Chapitre 3 - Programmation

Thibault LAURENT

30 Octobre 2018

Contents

Ce document a été généré directement depuis RStudio en utilisant l'outil Markdown. La version .pdf se trouve ici.

Résumé du chapitre

Après avoir présenté le principe des fonctions, nous nous arrêterons un moment sur les structures de contrôle (**if/else**, **for**, **while**, **repeat**). Les structures de contrôle ne sont pas propres aux fonctions (elles peuvent s'utiliser en-dehors des fonctions), mais elles font partie des bases de la programmation. Nous verrons également plusieurs outils spécifiques au langage **R**, qui seront très utiles pour programmer.

1 Les fonctions

A l'instar des macros **SAS**, les fonctions **R** contiennent des instructions qui sont exécutées les unes à la suite des autres à partir d'arguments fournis par l'utilisateur. Une fois les instructions exécutées, elles renvoient généralement un résultat qui peut prendre la forme de graphiques ou de calculs.

1.1 Les fonctions prédéfinies

Jusqu'à présent, nous avons utilisé des fonctions prédéfines dans **R**. Certaines de ces fonctions permettent de créer des objets (*c()*, *rep()*, *seq()*, *factor()*, etc.), de faire des calculs statistiques (*sum()*, *mean()*, *var()*, *quantile()*, etc.), d'importer des données (*read.table()*, *read.csv2()*, etc.). Le document "Exercice - pour

apprendre du vocabulaire en **R**" était un moyen de présenter un certain nombre de fonctions existantes. Nous verrons tout au long du cours encore beaucoup d'autres fonctions, prédéfinies ou bien incluses dans des packages **R**, qui nous permettront d'utiliser des outils statistiques sophistiqués. La classe de ces objets particuliers est **function**. Par exemple :

```
class(mean)
```

```
## [1] "function"
```

La spécificité de ces objets est leur structure :

```
str(var)
```

```
## function (x, y = NULL, na.rm = FALSE, use)
```

Le résultat ci-dessus montre que la fonction *var()* a un argument obligatoire **x** (on voit que c'est obligatoire car il n'est pas suivi d'une valeur par défaut) et des arguments optionnels **y**, **na.rm**, **use** qui prennent des valeurs par défaut. Pour visualiser le code de la fonction, il suffit d'appliquer la fonction *print()* sur une fonction :

```
print(var)
```

ou simplement :

```
var
```

```
## function (x, y = NULL, na.rm = FALSE, use)
## {
##     if (missing(use))
##         use <- if (na.rm)
##             "na.or.complete"
##         else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##         "everything", "na.or.complete"))
##     if (is.na(na.method))
##         stop("invalid 'use' argument")
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
##     .Call(C_cov, x, y, na.method, FALSE)
## }
## <bytecode: 0x14eb768>
## <environment: namespace:stats>
```

Ceci nous permet donc de voir que la fonction *var()* contient un certain nombre de mots clés (**if**, **else**, etc), fait appel à d'autres fonctions (*missing()*, *stopifnot()*, *pmatch()*, etc.), créé de nouveaux objets (**use**, **x**, etc.). Pour comprendre tout ce qui est écrit à l'intérieur d'une fonction, il est donc nécessaire d'avoir acquis un vocabulaire **R** dont nous continuerons à vous présenter les bases dans ce chapitre.

Remarque : dans le cours de "**R** avancé", nous verrons que le code de toutes les fonctions n'est pas nécessairement accessible du premier coup comme c'est le cas ici avec la fonction *var()*.

1.2 Programmer ses propres fonctions

Maintenant, on va essayer de créer notre propre fonction. Tout comme les objets qu'on créé au cours d'une session, les fonctions que l'on créé ne sont pas sauvegardées à la fin d'une session. Il sera donc nécessaire, lors

d'une session ultérieure, de les soumettre préalablement au logiciel. Pour cela, il existe plusieurs façons de procéder :

- on peut enregistrer ses fonctions dans un fichier portant l'extension *.R* et soumettre ce fichier par la commande *source()* avant de l'utiliser.
- Une autre façon de faire est de sauvegarder ces fonctions avec la fonction *save()*, comme on le fait avec les objets, puis de charger le fichier *.RData* créé avec la fonction *load()*.
- Enfin, les utilisateurs aguerris pourront créer leur propre package **R** et charger leur package au début de chaque session afin d'utiliser toutes les fonctions incluses dans le package.

