}
	(0.015)
Constant	-1.070^{***}
	(0.364)
Observations	176
Log Likelihood	-106.413
Akaike Inf. Crit.	222.826
Note :	*p<0.1 ; **p<0.05 ; ***p<0.01

44 Le principe de Sweave

44.1 Le principe

Le but de *Sweave* est de générer des rapports automatiques dont le corps du texte est en LaTeX et dont une partie du contenu peut varier selon le code R et les données.

Le code R se trouve entre des balises spécifiques et est interprété. Les résultats sont insérés dans un fichier LaTeX.

Ainsi, il permet d'inclure des résultats et des figures générés par le code R dans le rapport final.

En outre, il permet d'inclure ou non le code qui a généré les résultats et/ou la(es) figure(s).

44.2 Le format Rnw

Le format de fichier *Sweave* est traditionnellement appelé *.Rnw*. C'est une référence à *S+* où le format de fichier était *.Snw*.

Ce format de fichier mèle à la fois du code R et LaTeX : les éditeurs de texte ont donc des difficultés à souligner la syntaxe des deux langages.

Deux exceptions, *Emacs* et *RStudio* permettent de souligner la syntaxe à la fois du code LaTeX et du code R.

En pratique, le fichier *Rnw* va être compilé par R et R va inclure les résultats et les liens vers les figures dans un fichier *.tex*.

Puis il suffira alors de compiler le fichier LaTeX résultant pour obtenir le document final.

44.3 Hello World

```
\documentclass{article}
<HelloWorld>>=
print("Hello World")
@
```

Le résultat :

```
print("Hello World")

## [1] "Hello World"
```

45 Les nouveautés

45.1 knitr

Sweave est devenue obsolète suite à un nouveau paquet développé par Yihui Xie.

Ce nouveau paquet s'appele *knitr*. En plus de moderniser, d'ajouter des fonctionnalités à *Sweave*, il permet également de générer des documents HTML, Markdown (une sorte de HTML simplifiée), des documents RTF, ...

Sur la page de garde, Xie explique les raisons de knitr.

En plus *knitr* est totalement intégré à RStudio et facilitant grandement les choses. Il faut dire que Xie a rejoint l'équipe de dev de RStudio...

knitr est similaire à *Sweave* dans les principes généraux pour LaTeXd'où la persistance de la présentation plus haut.

Mais le document se referera à *knitr* plutôt qu'à Sweave.

Pour passer à knitr, il faut faire un petit changement dans la configuration de RStudio : dans le menu *Tools*, *Global Options*, *Sweave* puis choisir knitr dans *Weave Rnw....*

46 Mon premier fichier knitr

46.1 Hello World

Les accolades doubles permettent de définir le début d'un *S Chunk*. Il est terminé par un @.

Le code R à l'intérieur va être interprété et par défaut affiché. Puis dans un second temps, ce qui apparaitrait dans la console va apparaître dans un format compatible avec LaTeX.

Pour générer le fichier LaTeX, on peut le faire dans la console R avec la commande *knit('HelloWorld.Rnw')* ou le bouton *Compile PDF* en haut de la fenêtre de l'éditeur.

Une autre commande utile est *purl*. Elle permet de créer un fichier *.R* qui contient tout le code R contenu dans le fichier *Rnw*.

Le code résultant dans le fichier *.tex* est le suivant :

```
\begin{knitrout}\footnotesize
\definecolor{shadecolor}{rgb}{0.969, 0.969, 0.969}\color{fgcolor}\begin{kframe}
\begin{alltt}
\hlkwd{print}\hlstd{(}\hlstr{"Hello World"}\hlstd{)}
\end{alltt}
begin{verbatim}
## [1] "Hello World"
end{verbatim}
\end{kframe}
\end{knitrout}
```

On voit apparaître les différents environnements qui sont définis dans le fichier de style *knitr* qui vient avec R.

knitrout,kframe qui est l'environnement général

alltt qui est l'environnment pour les commandes R

verbatim qui est l'environnment pour les résultats affichés dans la console.

