Initiation à R

DUT STID 1AFI

1 Présentation de R et de son environnement

1.1 Qu'est-ce que R?

Le logiciel R est un langage dédié à la statistique. C'est un langage interprété et non compilé, c'est-à-dire que les commandes tapées au clavier sont directement exécutées sans qu'il y ait besoin de construire un programme complet. Il a pour avantage d'être open source et d'être très utilisé par la communauté scientifique. Toute nouvelle méthode statistique est en effet très rapidement implémentée dans ce langage, ce qui n'est pas toujours nécessairement le cas pour d'autres langages.

Pour simplifier le développement de programmes R, il est nécessaire d'avoir une interface graphique. Celle offerte par R est très simpliste, beaucoup d'utilisateurs lui préfèrent donc `Rstudio`. Cet outil permet de fluidifier considérablement le travail avec ce langage.

Enfin, R, dans l'esprit open source, offre la possibilité de charger des *packages* ou *library* afin d'étendre ses possibilités. Il en existe un très grand nombre sur le site du **CRAN**, parfois redondants entre eux.

1.2 Ecrire du code

Sous R, il est possible d'exécuter des commandes en mode console, les unes après les autres, mais aussi d'écrire un code dans un script et d'exécuter par la suite ce script :

- mode console : les commandes doivent être tapées directement dans la console. On peut alors utiliser les flèches haut et bas pour naviguer dans l'historique des commandes.
- mode script (à privilégier) : plusieurs commandes peuvent être écrites et exécutées en une seule fois. Il est particulièrement utile pour stocker les différentes commandes et commentaires associés. Pour exécuter un script, il existe plusieurs possibilités : on peut tout sélectionner et cliquer sur `Run` (ou taper sur `Ctrl+Entrée`), on peut aussi mettre le curseur sur la première ligne et cliquer sur `Run` (ou taper sur `Ctrl+Entrée`) autant de fois qu'il y a de lignes dans le script. Enfin, on peut cliquer sur `Source` pour lancer l'intégralité du script.

1.3 Créer, lister et effacer les objets en mémoire

Un objet peut être créé avec l'opérateur *assigner* qui s'écrit `<-` :

```
n <- 15
n
```

```
## [1] 15
```

Si l'objet existe déjà, sa valeur précédente est effacée :

```
n <- 10 + 2
n
```

```
## [1] 12
```

Attention, R est sensible aux majuscules!

La fonction `ls()` permet d'afficher une liste simple des objets en mémoire, c'est-à-dire que seuls les noms des objets sont affichés. Ces objets sont également présents dans le panneau **Environnement**. Ils seront effacés lors de la fin de la session. Un autre moyen de les supprimer est d'utiliser la commande `rm()`.

```
name <- "Carmen"
n1 <- 10
n2 <- 100
ls()
```

```
## [1] "n1"   "n2"   "name"
```

1.4 L'aide en ligne

L'aide en ligne de R est très utile pour l'utilisation de fonctions, librairies et jeux de données. Elle est disponible directement pour une fonction donnée : `?mean` affichera par exemple la page d'aide pour la fonction `mean()`. La commande `help(mean)` aura le même effet.

2 Eléments de langage

Nous allons voir ici les différentes structures de données existantes, ainsi que les moyens de les créer et de les manipuler.

2.1 Les objets

R manipule différents types d'objets qui sont caractérisés par leur nom, leur mode (contenu de l'objet : `numeric`, `character`, `logical`, `factor`, `NULL`) et leur longueur (nombre d'éléments de l'objet).

```
x <- 1
mode(x) # donne le mode de x
```

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

```
length(x) # donne la longueur de x
```

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

```
x <- "hello"
mode(x)
```

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

```
length(x) # à ne pas confondre avec le nombre de lettres composant l'objet
```

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

```
nchar(x)
```

```
## [1] 5
```

```
x <- TRUE
mode(x)
```

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

Il existe différents types d'objets qui sont détaillés dans les sections suivantes. Les fonctions `is.<type>()` permettent de tester si l'objet en paramètre est du type demandé.

2.2 Type `vector`

Tout est `vector` de base puisqu'un scalaire est simplement un vecteur de longueur 1.

```
a <- 3
is.vector(a)
```

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

Les fonctions de base pour créer des vecteurs sont :

- `numeric()` (vecteur de mode `numeric`)
- `character()` (vecteur de mode `character`)
- `logical()` (vecteur de mode `logical`)
- `c()` (concaténation)

```
v <- c(1,2,5)
v
```

```
## [1] 1 2 5
```

Il est également possible de donner une étiquette à chacun des éléments d'un vecteur :

```
names(v) <- c("a","b","c") # donne une étiquette
v
```

```
## a b c
## 1 2 5
```

Pour créer des vecteurs de type `numeric`, d'autres commandes peuvent être utiles :

- `seq` pour des suites d'éléments

```
seq(1,5)
```

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

```
seq(1,5,by=.5) # spécifie le pas
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seq(1,5,length=6) # spécifie la longueur
```

```
## [1] 1.0 1.8 2.6 3.4 4.2 5.0
```

- `rep` pour des éléments répétés

```
rep(1:5,times=3) # spécifie le nombre de répétitions du vecteur
```

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

```
rep(1:5,each=2) # spécifie le nombre de répétitions de chaque élément
```

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

- `runif()`, `rnorm()` pour des vecteurs aléatoires

```
runif(5) # loi uniforme
```

```
## [1] 0.9969329 0.6806073 0.9351906 0.2991255 0.5529003
```

```
rnorm(5)  # loi normale centrée réduite
```

```
## [1]  0.9552684  1.2044306 -0.7175932 -0.2343122 -1.7782845
```

Dans un vecteur simple, tous les éléments doivent être de même mode.

```
v1 <- c(1,"hello")
v1
```

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

```
mode(v1)
```

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

L'indiçage dans un vecteur se fait avec les crochets `[ ]`. On peut extraire un élément d'un vecteur par sa position ou par son étiquette.

```
v
```

```
## a b c
## 1 2 5
```

```
v[2]
```

```
## b
## 2
```

```
v["b"]
```

```
## b
## 2
```

Il existe quatre façons d'extraire plusieur éléments d'un vecteur :

- avec un vecteur d'entiers positifs. Les éléments se trouvant aux positions correspondant aux entiers sont extraits du vecteur dans l'ordre :

```
v <- c(0,2,-3,4,-5)
names(v) <- c("a","b","c","d","e")
v[c(1,3)]
```

```
##  a  c
##  0 -3
```

- avec un vecteur d'entiers négatifs. Les éléments se trouvant aux positions correspondant aux entiers sont alors éliminés du vecteur :

```
v[-c(1,3)]
```

```
##  b  d  e
##  2  4 -5
```

- avec un vecteur booléen. Le vecteur d'indiçage doit alors être de même longueur que le vecteur indicé. Seuls les éléments correspondant à une valeur `TRUE` sont extraits du vecteur :

```
v>0
```

```
##     a     b     c     d     e
## FALSE  TRUE FALSE  TRUE FALSE
```

```
v[v>0]
```

```
## b d
## 2 4
```

- avec un vecteur de chaîne de caractères (utile pour extraire les éléments d'un vecteur à condition que ceux-ci soient nommés) :

```r
v[c("a","c")]
```

```
##  a  c
##  0 -3
```

2.3 Type `matrix` et `array`

Les matrices ne sont rien d'autre que des vecteurs dotés d'un attribut dimension de longueur 2 contenant les dimensions de la matrice. La fonction de base pour créer des matrices est `matrix`. La fonction `matrix` remplit naturellement la matrice par colonnes. L'argument `by.row` permet d'inverser l'ordre de remplissage.

```r
matrix(0,nrow=2,ncol=5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
```

```r
matrix(1:10,2,5)
```

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

```r
m <- matrix(1:10,2,5,byrow=TRUE)
m
```

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

```r
dim(m)
```

```
## [1] 2 5
```

On utilise les `[,]` pour accéder aux éléments de la matrice, avec tout ce qui est pour les lignes avant la virgule, et pour les colonnes, après. On peut aussi ne donner que la position de l'élément dans le vecteur sous-jacent. Lorsqu'une dimension est omise dans les crochets, tous les éléments de cette dimension sont extraits.

```r
m[1,2] # ligne 1, colonne 2
```

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

```r
m[c(1,2),3] # lignes 1 et 2, colonne 3
```

```
## [1] 3 8
```

```r
m[3] # élément 3 (à lire colonne par colonne)
```

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

```r
m[1,] # ligne 1
```

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

Des fonctions permettent de fusionner des matrices ayant au moins une dimension identique :

- `rbind` : fusionne ligne par ligne des matrices ayant le même nombre de colonnes,
- `cbind` : fusionne colonne par colonne des matrices ayant le même nombre de lignes.