1.2.1 Syntaxe

Pour créer une fonction, on utilise la syntaxe générale suivante :

```
ma_fonction <- function(arg1, arg2, arg3) {
# étape de vérification
  stopifnot(is.numeric(arg1), is.numeric(arg2), is.numeric(arg3))
# début instructions
  a <- arg1 + arg2
  res <- a^2 + 2*arg3
# fin instructions
# Et si on veut retourner un objet en sortie, on utilise return()
  return(res)
}
```

Les éléments importants dans une fonction sont :

- **ma_fonction** : nom de la fonction.
- **arg1, arg2, arg3** : les arguments d'entrée. Il peut y en avoir autant que l'on souhaite et il peut s'agir d'objets de tous types (vecteurs, matrices, jeux de données, etc). Contairement à d'autres langages, il n'est pas nécessaire de préciser leur types.
- **vérification** : il s'agit d'une étape facultative, mais qui permet de vérifier que l'utilisateur a correctement rempli les arguments d'entrée. Par exemple, est-ce que les arguments d'entrée sont du bon type, si un argument d'entrée est un vecteur a-t-il la bonne dimension, etc. Ici, la fonction *stopifnot()* arrête la fonction si au moins une des expressions qui lui sont données en argument d'entrée n'est pas vérifiée. Nous la verrons plus en détails dans le paragraphe suivant.
- **instructions** : les instructions peuvent prendre plusieurs formes. Ici, nous faisons uniquement de la création de nouvels objets, utilisations de boucles, de fonctions de base, etc. Nous nous restreindrons à des instructions propres au langage **R**, mais dans de nombreuses fonctions vous trouverez des instructions de type *.Internal()*, *.Call()* ou *UseMethod()* qui correspondent à des appels de fonctions écrites dans un autre langage (**Fortran**, **C**, etc.), dans le but d'améliorer les temps de performances.
- **return** : une fonction ne peut "renvoyer" qu'un seul objet. C'est pourquoi lorsqu'on a plusieurs informations à retourner, on utilisea des objets de type **list**. Une autre solution pour les utilisateurs avertis, sera de définir et créer sa propre classe d'objets, mais ce n'est pas quelque chose que nous aborderons dans ce cours. La fonction *return()* n'est pas obligatoire, mais elle permet de marquer clairement la fin d'une fonction car en plus de retourner l'objet mis entre parenthèses, elle arrête l'évaluation de la fonction.

Remarque : pour délimiter le début et la fin d'une fonction, une accolade ouvrante est placée juste après la définition des paramètres d'entrée et une accolade fermante est placée à la fin de la fonction.

1.2.2 Notre 1ère fonction

On souhaite créer une fonction appelée **rate** qui calcule le taux d'accroissement moyen annuel d'un produit dont le prix est passé de p1 à p2 entre la date t1 et la date t2. Pour cela, la fonction aura :

- en arguments d'entrée : le prix **p1** à la date **t1**, le prix **p2** à la date **t2**.
- en argument de sortie : le taux d'accroissement **r** du produit et la durée de la période d'observation **d** (qui vaut **t2-t1**).

Pour faire cela, on propose le code suivant :

```
rate <- function(p1, t1, p2, t2) {
# étape de vérification : les arguments doivents tous êtres de type numeric
  stopifnot(is.numeric(p1), is.numeric(t1), is.numeric(p2), is.numeric(t2))
# instructions
  d <- t2 - t1 # on calcule la duree entre t1 et t2
  r <- (p2/p1)^(1/d) - 1 # on calcul le taux d'accroissement moyen
  res <- list(r = r, d = d) # résultat sous forme de list
}
```

Remarque : dans cet exemple, on n'a pas utilisé la fonction *return()*. Dans ce cas, **R** retourne automatiquement la dernière expression qui a été évaluée, ici **res**.

Application :

- Quel est le taux annuel d'accroissement moyen d'un article dont le prix est passé de 100 € à 500 € entre 2000 et 2015 ?

Pour être sûr de ne pas se tromper dans l'ordre des arguments d'entrée, le mieux est de préciser le nom des arguments qu'on appelle à l'intérieur de la fonction :

```
res1 <- rate(p1 = 100, t1 = 2000, p2 = 500, t2 = 2015)
res1
```

```
## $r
## [1] 0.1132636
##
## $d
## [1] 15
```

Dans ce premier exemple, le taux d'accroissement annuel moyen est égal à 11.3% sur une période de 15 ans.