Pour l'exemple, voilà le code pour obtenir un *summary*...

```
<<>>=
summary(rnorm(1000))
@
```

```
summary(rnorm(1000))

##     Min.  1st Qu.   Median     Mean  3rd Qu.
## -2.87900 -0.61930  0.04712  0.04112  0.71540
##     Max.
##  3.87300
```

46.2 Les balises LaTeX

Les environnements LaTeX définis par le fichier de style sont des environnements qui proviennent de l'environnement *verbatim*.

Ils sont personnalisables, en partie, en utilisant les packages *fancyvrb* notamment.

46.3 La portée des variables

Un environnement est créé lors du lancement du premier *chunk*. Puis cet environnement est conservé dans chaque *chunk* ultérieur. On peut donc utiliser une variable définie dans un *chunk* précédent.

46.4 Sexpr...

Moins qu'un bloc entier on peut extraire une seule valeur, par exemple une p-value d'un t.test...

```
<TTEST,echo=F>>=
ts <- t.test( rnorm(1000), rnorm(1000,1) )
@
```

Ensuite pour l'insérer dans le texte, on utilise le mot clef *Sexpr*.

Le code est le suivant : La p-value obtenue est de \Sexpr {ts $ p.value }.

Ce qui donne...

La p-value obtenue est de $1.9377497 \times 10^{-91}$.

Plus élégant...

La p-value obtenue est de \Sexpr {round(ts$p.value,3) }.

Ce qui donne...

La p-value obtenue est de 0.

47 Les options de knitr

47.1 Les options du mode Sweave

label ou premier argument on peut donner un nom au bloc Sweave. Dans ce cas, lors de l'interprétation, cela permet de situer plus facilement d'où vient l'erreur

echo si echo est *FALSE* alors les commandes R qui générent les résulats ne sont pas affichés *pdf* pour la compilation du document LaTeX

fig.width,fig.height définit la largeur et la hauteur de l'image produite

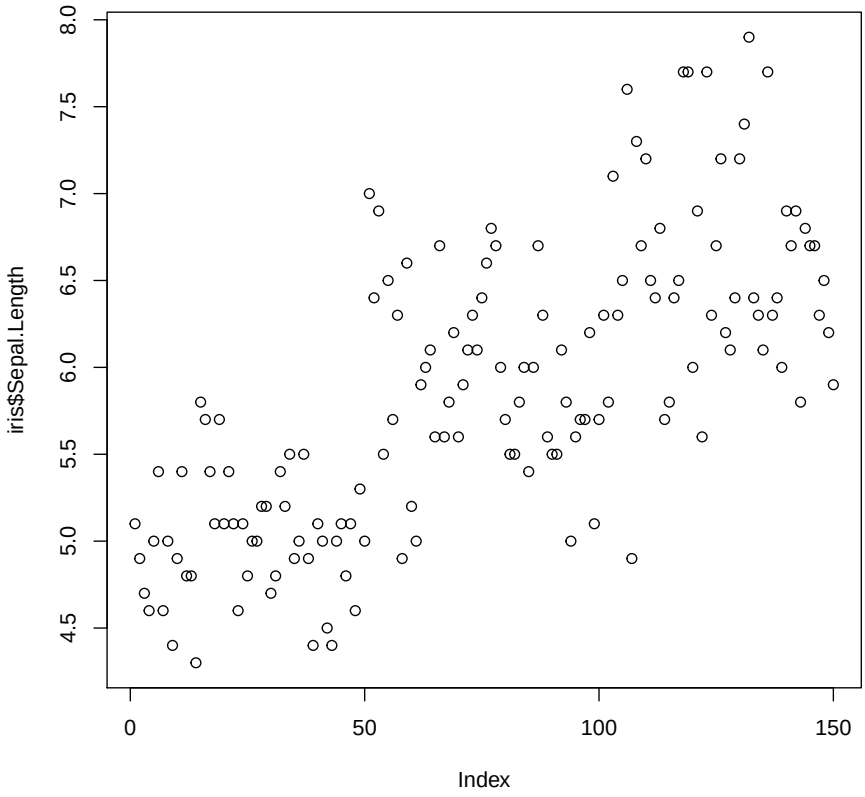
out.width,out.height définit la largeur et la hauteur de l'image dans le document

```
<echo=F,label=mychunk>>=
summary(rnorm(1000))
@
```

donne

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.
## -3.25200 -0.61880  0.02246  0.00763  0.66560
##     Max.
##  2.88300
```