La généralisation d'une matrice à plus de deux dimensions est un tableau `array`. La fonction de base pour créer des tableaux est `array()`.

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

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

2.4 Type `list`

La liste est le mode de stockage le plus général de `R`, permettant de mettre ensemble des éléments de différentes `type` et `mode` (y compris `list`, permettant ainsi d'emboîter des listes). La fonction de base pour créer des listes est `list()`.

```
l <- list(1:5,month.abb,pi)
l
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
##
## [[3]]
## [1] 3.141593
```

La liste demeure un vecteur. On peut donc l'indicer avec `[]`. Cependant, cela retourne une liste contenant le ou les éléments indicés. C'est rarement ce que l'on souhaite. Pour indicer un élément d'une liste et n'obtenir que cet élément, il faut utiliser `[[]]`. On ne peut alors extraire qu'un seul élément à la fois.

```
l[1]
```

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

```
l[[1]]
```

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

Si les éléments d'une liste sont nommés par des étiquettes, on peut alors utiliser l'opérateur $ à la place de `[[]]`.

```
l <- list(x=1:5,mois=month.abb,pi=pi)
l$x
```

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

La fonction `unlist()` convertit une liste en vecteur simple. Pour combiner deux listes, il suffit de les concaténer avec la fonction `c(list1,list2)`.

3 Traitement de données

3.1 Le format `data.frame`

Même si les vecteurs, matrices et listes sont les objets les plus fréquement utilisés sous `R`, un grand nombre de procédures statistiques reposent sur les `data.frame` pour le stockage des données. Bien que visuellement similaire à une matrice, un `data.frame` est plus général puisque les colonnes peuvent être de modes différents. La commande de base pour créer un `data.frame` est `data.frame()` ou `as.fata.frame()` pour convertir un autre type d'objets en data frame. On peut rendre les colonnes d'un data frame visibles dans l'espace de travail avec la fonction `attach`, puis les masquer avec `detach`. L'élément distinctif entre un data frame et une liste générale, c'est que tous les éléments du premier doivent être de même longueur et que, par conséquent, `R` les dispose en colonnes.

```
data(mtcars) # jeu de données existant sous R
str(mtcars) # contenu du jeu de données
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Pour accéder aux données, nous pouvons utiliser le même procédé que pour une `matrix` ou que pour une `list`.

```
mtcars[,1] # extraction de la colonne 1
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

```
mtcars["mpg"] # utilisation de l'indiçage
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

```
mtcars$mpg # spécificité du data.frame
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

3.2 Importation de données

Il existe des données déjà présentes dans le logiciel `R`, soit de base, soit suite à l'ajout d'un `package`. Pour avoir la liste de ces données, vous pouvez taper `data()`. Pour voir un jeu de données, il suffit de taper son nom dans la console (`mtcars` par exemple).

Pour accéder à son propre jeu de données, on a le choix entre deux types d'adressage pour situer le fichier :

- relatif : le chemin vers le fichier est indiqué par rapport au répertoire courant (ex : “fichier.ext” si le fichier est dans le répertoire courant)
- absolu : le chemin vers le fichier est entièrement défini et ne dépend pas du répertoire courant (ex : “C:/chemin/vers/fichier.ext”).

Il est toujours possible de définir le répertoire dans lequel travailler avec la commande `setwd()`. La commande `getwd()` permet quant à elle de savoir dans quel répertoire on travaille.

`R` peut lire des données stockées dans des fichiers texte (ASCII) à l'aide de la fonction `read.table()` (qui a plusieurs variantes qui ne sont pas dans le package de base). Cette fonction a pour effet de créer un tableau de données et est donc le moyen principal pour lire des fichiers de données.

```
my_data <- read.table("iris.csv")
str(my_data)
```

```
## 'data.frame':    151 obs. of  1 variable:
##  $ V1: Factor w/ 148 levels "4.3,3.0,1.1,0.1,setosa",..: 148 34 19 10 6 30 50 8 26 2 ...
```

Le séparateur de champ doit être défini suivant le fichier (`sep=\t` pour la tabulation par exemple). Les variables sont nommées par défaut `V1,...`. La commande `header` (`FALSE` ou `TRUE`) indique si le fichier contient les noms des variables sur la première ligne et permet donc ainsi d'attribuer des noms aux variables. Il existe d'autres options dont les détails peuvent être trouvés dans l'aide.

```
my_data <- read.table("iris.csv",sep=",",header=TRUE)
str(my_data)
```

```
## '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 ...
```

Parmi les variantes existantes, on trouve `read.csv()`, `read.csv2()`, `read.delim()` avec des valeurs par défaut légèrement différentes.

3.3 Les données manquantes

Dans les applications statistiques, il est souvent utile de pouvoir représenter des données manquantes. Dans `R`, l'objet spécial `NA` remplit ce rôle. Par défaut, le mode de `NA` est `logical`. Toute opération impliquant une donnée `NA` a comme résultat `NA` mais certaines fonctions (`sum`, `mean`,...) ont un argument `na.rm` qui, lorsque `TRUE`, élimine les données manquantes. La valeur `NA` n'est égale à aucune autre, pas même à elle même. Par conséquent, pour tester si les éléments d'un objet sont `NA` ou non, il faut utiliser la fonction `is.na`.

```
is.na(NA)
```

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

```
NA==NA
```

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

3.4 Le type `factor` pour les variables qualitatives

`R` dispose d'un type particulier permettant de coder des variables qualitatives : le type `factor`. On le retrouve dans de nombreux `data.frame`. Un `factor` est un vecteur avec une liste prédéfinie de valeur : les niveaux

(`levels`). Comme nous pouvons le voir en transformant la variable en `numeric`, chaque modalité est codée numériquement, la liste des niveaux étant stockée séparément.

```
class(iris$Species)
```

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

```
levels(iris$Species)
```

```
## [1] "setosa"     "versicolor" "virginica"
```

```
iris$Species[c(1:3,63:64)]
```

```
## [1] setosa     setosa     setosa     versicolor versicolor
## Levels: setosa versicolor virginica
```

```
as.numeric(iris$Species[c(1:3,63:64)])
```

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

```
as.character(iris$Species[c(1:3,63:64)])
```

```
## [1] "setosa"     "setosa"     "setosa"     "versicolor" "versicolor"
```

4 Opérateurs et fonctions

4.1 Opérations arithmétiques

Comme dans tous les langages, nous avons accès aux opérateurs mathématiques classiques : `+`, `-`, `*`, `/`, `^`, `sqrt()` (racine carrée). En ce qui concerne les vecteurs, les opérations sont effectuées élément par élément :

```
x<- c(1,2,3)
y <- c(4,5,6)
x + y
```

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

```
x*y
```

```
## [1]  4 10 18
```

Si les vecteurs impliqués ne sont pas de la même longueur, les plus courts sont recyclés de façon à correspondre au plus long vecteur.

```
1:10 + 2
```

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

Si la longueur du plus long vecteur est un multiple de celle du (ou des) autres vecteurs, ces derniers sont recyclés un nombre entier de fois. Sinon, le plus court vecteur est recyclé un nombre fractionnaire de fois mais comme ce résultat est rarement souhaité, un avertissement est affiché.

```
1:10 + 1:5 +rep(c(2,4),5)
```

```
##  [1]  4  8  8 12 12 11 11 15 15 19
```

```
1:10 + rep(1:5,2) +rep(c(2,4),5)
```

```
##  [1]  4  8  8 12 12 11 11 15 15 19
```

```r
1:10 + c(2,4,6)
```

```
## Warning in 1:10 + c(2, 4, 6): la taille d'un objet plus long n'est pas
## multiple de la taille d'un objet plus court
```

```
##  [1]  3  6  9  6  9 12  9 12 15 12
```

Pour les matrices, la commande `*` effectue la multiplication des matrices terme à terme. Si l'on veut effectuer un produit matriciel, il faut utiliser `%*%`.