Il est tout à fait possible de ne pas préciser le nom des arguments d'entrée de la fonction, mais il faut veiller à bien respecter l'ordre d'apparition des arguments d'entrée :

```
(res1 <- rate(100, 2000, 500, 2015))
```

```
## $r
## [1] 0.1132636
##
## $d
## [1] 15
```

Remarque : pour afficher la valeur d'un objet sur lequel on vient de faire une affectation, on encadre l'expression par des parenthèses. Ainsi le code précédent est équivalent à :

```
res1 <- rate(p1 = 100, t1 = 2000, p2 = 500, t2 = 2015)
print(res1)
```

```
## $r
## [1] 0.1132636
```

```
##
## $d
## [1] 15
```

- Quel est le taux annuel d'accroissement moyen d'un article dont le prix est passé de 200 € à 100 € entre 2000 et la fin du 1er semestre de 2012 ?

```
# application 2
(res2 <- rate(200, 2000, 100, 2012.5))
```

```
## $r
## [1] -0.05394235
##
## $d
## [1] 12.5
```

Ici, le taux d'accroissement annuel moyen est de -5.4% sur une période de 12 ans et demi.

Le résultat est donné sous forme de **list** car à l'intérieur de la fonction, l'objet **res** est un objet de classe **list**. Celui-ci est composé de deux éléments de types **numeric** :

```
class(res1)
```

```
## [1] "list"
```

```
str(res1)
```

```
## List of 2
##  $ r: num 0.113
##  $ d: num 15
```

1.2.3 La fonction *stopifnot()*

Lorsqu'on fait des vérifications sur les paramètres d'entrée d'une fonction, la fonction *stopifnot()* permet de vérifier plusieurs conditions à la fois. Elle s'utilise ainsi :

```
stopifnot(<condition 1>, <condition 2>, ...)
```

Le genre de vérifications qui peuvent être faits :

- le type des arguments d'entrée,
- la dimension des arguments d'entrée,
- présence de valeurs manquantes.

Dans d'autres langages de programmation, l'étape de vérification se fait avec l'expression **if/else**. En **R**, la fonction *stopifnot()* est équivalent à l'expression suivante.

```
if(!<condition 1> | !<condition 2> | ...) stop()
```

1.2.4 Erreur dans les arguments d'entrée

Que se passe-t-il si on applique cette fonction et qu'on se trompe dans les arguments d'entrée ?

- Cette erreur peut se manifester par un message d'erreur, si par exemple les types des arguments d'entrée ne sont pas les mêmes que ceux attendus. Par exemple :

```
rate("100", 2000, 500, 2015)
```

```
## Error in rate("100", 2000, 500, 2015): is.numeric(p1) is not TRUE
```

- Une source d'erreur fréquente est dûe à une permutation dans l'ordre des arguments d'entrée. S'il n'y a pas de conflits avec le type des arguments d'entrée, il n'y aura pas de message d'erreurs. En effet, dans l'exemple suivant (où on a permuté les valeurs de **t1** et **p2**), on ne peut se rendre compte de l'erreur que parce que le résultat n'est pas cohérent (période de 1515 ans !) :

```
(res1 <- rate(100, 500, 2000, 2015))
```

```
## $r
## [1] 0.001979337
##
## $d
## [1] 1515
```

1.2.5 Arguments d'entrée par défaut

Lors de la construction d'une fonction, l'utilisateur peut donner à certains arguments des valeurs par défaut en leur attribuant d'office une valeur. Dans notre exemple, on attribue à **t1** la valeur 2010 et **t2** la valeur 2018. Dans ce cas, on permute les arguments d'entrée de telle sorte que les arguments "facultatifs" soient placés après les autres arguments. Par exemple :

```
rate <- function(p1, p2, t1 = 2010, t2 = 2018) {
  d <- t2 - t1
  r <- (p2/p1)^(1/d)-1
  res <- list(r = r, d = d)
  res
}
```

Application :

- Quel est le taux annuel d'accroissement moyen d'un article dont le prix est passé de 100 € à 200 € entre 2010 et 2018 ?

```
# exemple 3
rate(100, 200)
```

```
## $r
## [1] 0.09050773
##
## $d
## [1] 8
```

Dans ce cas, on n'est pas obligé de renseigner les valeurs de **t1** et **t2** puisqu'elles coïncident avec les valeurs qu'on souhaitent leur affecter.

Questions :

Q1 Créer une fonction *type_variable_df()* qui prend comme argument d'entrée un objet **data.frame** et renvoie pour chaque colonne son type (qu'on obtiendra avec la fonction *class()*). On pourra utiliser la fonction *lapply()* qui s'utilise aussi bien sur les **list** que les **data.frame** afin de faire le calcul par colonne.