```
<echo=F,width=4,height=4>>=
a <- rnorm(1000)
plot( a, a*5 + 10 + rnorm(1000) )
@
```



47.2 Les options du mode knitr

Il est possible de modifier les options des *chunk* pour l'ensemble du document.

Il suffit pour cela dans le fichier *.Rnw* de spécifier dans le préambule :

```
<options,eval=F>>=
opts_chunk$set(
fig.path='graphiques/beamer-',fig.align='center',
fig.show='hold',size='footnotesize',fig.height=3.3,
fig.width=3.3,out.width="\\textwidth",cache=T,echo=T)
@
```

47.3 Les options "results" des chunks

Il existe une autre option très importante, c'est l'option *results*.

Par défaut, elle vaut *markup*. Dans ce cas ce qui est affiché dans le fichier LaTeX final est de type *verbatim* amélioré.

Mais on peut lui donner comme valeur *hide*. Dans ce cas le *chunk* est exécuté mais aucune sortie n'est visible.

L'option *asis*... Dans ce cas ce qui sort du *chunk* est brut (donc du code LaTeX).

Et enfin "hold" qui ne fait pas de sortie au fur et à mesure mais accumule et affiche tout à la fin.

```
<Tex,results=tex>>=
cat("\\LaTeX est pratique")
@
```

donne

```
cat("\\LaTeX est pratique")
```

LaTeXest pratique

47.4 Les options erreurs/warnings des chunks

warning TRUE. Indique si on affiche les warnings si TRUE

message TRUE. Affiche les messages si TRUE

error TRUE. S'arrête en cas d'erreurs si FALSE.

47.5 Autres options des chunks

cache FALSE. Indique si on souhaite stocker les calculs... Ainsi des calculs lourds peuvent n'être refait que si on modifie le chunk ou ses arguments

eval FALSE. Ne traite pas le code R dans le chunk

fig.path Stocke les graphiques non dans le répertoire courant mais dans le répertoire de son choix.

47.6 Calcul dans les paramètres des chunks

Par exemple vous souhaitez faire un graphique que dans le cas où la date du jour est postérieur au 1er janvier... Les valeurs passées au chunk peuvent être issu d'un calcul dans R.

Ainsi vous pouvez écrire :

```
<second_semestre,eval=ymd(today()) > dmy("01-01-16")>>=
... graphique du 2nd semestre ...
@
```

Vous pouvez ainsi refaire des calculs en changeant la valeur de "cache" vous-mêmes si vous changez quelque chose dans les chunks précédents.

...

48 xtable

48.1 Le principe du package

Le principe du package *xtable* est qu'il permet d'exporter une *data.frame* sous la forme d'un tableau HTML ou LaTeX.

Il suffit pour cela d'appeler la fonction xtable avec une *data.frame* (ou d'autres objets comme les objets *lm* par exemple).

48.2 Le fonctionnement

```
<Iris,results=tex,echo=F>>=
require(xtable)
h <- head(iris,4)
tab <- xtable(h,
caption="Iris (Extrait)",
label="tab:Iris",
align=c("ccccc|c"),
digits=2
)
print(
 tab,
 type="latex",
 include.rownames=FALSE
)
@
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.10	3.50	1.40	0.20	setosa
4.90	3.00	1.40	0.20	setosa
4.70	3.20	1.30	0.20	setosa
4.60	3.10	1.50	0.20	setosa

Table 4 – Iris (Extrait)

48.3 Les options de xtable

Les possibilitées de *xtable* ne seront pas toutes évoquées...

Les principales options sont les suivantes :

caption la légende du tableau

label le label du tableau

align le terme d'alignement classique qu'on trouve après un *tabular*. Toutefois il faut noter que la première colonne est composée des *row.names*. Il faut donc k+1, avec k le nombres de colonnes, paramètres d'alignement.

digits le nombre de chiffres significatifs

display le type d'affichage, "d" pour les entiers, "s" pour les variables caractères, ...