4.2 Quelques fonctions utiles

Le langage `R` compte un très grand nombre de fonctions internes permettant de :

- extraire des éléments particuliers de vecteurs :

```r
head(1:10) # début du vecteur
```

```
## [1] 1 2 3 4 5 6
```

```r
tail(1:10) # fin du vecteur
```

```
## [1]  5  6  7  8  9 10
```

```r
unique(c(2,4,2,5,9,5,1)) # élimine les doublons
```

```
## [1] 2 4 5 9 1
```

- ordonner des vecteurs :

```r
sort(c(4,-1,2,6)) # ordonne les vecteurs
```

```
## [1] -1  2  4  6
```

```r
rank(c(4,-1,2,6)) # donne les rangs des éléments du vecteur si on les ordonne
```

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

```r
order(c(4,-1,2,6)) # donne les positions des éléments du vecteur une fois ordonnés
```

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

```r
rev(c(4,-1,2,6)) # renverse l'ordre du vecteur
```

```
## [1]  6  2 -1  4
```

- rechercher les positions des éléments dans un vecteur :

```r
x <- c(4,-1,2,-3,6)
which(x>0) # position des éléments satisfaisant la condition
```

```
## [1] 1 3 5
```

```r
which.min(x) # position du min
```

```
## [1] 4
```

```r
which.max(x) # position du max
```

```
## [1] 5
```

```r
match(2,x) # position des éléments égaux à 2
```

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

```r
x %in% c(-1:2) # vecteur booléen TRUE/FALSE satisfaisant la condition
```

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

- arrondir les éléments d'un vecteur :

```r
x <- 10*runif(5)
x
```

```
## [1] 7.048267 8.298033 9.875241 2.222043 8.914519
```

```r
round(x) # arrondi
```

```
## [1]  7  8 10  2  9
```

```r
round(x,3) # arrondi à 3 chiffres après la virgule
```

```
## [1] 7.048 8.298 9.875 2.222 8.915
```

```r
floor(x) # partie entière
```

```
## [1] 7 8 9 2 8
```

- faire des opérations sur les matrices :

```r
x <- matrix(1:4,nrow=2,ncol=2)
x
```

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

```r
nrow(x) # nombre de lignes
```

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

```r
ncol(x) # nombre de colonnes
```

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

```r
rowSums(x) # somme de chaque ligne
```

```
## [1] 4 6
```

```r
rowMeans(x) # moyenne de chaque ligne
```

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

```r
t(x) # transposée de x
```

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

```r
diag(x) # diagonale de x
```

```
## [1] 1 4
```

4.3 Boucles

Les boucles doivent être utilisées avec parcimonie en `R` car elles sont généralement inefficaces. Dans la majeure partie des cas, il est possible de vectoriser les calculs pour éviter les boucles explicites ou encore d'utiliser les fonctions `apply, sapply, tapply, mapply, lapply` pour réaliser des boucles de manière plus efficace.

Les boucles s'écrivent sous la forme suivante :

- `if (condition) branche.vraie else branche.fausse` : si `condition` est vraie, `branche.vraie` est exécutée, sinon ce sera `branche.fausse`

```
x <- -2
if (x<0){
  cat("x est négatif") # affiche le message
} else {
  cat("x est positif")
}
```

```
## x est négatif
```

- `for (variable in suite) expression` : exécute `expression` successivement pour chaque valeur de `variable` contenue dans `suite`

```
x <- 0
for (i in (1:10)){
  x <- x+1
}
x
```

```
## [1] 10
```

- `while (condition) expression` : exécute `expression` tant que `condition` est vraie

```
x <- 10
while (x>0){
  x <- x-1
}
x
```

```
## [1] 0
```

4.4 Fonctions définies par l'usager

On définit une nouvelle fonction avec la syntaxe suivante : `fun <- function(arguments) expression` où

- `fun` est le nom de la fonction
- `arguments` est la liste des arguments, séparés par des virgules
- `expression` constitue le corps de la fonction, soit une expression ou un groupe d'expressions réunies par des accolades.

La plupart des fonctions sont écrites dans le but de retourner un résultat. Ici, une fonction retourne tout simplement le résultat de la dernière expression du corps de la fonction. On peut utiliser la fonction `return()` pour retourner un résultat qui n'est pas à la dernière ligne de la fonction.

```
Moyenne <- function(x){
  if (mode(x)=="numeric"){
    M <- mean(x)
    return(M)
  } else {
    cat("x n'est pas numérique")
  }
}
Moyenne(c(1,3,5))
```

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

```
Moyenne(c("a","b","c"))
```

```
## x n'est pas numérique
```

5 Statistiques descriptives

La fonction `summary()` calcule des statistiques basiques sur un vecteur, celles-ci étant dépendantes du type du vecteur. Si elle est appliquée sur un `data.frame`, elle s'applique sur chaque variable.

```
summary(mtcars$mpg)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.43   19.20   20.09   22.80   33.90
```

5.1 Analyse statistique univariée

Pour les variables quantitatives, on peut accéder aux fonctions de calcul de statistiques descriptives qui sont directement codées sous `R`.

```
mean(mtcars$mpg) # moyenne
```

```
## [1] 20.09062
```

```
var(mtcars$mpg) # variance
```

```
## [1] 36.3241
```

```
sd(mtcars$mpg) # écart-type
```

```
## [1] 6.026948
```

```
min(mtcars$mpg) # min
```

```
## [1] 10.4
```

```
max(mtcars$mpg) # max
```

```
## [1] 33.9
```

```
range(mtcars$mpg) # (min, max)
```

```
## [1] 10.4 33.9
```

```
median(mtcars$mpg) # médiane
```

```
## [1] 19.2
```

```
quantile(mtcars$mpg) # quantile
```

```
##     0%    25%    50%    75%   100%
## 10.400 15.425 19.200 22.800 33.900
```

```
quantile(mtcars$mpg,probs=0.99)
```

```
##    99%
## 33.435
```

Pour les variables qualitatives et quantitatives discrètes, la fonction `table()` (ou `prop.table()`) permet de tracer un tableau de contingence.

```
table(mtcars$am) # tableau de contingence
```

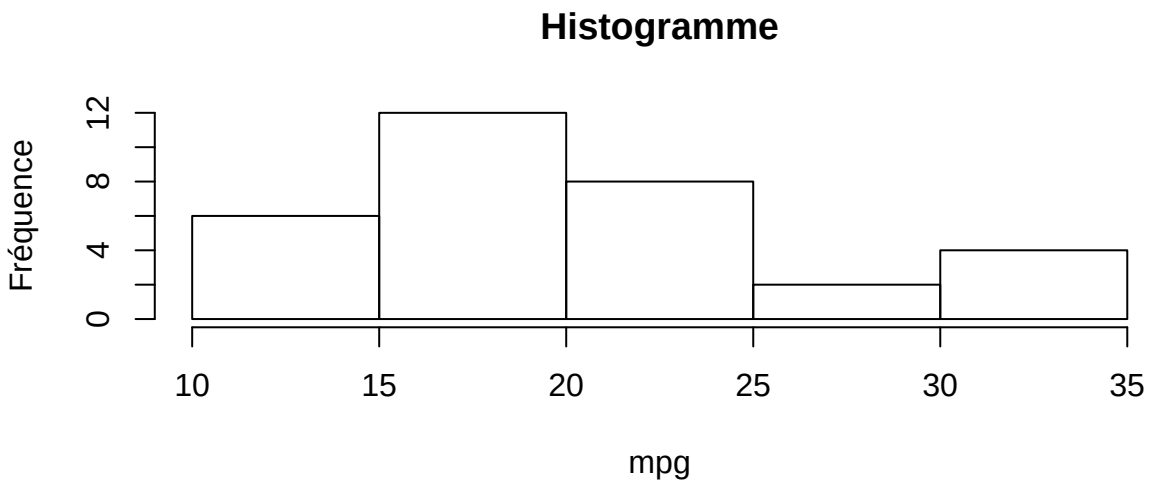
```
##
##  0  1
## 19 13
```

```
prop.table(table(mtcars$am)) # tableau des proportions
```

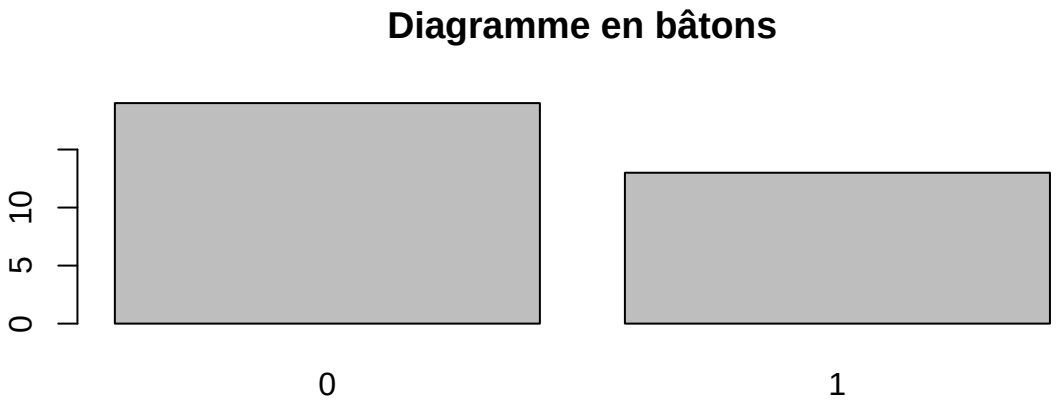
```
##
##       0       1
## 0.59375 0.40625
```

Il existe tout un ensemble de fonctions graphiques dans `R` permettant de tracer, pour les variables quantitatives, des histogrammes, et pour les variables qualitatives, des diagrammes en bâtons et des boîtes à moustaches notamment.

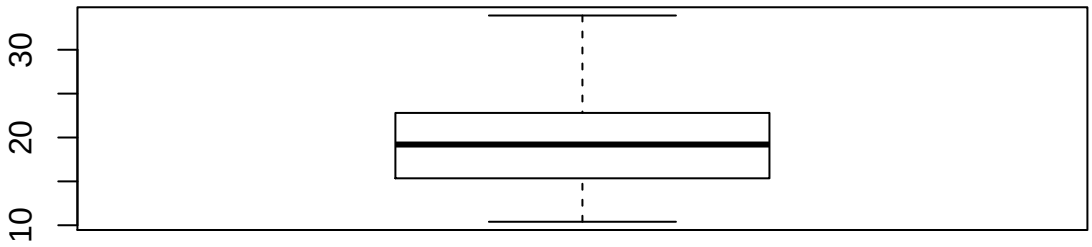
```
hist(mtcars$mpg, # commande de base pour l'histogramme
    xlab = "mpg", # légende de l'axe x
    ylab = "Fréquence", # légende de l'axe y
    main = "Histogramme") # titre du graphique
```




```
t <- table(mtcars$am) # tableau de contingence pour la variable am
barplot(t, # commande de base pour les diagrammes en bâtons
        main="Diagramme en bâtons",xlab="Manuel ou automatique")
```




```r
boxplot(mtcars$mpg) # commande de base pour les boîtes à moustaches
```

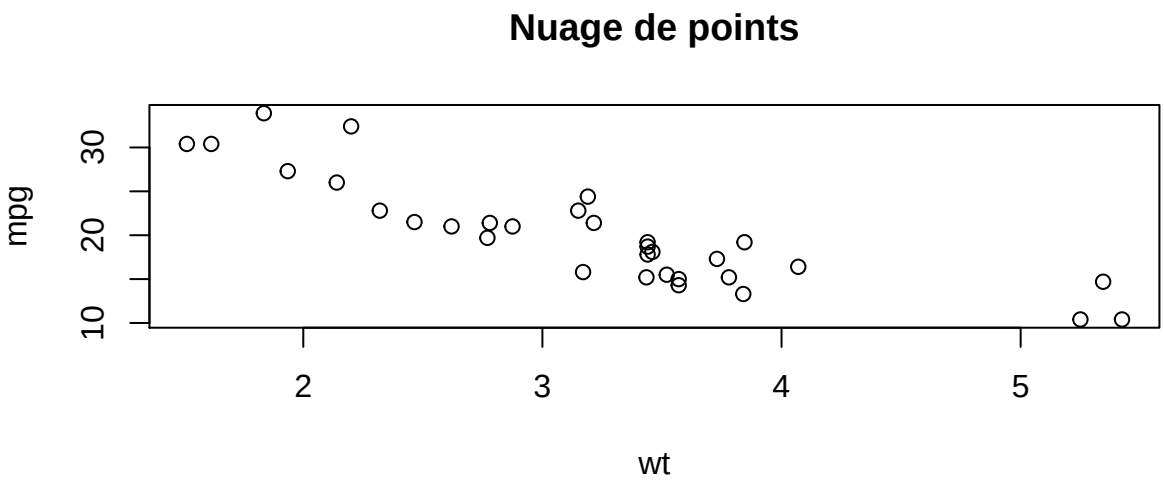


5.2 Analyse statistique bivariée

Il faut distinguer plusieurs cas :

- Quantitatif vs. quantitatif : dans ce cas, on peut calculer les statistiques usuelles (covariance, corrélation) et tracer un nuage de points

```r
plot(x = mtcars$wt, y = mtcars$mpg, # commande de base pour les nuages de point
     xlab="wt", ylab="mpg", main = "Nuage de points") # noms des axes et titre
```




```r
cov(mtcars$mpg, mtcars$wt) # covariance
```

```
## [1] -5.116685
```

```r
cor(mtcars$mpg, mtcars$wt) # corrélation
```

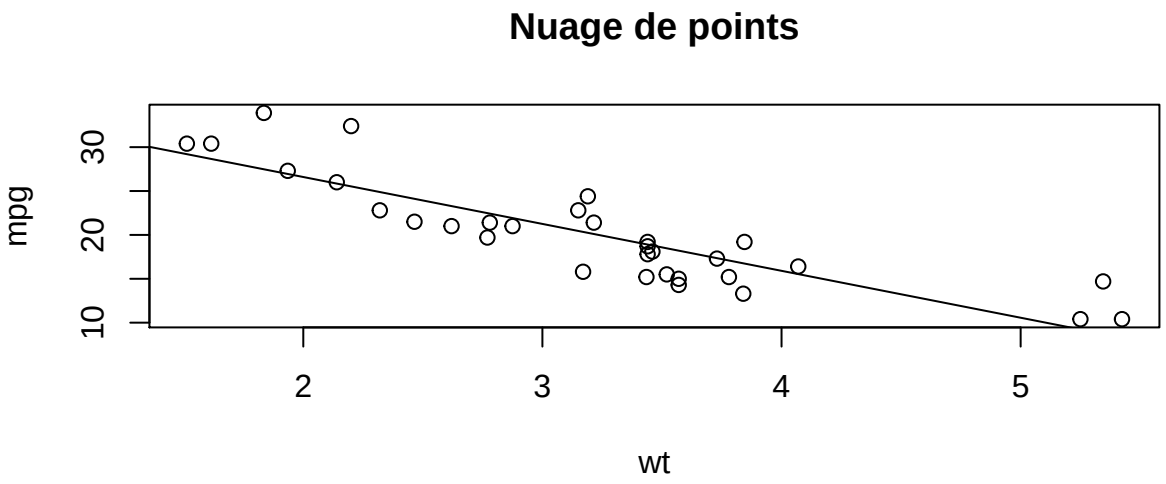
```
## [1] -0.8676594
```

On peut aller plus loin en ajustant un modèle linéaire simple aux données.

```r
model <- lm(mpg ~ wt, # définition du modèle mpg = alpha * wt + beta avec la fonction lm()
            data=mtcars) # jeu de données utilisé
# De manière équivalente
# model <- lm(mtcars$mpg ~ mtcars$wt)
model # on retrouve les estimateurs du modèle linéaire
```

```
##
## Call:
## lm(formula = mpg ~ wt, data = mtcars)
##
## Coefficients:
## (Intercept)           wt
##      37.285       -5.344
```

```r
plot(y = mtcars$mpg, x = mtcars$wt, xlab="wt", ylab="mpg", main = "Nuage de points")
abline(model) # permet d'ajouter une droite à un graphique
```

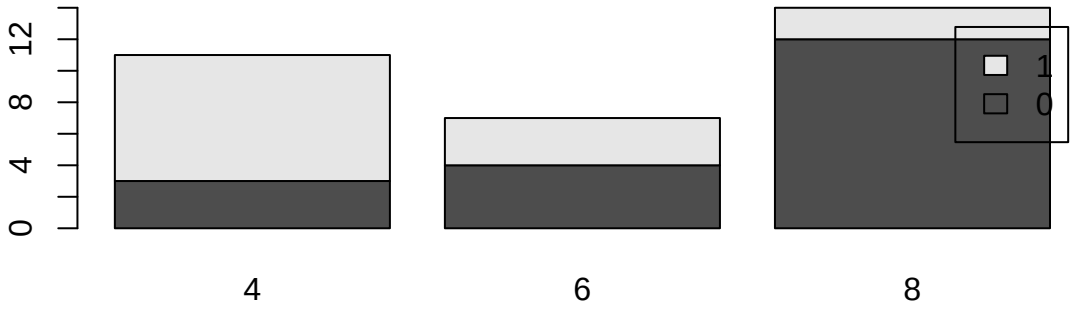


- Qualitatif vs. qualitatif : dans ce cas, on peut tracer un tableau croisé et des diagrammes en barres.