48.4 Utilisation

On crée un objet de type *xtable* en appelant *xtable* avec une *data.frame* et les options choisies.

Puis il faut lancer son impression. Elle peut se faire sur la console (en *Sweave* directement dans le document LaTeXavec l'option *results=tex*) ou bien dans un fichier.

Dans un fichier cela permet de récupérer le résultat dans un fichier LaTeX avec un *include* ou un *input*.

48.5 Les options de print.xtable

Comme pour *xtable*, voici un bref aperçu des options...

type "latex" par défaut ou "html"

file le nom du fichier. Par défaut *NULL* ce qui signifie dans la console

floating *TRUE/FALSE* indique si le tableau est défini comme un flottant (par défaut *TRUE*)

table.placement pour le positionnement de la table, typiquement "htp". Par défaut "ht"

caption.placement bottom ou top...

NA.string ce qui doit être indiqué pour les *NA* par défaut ""

include.rownames *TRUE/FALSE*, indique si on veut les *row.names* ou non

...

49 Depuis R

49.1 Depuis la console R

Il suffira de taper cette commande :

```
require(knitr)
knit("02_Aide_sous_R.Rnw")
```

Le fichier *.tex* et le pdf sera créé dans le répertoire courant.

```
Le chargement a n???cessit??? le package : knitr


processing file: 18_Sweave.Rnw
  |...                                                              |   5%
```

```
  ordinary text without R code

  |......                                                           |  10%
label: intro (with options)
List of 1
 $ echo: symbol F

  |.........                                                        |  14%
  ordinary text without R code

  |............                                                     |  19%
label: HelloWorld
  |................                                                 |  24%
  ordinary text without R code

  |....................                                             |  29%
```

Pour obtenir le code R contenu dans le fichier *Rnw*, il suffira de taper :

```
purl("Mon fichier.Rnw")
```

Le fichier *.R* sera créé dans le répertoire courant.
Ou bien

```
require(knitr)
knit("Mon fichier.Rnw",purl=TRUE)
```

Le fichier *.R* sera créé dans le répertoire courant.
Il est à noter que parmi les options de *knit*, il y a les arguments :

output pour renommer le fichier créé

tangle pour appeler *automatiquement*

encoding pour changer l'encodage des fichiers. Utiles pour basculer de latin1 à Unicode ou inversement

49.2 Depuis R Studio

...Le bouton "Compile PDF"...
et tout ce qui a été dit précedemment sur la console R :

```
require(knitr)
knit("Mon fichier.Rnw",tangle=TRUE)
```

49.3 En cas d'erreur...

On obtiendra quelque chose finissant par...

```
Erreur :  chunk 10 (label = Plot)
Error in eval(expr, envir, enclos) :
  impossible de trouver la fonction "ploti"
Exécution arrêtée
```

Dans ce cas, dans le *chunk* 10 dont le label est *Plot*, une faute s'est glissée dans l'orthographe de la fonction plot.

50 En batch

50.1 Traitement conditionnel dans LaTeX

Cela offre des possbilités sans limite avec le package *ifthen* de LaTeX.

```
<Partie,results=tex,echo=F>>=
cat("\\newboolean{mavar1}")
cat("\\newboolean{mavar2}")
if ( TRUE ) cat("\\setboolean{mavar1}{true}")
if ( TRUE ) cat("\\setboolean{mavar2}{false}")
@
```

Plus loin dans le code LaTeX...

```
\ifthenelse{\boolean{mavar1}}{\LaTeX}{Pas \LaTeX}
\ifthenelse{\boolean{mavar2}}{\LaTeX}{Pas \LaTeX}
```

1 LaTeX
2 Pas LaTeX

50.2 Les options du mode Sweave

```
\newcounter{xyz}
\setcounter{xyz}{0}
<echo=F>>=
a <- 5
@
\whiledo{\value{xyz}<\ Sexpr{a}}%
{%
\thexyz
\stepcounter{xyz} \\%
}%
```

0
1
2
3
4

51 Documentation libre sur internet

51.1 Les manuels

Comme évoqué lors de la formation, le site R-project.org abrite les manuels de R. Ils sont une solide base sur R couvrant l'installation jusqu'à la création de paquets. On y trouve aussi la liste des fonctions R de base qui fait quelques milliers de pages.