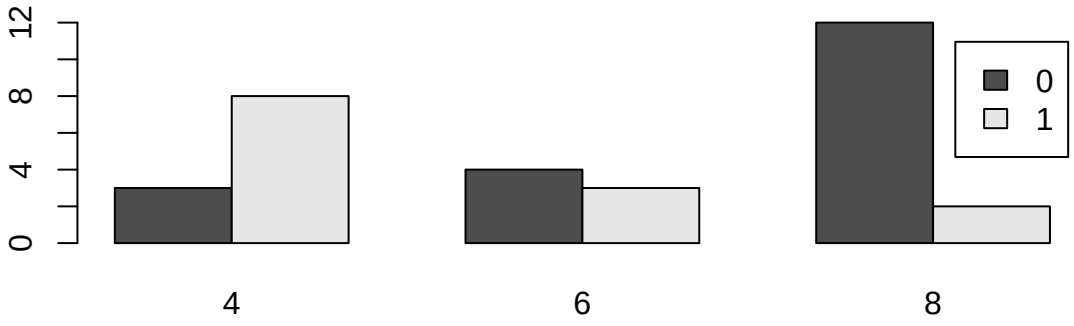
```r
t <- table(mtcars$am,mtcars$cyl) # tableau croisé
t
```

```
##
##      4  6  8
##   0  3  4 12
##   1  8  3  2
```

```r
barplot(t, # commande de base pour le diagramme en bâtons
        legend = c("0","1")) # définition de la légende
```



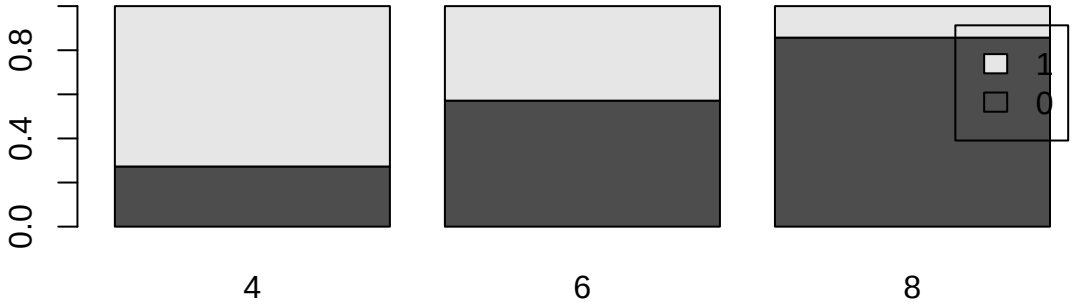

```
barplot(t,beside = TRUE, legend = c("0","1")) # autre représentation
```




```
prop.table(t,2) # tableau de proportion marginal
```

```
##
##             4         6         8
##   0 0.2727273 0.5714286 0.8571429
##   1 0.7272727 0.4285714 0.1428571
```

```
barplot(prop.table(t,2), legend = c("0","1")) # diagrammme en bâtons ajusté
```

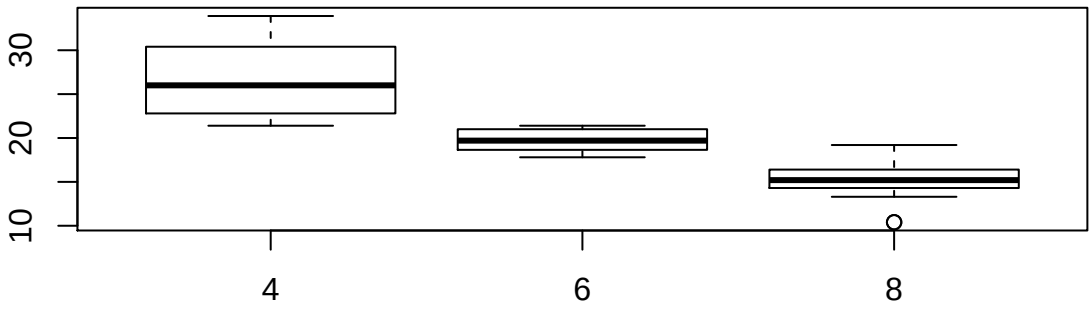


- Qualitatif vs. quantitatif : dans ce cas, en plus d'obtenir les statistiques par modalité de la variable qualitative, on peut représenter les boîtes à moustaches.

```
tapply(mtcars$mpg, mtcars$cyl, mean) # applique la fonction moyenne aux modalités cyl
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

```
boxplot(mtcars$mpg ~ mtcars$cyl) # boîtes à moustaches par modalité cyl
```



5.3 Introduction à ggplot2

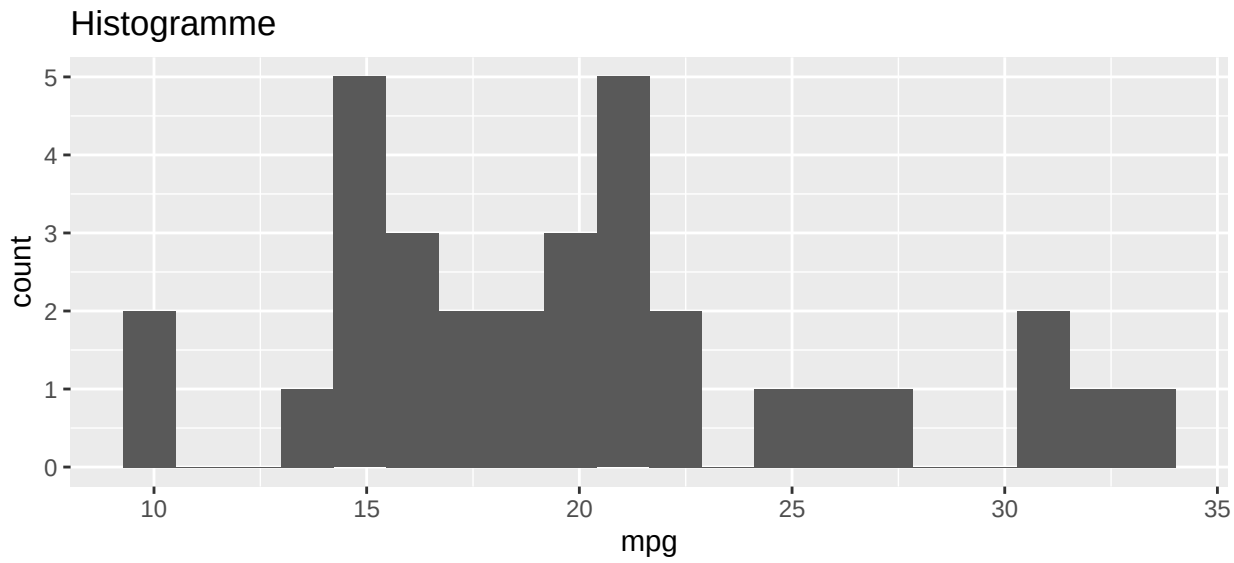
Le package `ggplot2` est l'un des plus connus pour la visualisation de données. Il dispose d'une documentation très complète (https://ggplot2.tidyverse.org/reference/). Parmi les fonctions disponibles, la fonction `ggplot()` permet d'initialiser un graphique et d'y ajouter des couches successivement :

- `ggplot()` créé un graphique (et le renvoie ou le stocke dans une variable pour éventuellement lui ajouter des couches),
- `aes()` définit les aspects esthétiques (axes `x` et `y`, `color`, `fill`, `size`,...),
- `geom_xxx()` indique la représentation à choisir (`xxx` peut être remplacé par `histogram`, `boxplot`,...),
- `stat_xxx` indique les transformations statistiques à utiliser,
- `coord_xxx` modifie les systèmes de coordonnées,
- `facet_grid()` découpe le graphique en plusieurs facettes selon la variable choisie,
- `theme_xxx`, `labs()`, `xlab()`, `ylab()`, `ggtitle()`,... améliore le rendu visuel du graphique.

Hormis la fonction `aes()` qui s'utilise à l'intérieur des autres, toutes ces fonctions peuvent s'aditionner pour compléter le graphique.

Voici des exemples de code permettant de générer les représentations décrites dans la section précédente :

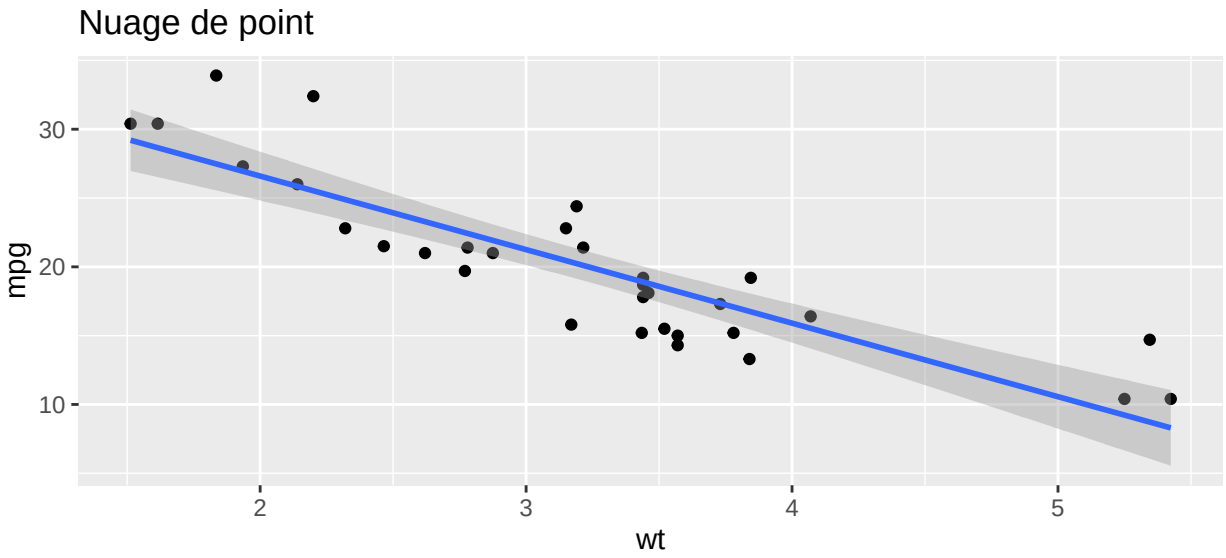
```
library(ggplot2)
# Histogramme
ggplot(mtcars, aes(mpg)) + # création du graphique avec pour variable d'intérêt mpg
  geom_histogram(bins=20) + # histogramme à 20 barres
  ggtitle("Histogramme")
```



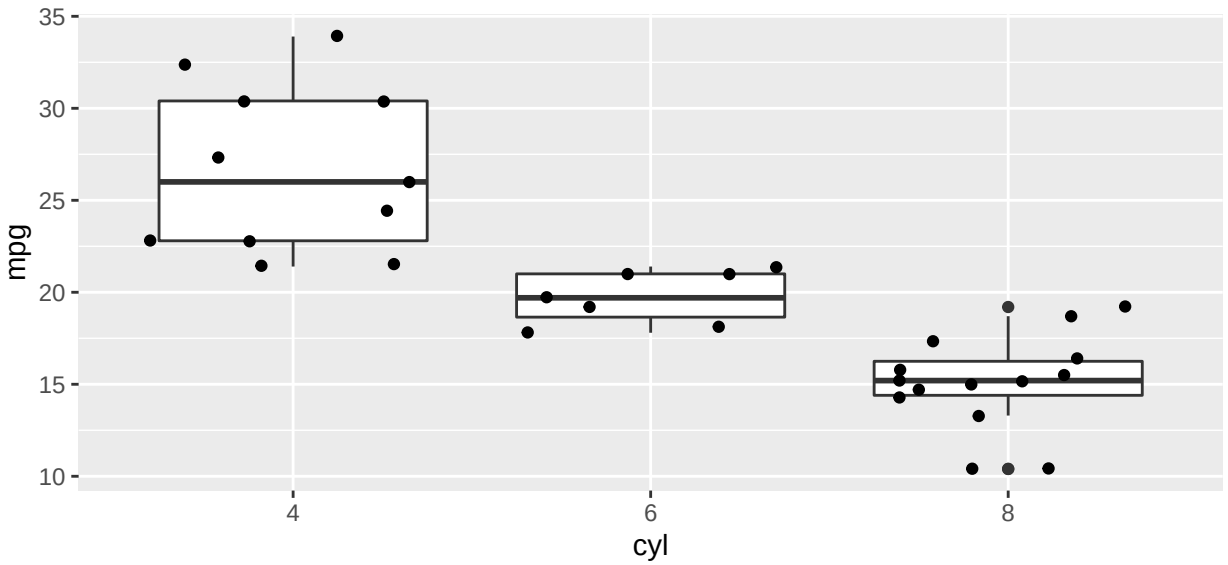

```
# Nuage de points
model <- lm(mpg ~ wt, data=mtcars) # modèle linéaire

ggplot(data = mtcars, aes(wt, mpg)) +  # création du graphique et définition des axes
          ggtitle("Nuage de point") + # ajout d'un titre
          geom_point() + # représentation sous forme de nuage de points
          geom_smooth(method="lm") # ajout du modèle linéaire
```

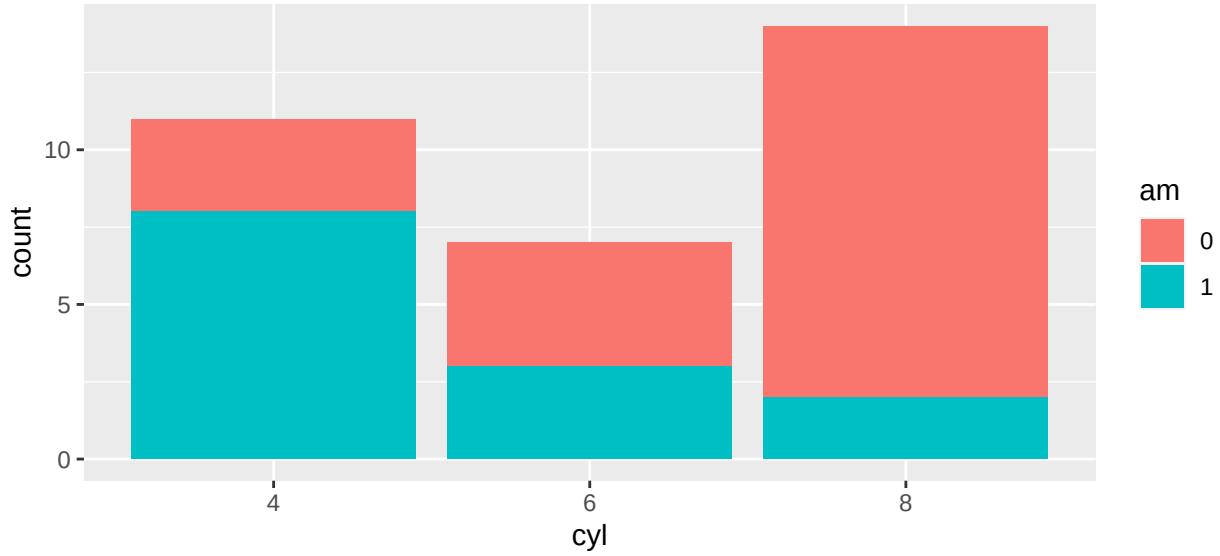
```
## `geom_smooth()` using formula 'y ~ x'
```




```
# Boîtes à moustache par modalité
ggplot(data = mtcars, aes(factor(cyl),mpg)) + # les modalités doivent être des facteurs
    xlab("cyl") + # nom de l'axe x
    geom_boxplot() + # représentation sous forme de boîte à moustaches
    geom_jitter() # ajout des points
```




```r
# Diagramme en bâtons par modalité
ggplot(mtcars, aes(factor(cyl), fill = factor(am))) + # les modalités doivent être des facteurs
  xlab("cyl") + # nom de l'axe x
  labs(fill="am") + # nom de la légende
  geom_bar() # représentation sous forme de barres
```



6 Concepts avancés

6.1 Fonction `apply()`

La fonction `apply()` sert à appliquer une fonction quelconque sur une partie d'une matrice, en évitant d'écrire des boucles. La syntaxe de la fonction est la suivante `apply(X, MARGIN, FUN, ...)` où

- `X` est une matrice,
- `MARGIN` indique la dimension (ligne 1, colonne 2) sur laquelle la fonction doit s'appliquer,
- `FUN` est la fonction à appliquer.