Ils sont réalisés par le « noyau dur » des développeurs de R. Toutefois, en dehors de la liste des fonctions de R, ils restent assez succints.

51.2 Les documents suggérés...

Ils se trouvent sur la page contributed documentation.

Les ouvrages pour débuter les plus appréciés sont généralement :

- R pour les débutants, d'Emmanuel Paradis
- Brise Glace-R, traduction d'IcebreakeR.
- R pour les sociologues
- ...

Ces documents ont une approche basés sur les exemples essentiellement. Ils permettent de maîtriser les fonctions de base de R mais ne permettent généralement pas d'appréhender tout le potentiel de R.

Un ouvrage offrant un plus de distance est l'ouvrage de Vincent Goulet.

Pour ceux qui ne redoutent pas l'anglais, la lecture des documents de J. Faraway et F. Harrell Jr. sont très intéressants notamment pour ceux intéressés par les méthodes de régression.

Les documents de C. Genolini sont remarquables mais nécessitent une certaine maîtrise de R.

51.3 Ouvrages spécialisés

D'autres documents sont plus spécifiques comme ceux sur l'économétrie, l'actuariat, ... Quelques références supplémentaires :

- Pour l'analyse de questionnaires conatifs et cognitifs, le site Personnality Project qui abrite un ouvrage de très bonne facture en cours de rédaction.
- Oeuvre du RUG Element-R, un ouvrage sur la cartographie en français
- L'ouvrage de G. Sanchez sur le PLS Path Modeling
- TraMineR, pour l'analyse de trajectoires
- ...

52 Ouvrages payants

52.1 Statistiques

Dans cette section sont indiqués quelques livres intéressants avec un bref commentaire. L'opinion des formateurs n'étant pas parole d'évangiles vous pour-

rez trouver des informations sur les livres cités ci-dessous sur la page Books de R.

Couvrant une large palette de méthodes statistiques, l'ouvrage de ? (?) est un livre très intéressant qui permet de trouver rapidement comment réaliser ces méthodes avec R.

Le livre de ? (?) illustre différents méthodes à travers quelques exemples. Néanmoins sa qualité est un peu en retrait par rapport au livre de ? (?).

L'ouvrage de ? (?) est un livre d'initiation à la fois aux statistiques (simples) et à R. L'auteur fait partie des développeurs de R. Il est didactique et intéressant.

52.2 Langages et programmation

Le livre de ? (?) est un livre remarquable qui couvre le niveau débutant à avancé. Toutefois son approche est plus informatique que statistique. Il est possible que sa lecture soit déroutante si on a pas de bonnes connaissances en programmation.

Le livre de ? (?) est intéressant et s'adresse aussi à un public intermédiaire et averti. Il couvre notamment la manipulation de données.

L'ouvrage de ? (?) est précieux. Toutefois il s'adresse à un public averti. Il s'agit de la version papier de documents librement téléchargeables sur le web (voir partie précédente). Pour les parisiens, l'auteur est dans le RUG Semin-R.

Le livre de ? (?) est un must-have. Remarquable, il couvre les possibilités offertes par R en tant que langage fonctionnel. Il couvre aussi le débogage des fonctions R ainsi que la programmation objet. Mais il s'adresse à un public venant plus de l'informatique que des statistiques.

Les *blue book* de ? (?), illustrant des exemples statistiques avec R et le *yellow book* de (?, ?) sont des classiques. Le *yellow book* ne s'intéresse qu'au langage lui-même. L'intérêt de ces livres est surtout historique.

Le must-have est le livre de (?, ?)

52.3 Graphiques

Le must-have est le livre de (?, ?). Il couvre toutes les possibilités graphiques de R. Attention à bien acheter la deuxième version infiniment plus intéressante car elle couvre les graphiques traditionnels de R (présentés lors de la formation), le paquet *grid* et les packages *Lattice* et *ggplot2*.