```r
x <- matrix(sample(1:100, 15, rep = TRUE), 3, 5)
x
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   89   17   18  100   54
## [2,]    1   96   85    5   66
## [3,]   43   29   41    2   93
```

```r
apply(x,1,var) # calcul de la variance pour chacune des lignes de x
```

```
## [1] 1498.3 2005.3 1092.8
```

```r
apply(x,2,min) # calcul du minimum pour chacune des colonnes de x
```

```
## [1]  1 17 18  2 54
```

6.2 Fonctions `lapply()` et `sapply()`

Les fonctions `lapply()` et `sapply()` sont similaires à la fonction `apply()` puisqu'elles permettent d'appliquer une fonction aux éléments d'une structure (vecteur ou liste). Plus précisément, la fonction `lapply()` applique une fonction `FUN` à tous les éléments d'un vecteur ou d'une liste `X` et retourne le résultat sous la forme d'une liste. La fonction `sapply()` est similaire, sauf que le résultat est retourné sous la forme d'un vecteur (dans la mesure du possible, le résultat est donc simplifié). Elles ont la même syntaxe : `lapply(X,FUN,...)` et `sapply(X,FUNC,...)`.

```r
x <- list(c(4,1,8,3,5),c(2,8,9,1,4,6),c(9,4,8,10,7,1,2))
x
```

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

```r
lapply(x,mean) # calcule la moyenne de chaque élément de la liste
```

```
## [[1]]
## [1] 4.2
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] 5.857143
```

```r
sapply(x,mean) # idem mais le résultat est donné sous la forme d'un vecteur
```

```
## [1] 4.200000 5.000000 5.857143
```

6.3 Fonction particulière

On a parfois besoin d'utiliser une fonction spécifique dans les fonctions comme `lapply()` ou `sapply()`. On peut alors la définir au préalable et l'utiliser comme une autre.

```r
l = list(a = "chaîne",
  b = 12,
  c = 1:10,
  d = head(mtcars),
  e = list(x = 1:10, y = log(1:10)))

infoElement <- function(e) {
    return(c(classe = class(e), longueur = length(e)))
} # fonction permettant de retourner la classe et la longueur d'un objet

sapply(l, infoElement) # applique la fonction infoElement à tous les éléments de la liste
```

```
##          a           b         c         d            e
## classe   "character" "numeric" "integer" "data.frame" "list"
## longueur "1"         "1"       "10"      "11"         "2"
```

Il est également possible de déclarer directement la fonction dans les fonctions `lapply()` et `sapply()`. On parle alors de fonction anonyme.

```r
sapply(l, function(e) {
  return(c(classe = class(e), longueur = length(e)))
})
```

```
##          a           b         c         d            e
## classe   "character" "numeric" "integer" "data.frame" "list"
## longueur "1"         "1"       "10"      "11"         "2"
```

7 Reporting avec Rmarkdown

R permet de produire des documents (HTML, pdf entre autres) incluant à la fois du texte et des commandes R de manière automatique avec Rmarkdown. A titre d'information, ce document a été créé grâce à Rmarkdown. Il est à la fois très simple à utiliser et à lire, à privilégier donc pour écrire les compte-rendus de TP R.

7.1 Document Rmarkdown

Pour créer un document Rmarkdown dans Rstudio, il faut cliquer *Fichier*, puis *Nouveau* et *Rmarkdown*. Une interface apparaît alors demandant de choisir entre un document, une présentation, une application Shiny ou de choisir une mise en page prédéfinie. Pour rester sur le mode *document* (qui est ce qui nous intéresse), on peut alors indiquer le titre et l'auteur, et choisir le format de sortie. Ce document Rmarkdown sera enregistré avec l'extension .rmd.

La première partie du document est son en-tête. Elle se situe en tout début de document, délimité par trois tirets avant et après. Elle contient les métadonnées du document comme le titre, le ou les auteurs, la date et les options de sortie.

```
---
title: "Mon document Rmarkdown"
author: "Prénom Nom"
date: "Année 2018-2019"
output: html_document # produit un doc hmtl
      # remplacer par pdf_document pour un pdf (nécessite que latex soit installé)
---
```

7.2 Ecrire du texte

Si vous connaissez LaTeXet que l'output est `pdf_document`, le texte peut être entièrement écrit en LaTeX, ce qui est particulièrement utile pour écrire des formules mathématiques. Dans le cas contraire, la syntaxe utilisée est le langage `markdown`.

Les titres sont écrits avec des `#` en début de ligne, pour produire des titres de niveau 1 (`<h1>`) à 6 (`<h6>`).

```
# Titre de niveau 1
## Titre de niveau 2
...
###### Titre de niveau 6
```

Les paragraphes s'écrivent simplement. Attention cependant, un passage à la ligne ne suffit pas à changer de paragraphe, il est nécessaire de laisser au moins une ligne entre deux paragraphes.

```
blablabla
blablabla # pas de changement de paragraphe

blablabla # changement de paragraphe
```

Pour mettre en gras (italique) un ou plusieurs mots, on les encadre par des `**` (`*`). Pour faire un lien vers une page web, on peut uiliser la syntaxe `<url>` dans laquelle on inclus le lien vers la page concernée.

```
Ce **mot** sera en gras, celui-là en *italique*.
```

Ce **mot** sera en gras, celui-là en *italique*.

Pour terminer, il est possible de créer des listes (non-ordonnées et ordonnées). Ces listes doivent être séparées des paragraphes précédents et suivants par, au moins, une ligne. De plus, pour les sous-listes, il faut mettre une tabulation supplémentaire.

Le code suivant :

```
# création d'une liste (et sous-liste) non-ordonnée
* premier élément

* deuxième élément

    * sous-élément

    * sous-élément

# création d'une liste (et sous-liste) ordonnée
1. premier élément

2. deuxième élément

    1. sous-élément

    2. sous-élément
```

produira :

- premier élément
- deuxième élément
  - sous-élément
  - sous-élément

1. premier élément
2. deuxième élément
    1. sous-élément
    2. sous-élément

7.3 Inclure du code

Pour créer un bloc de code (*chunck*), il faut utiliser la syntaxe suivante :

```
```{r}
# ici on met notre code
```
```

Pour inclure du code au sein d'une phrase, il suffira d'utiliser l'apostrophe simple, ce qui permettra de produire par exemple `mtcars`.

Lorsque le document est compilé au format pdf ou HTML, chaque bloc est exécuté tour à tour et le résultat est inclus dans le document final. Les blocs sont liés entre eux, dans le sens où les données importées ou calculées dans un bloc sont accessibles aux blocs suivants.

Il existe des options permettant de modifier le comportement de ces blocs de code. Ces options sont à placer à l'intérieur des accolades, séparées de `r` par une virgule :

- `nom_du_bloc` ajoute un nom (doit être défini de manière unique) au bloc,
- `echo=TRUE/FALSE` permet de rendre visible/cacher le code `R`,
- `eval=TRUE/FALSE` exécute (ou non) le code `R` à la compilation,
- `warning=TRUE/FALSE` affiche (ou non) les avertissements générés par le bloc,
- `message=TRUE/FALSE` affiche (ou non) les messages générés par le bloc.

Il est tout à fait possible d'inclure des tableaux et des graphiques qui sont produits par `R` au document final en les ajoutant à un bloc de code.

Par défaut, les tableaux issus de la fonction `table` sont affichés comme ils apparaissent dans la console de `R`.