Le package *Lattice* permet des réaliser des graphiques avancés. Il permet notamment d'automatiser la création de graphiques par variables catégorielles, faire des tableaux de graphiques aisement, ... Ce paquet dévelopé par un membre du R Core est en perte de vitesse. Son concurrent est *ggplot2* qui présente les mêmes fonctionnalités avec une qualité graphique supérieure et plus d'options.

Le paquet *Lattice* a son livre (?, ?) comme ggplot2(?, ?).

Le livre de ? (?) est le manuel du package. Une approche plus didactique est proposé dans le livre de (?, ?). Basé sur des exemples, il est pratique pour débuter mais l'ouvrage de (?, ?) reste la référence du *langage* de *ggplot2*.

A noter que le paquet ggplot2, a été développé dans l'esprit du livre de ? (?). Ce livre est très intéressant pour acquérir les bonnes pratiques en matière de graphiques.

52.4 Régression

Les ouvrages de ? (?), (?, ?) et ? (?) sont assez anciens. Mais ce sont d'excellents livres de référence et qui couvrent les régressions linéaires, logistique, ... Outre l'utilisation de R, ce sont d'excellents livres concernant la régression [1].

Un excellent livre en français sur la régression linéaire avec R de (?, ?) est paru chez Springer. Il mèle théorie et pratique avec R.

Plus complexe, l'ouvrage de ? (?) est très intéressant.

52.5 Applications particulières

Dans (?, ?), le package *survey* de l'auteur est décrit à travers des exemples de sondage simple, stratifié et à plusieurs degrés. Le problème est que l'auteur a énormement travaillé sur son package et beaucoup de fonctionnalités de son package ne sont pas présents dans le livre.

Le livre de référence pour l'analyse de données spatiales et la cartographie est l'ouvrage de ? (?).

Le livre de ? (?) est très intéressant et illustre bien les méthodes Monte-Carlo dans R.

Pour les analyses longitudinales, l'ouvrage de ? (?) offre beaucoup d'informations pour ceux qui sont intéressés par les mesures répétées et les études longitudinales.

Le livre d'? (?) couvre le problème des statistiques bayésiennes et la façon de réaliser les calculs dans R.

Le livre de ? (?) est un classique... Un must-have pour tous ceux qui veulent travailler avec des modèles mixtes sous R. Le livre est basé sur le paquet *nlme*. Un package plus récent *lme4* existe. Les deux paquets présente beaucoup de similitudes donc... Quelques nouveautés sont présentes *lme4* mais dont le temps d'exécution est beaucoup plus lent que *nlme*.

Un livre (?, ?) plus récent, orienté écologie mais d'une grande qualité, couvre les modèles mixtes, hiérarchiques, les GLMs, ...

Le livre de ? (?) est très spécifique. Il traite de la parallélisation des calculs sous R. Il permet de découvrir les différentes possibilités pour faire du calcul parallèle sous R. Sa parution est juste antérieure à l'intégration du package *parallel* par défaut dans R. Par conséquent il couvre la version beta de *parallel*. Si la parallélisation des calculs est très aisé sous GNU/Linux, il traite aussi des méthodes plus complexes (disponibles sous Windows et sous GNU/Linux) pour faire du calcul parallèle.

Pour l'analyse de questionnaire, un livre est disponible et a été écrit par un grand statisticien français. Il s'agit de l'ouvrage de ? (?). Il s'adresse à un public

1. Ils sont assez techniques

de niveau débutant à modéré. On y trouve les codes pour réaliser des analyses factorielles (à l'anglaise, sur variables latentes) et la validation de questionnaire. Il est assez orienté vers la médecine, l'auteur étant directeur d'un laboratoire de recherche en psychiatrie.

Pour le paquet FactoMineR, pour l'analyse de données à la française, deux livres sont disponibles.

Le premier sur les méthodes traditionnnelles (ACP, ACM, ...) (?, ?) et le second sur l'analyse factorielle multiple et l'analyse de données mixtes (?, ?).

52.6 Citations

Références