```
##
##  0  1
## 19 13
```

On peut utiliser la fonction `kable` de la librairie `knitr` pour améliorer leur présentation. On aura alors des tableaux plus adaptés.

Var1	Freq
0	19
1	13

7.4 Compiler un document

On peut à tout moment compiler un document `Rmarkdown` pour obtenir et visualiser le document généré dans le format voulu choisi à l'aide du bouton `Knit` (*tricoter* en anglais). Un onglet `Rmarkdown` s'ouvre alors dans la même zone que l'onglet console et indique la progression de la compilation (ainsi que les messages d'erreur éventuels). Si tout va bien, le document s'affiche dans la fenêtre `Viewer` de `Rstudio`, soit dans le logiciel par défaut de votre ordinateur.

8 Introduction à Shiny

`R` permet de réaliser des applications web interactives (dites WebApps) qui rendent plus dynamiques les explorations de données et les résultats d'analyse statistiques grâce à l'utilisation de `Shiny`. La grande force de `Shiny` provient du fait qu'il n'y ait absolument pas besoin de connaître les langages HTML, CSS, JavaScript puisque tout est codé en `R`.

Il existe de nombreux tutoriaux et exemples sur internet pour le développement d'applications `Shiny`. On peut par exemple avoir un aperçu des possibilités offertes par le package avec la galerie Shiny (http://shiny.rstudio.com/gallery/), le Shiny User Showcase (https://www.rstudio.com/products/shiny/shiny-user-showcase/) et le site Show me Shiny (http://www.showmeshiny.com/).

8.1 Structure d'une application `shiny`

Une application fait intervenir les notions suivantes :

- développeur : créateur(s) de l'application,
- utilisateur : personnes accédant à l'application,
- interface (UI) : ensemble des éléments visibles par l'utilisateur comprenant les entrées (*inputs*) que l'utilisateur pourra modifier et les sorties (*outputs*) qui sont générés automatiquement par l'application à partir des choix de l'utilisateur,
- exécution : calcul des éléments de sorties à partir des éléments d'entrée.

Pour créer une application `shiny` dans `Rstudio`, il faut cliquer sur *Fichier*, puis *Nouveau* et *Shiny Web App*. Une interface apparaît alors demandant de choisir le nom de l'application, le chemin de sauvegarde et le nombre de fichiers à créer correspondants (*single* ou *multiple*). Une application `shiny` est en effet composée de deux parties distinctes `server.R` et `ui.R` qui peuvent être réunies dans un seul et même fichier `app.R` :

- `server.R` définit les opérations à réaliser (exécution) lors de la création de la page et lors des différentes interactions avec l'utilisateur,
- `ui.R` définit l'interface utilisateur, c'est-à-dire la mise en page et l'apparence de l'application (titre, boutons, sélecteurs,...).

La structure des deux fichiers `server.R` et `ui.R` est différente. Dans `ui.R`, on trouve notamment :

- des éléments qui structurent la page Web (`fluidPage()`, `sidebarLayout()`, `mainPanel()`),
- des éléments graphiques statiques (`titlePanel()`),
- des éléments de saisie (`sliderInput()`),
- des éléments de rendu dynamique (`plotOutput()`).

Dans `server.R`, on trouve la lecture des données et les méthodes de construction d'éléments dynamiques qui sont ensuite rendus par l'interface.

Ce qu'il faut retenir, c'est que les entrées sont lues depuis la partie `ui.R` puis envoyées à la partie `server.R` pour être traitées. Le résultat de ce traitement est ensuite restitué dans la partie `ui.R`. Tout ceci se produit de manière dynamique.

On lance directement l'application en cliquant sur le bouton `RunApp` lorsqu'un des deux fichiers est ouvert. On peut aussi exécuter la commande `runApp("~/mondossier/mon_appli")`.

8.2 Ecrire le fichier `server.R`

Le fichier `server.R` contient une fonction principale `shinyServer()`, fonction d'`inputs` et d'`outputs`. A l'intérieur de cette fonction, on définit les différentes composantes de l'application `shiny` à l'aide des fonctions réactives dynamiques suivantes :

- `renderText()` pour restituer une chaîne de caractères,
- `renderPlot()` pour restituer des graphiques,
- `renderTable()` pour restituer un tableau,...

Les `outputs` sont enregistrés sous la forme `output$<nom_de_la_composante>` et seront utilisés plus tard dans la partie `ui.R`. Ils peuvent dépendre des `inputs`, définis dans la partie `ui.R`. Ils sont alors automatiquement mis à jour.

Voici un exemple de code `server.R` :

```
shinyServer(function(input,ouput)){

  # créer un graphique
  ouput$mon_nuage <- renderPlot({ # sauvegarde l'output graphique sous le nom mon_nuage
    plot(mtcars$mpg,mtcars$wt) # code R
  })

  # créer du texte
  output$mon_texte <- renderText({ # sauvegarde d'output texte sous le nom mon_texte
    titre <- input$titre # utilise l'input défini dans la partie ui.R
    cat(paste0("Voici l'input : ",titre)) # colle l'input à un texte
  })

}
```

Si l'on souhaite créer des objets à utiliser dans de multiples `outputs`, on peut utiliser la fonction `reactive` :

```
x <- reactive({ # sauvegarde l'input sous l'objet x
  input$titre
})

output$y <- renderText({
  x() # utilisation de x une première fois
})

output$z <- renderText({
  x() # utilisation de x une seconde fois
})
```

8.3 Ecrire le fichier `ui.R`

Pour fonctionner, le fichier `ui.R` doit contenir au minimum les instructions `shinyUI(fluidPage())`. Dès lors, on peut définir la mise en page, le type d'`inputs` souhaité ainsi que les `outputs` à afficher.

La mise en page de base d'une application `shiny` est très simple. Elle se compose d'un titre, spécifié par la fonction `titlePanel()`, d'une liste d'`inputs` et d'`outputs`.

Différents types d'`inputs` peuvent être définis, parmi lesquels :

- `numericInput()` pour saisir directement une valeur numérique,

- `sliderInput()` pour définir une valeur numérique à l'aide d'un curseur,
- `checkboxInput()` pour choisir (en cochant) parmi une sélection de variables,
- `selectInput()` pour choisir un unique élément parmi une sélection de variables. Si la liste de choix possible est courte, on peut aussi utiliser la fonction `radioButtons()`, qui affiche l'ensemble des choix possibles,
- `textInput()` pour saisir une chaîne de caractères simples,
- `dateInput()` pour sélectionner une date.

Les `outputs` définis dans la partie `server.R` et que l'on souhaite afficher doivent être récapitulés dans la partie `ui.R`. Pour cela, on fait correspondre à chaque objet, sa fonction `<.>Output` associée. Pour afficher un graphique, on utilisera par exemple `plotOuput()` et pour un tableau `tableOutput()`.

Voici un exemple de code `ui.R`, qui peut être utilisé en complément du code `server.R` décrit précédemment :

```
shinyUI(fluidPage( # mise en page de base

  titlePanel("Première application"), # définit le titre de l'application

  textInput("titre", # définit l'input$titre
            "Titre du graphique", # nom à afficher pour l'input$titre
            value="Ceci est mon titre par défaut"), # valeur par défaut

  plotOutput("mon_nuage"), # pour afficher le graphique output$mon_nuage
  textOutput("mon_texte") # pour afficher le texte output$mon_texte

))
```

8.4 Mise en page sous *shiny*

La mise en page d'une application `shiny` est définie dans la partie `ui.R`. Par défaut, les `inputs` sont listés les uns sous les autres et sont suivis des `outputs`. Il existe différentes fonctions pour changer cette mise en page. La fonction `sidebarLayout()`, classiquement utilisée, permet ainsi de prévoir une mise en page en deux zones :

- une zone `sidebarPanel()` servant à définir les `inputs`,
- une zone principale (ratio 1/3 - 2/3) `mainPanel()` permettant d'afficher les `outputs`.

Le code s'écrit alors :

```
shinyUI(fluidPage(
  titlePanel("Mon titre ici")

  sidebarLayout(
    sidebarPanel( # définition des inputs
      selectInput(...)
    ),

    mainPanel( # définition des outputs
      plotOutput(...)
    )
  )
))
```

Des mises en page plus complexes existent et permettent de personnaliser l'aspect de son application `shiny` :

- `splitLayout()` et `flowLayout()` divisent la page en régions de même taille,
- `fluidRow()`, utilisée en complément de `column()` permet de moduler la page en régions de taille à définir,

```
shinyUI(fluidPage(
  titlePanel("Mon titre ici")

  fluidRow( # une première ligne ici
     column(width=6,...) # premier bloc
     column(width=3,...) # deuxième bloc
  ),

  fluidRow( # une deuxième ligne ici
    column(width=9,...) # un bloc de taille 9
  )
))
```

- `navbarPage()`, utilisée en complément de `tabPanel()` permet de définir des onglets.

```
shinyUI(fluidPage(
  navbarPage("Titre de la barre de navigation", # définition d'une barre de navigation
    tabPanel("Un onglet", # premier onglet
      # ici mise en page de l'onglet
    ),

    tabPanel("Un deuxième onglet",
      # mise en page de l'onglet
    )
  )
))
```

Le graphique suivant présente de manière simplifiée ces différentes mises en page.