Q2 Créer une fonction *puissance()* qui prenne comme argument d'entrée un vecteur **x** de type **numeric** et l'argument **type** qui prend comme valeur "carree" (par défaut) ou "cube". Cette fonction renverra la valeur de **x** prise au carrée ou au cube selon la valeur de **type**.

Remarque : cette fonction nécessite d'avoir certaines bases avec la condition **if/else** que nous verrons dans la section suivante. Le lecteur pourra donc résoudre cet exercice à sa deuxième lecture.

Réponse :

Q1 Une solution est :

```r
type_variable_df <- function (df){
  # verification
  stopifnot(is.data.frame(df))
  # resultat
  lapply(df, class)
}
```

Application : il existe beaucoup de jeux de données (tous de type **data.frame**) disponibles sous **R**. Pour les connaître, il suffit de taper dans la console :

```r
data()
```

On applique notre fonction au **data.frame** intitulé **sleep** :

```r
type_variable_df(sleep)
```

```
## $extra
## [1] "numeric"
##
## $group
## [1] "factor"
##
## $ID
## [1] "factor"
```

Q2 Une solution est :

```r
puissance <- function(x, type) {
  # verification
  stopifnot(is.numeric(x), type %in% c("carree", "cube"))
  # conditions
  if (type == "carree") {
    x <- x^2
  } else {
    x <- x^3
  }
  return(x)
}
```

Exemple d'application :

```r
puissance(1:10, "carree")
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```r
puissance(1:10, "cube")
```

```
##  [1]    1    8   27   64  125  216  343  512  729 1000
```

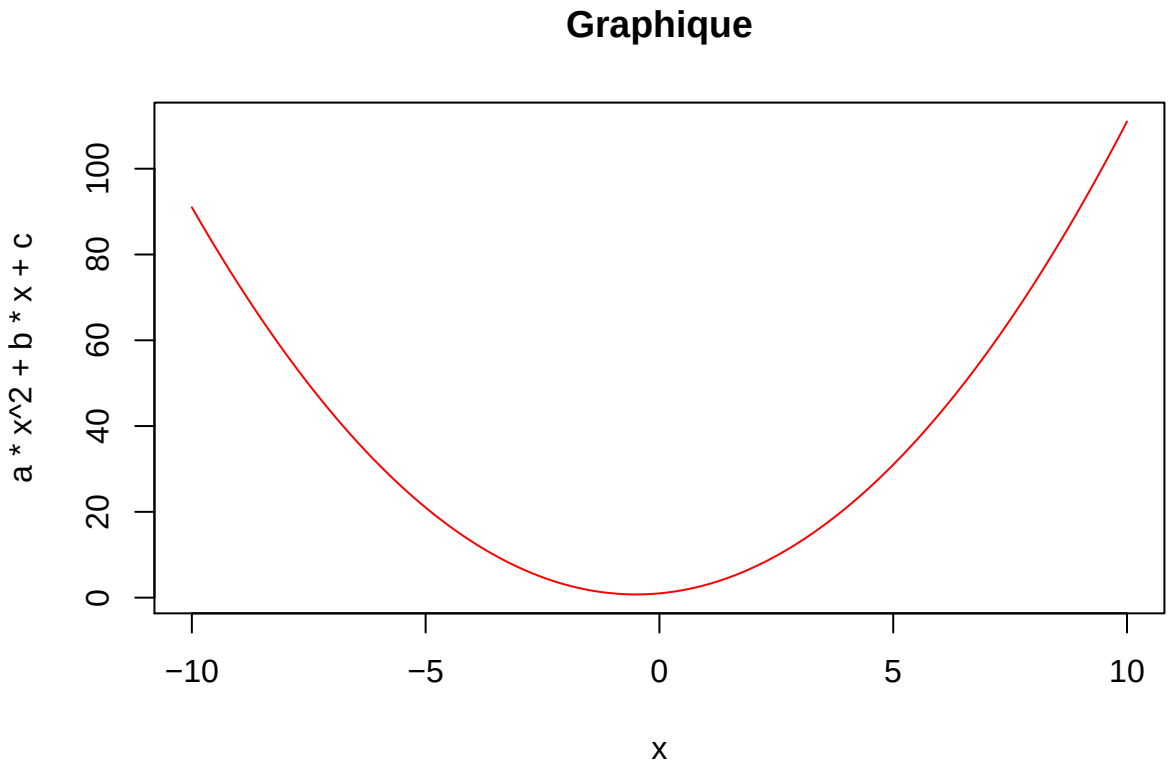
1.2.6 Quand mettre ... comme argument d'entrée d'une fonction ?

On souhaite créer une fonction qui va appeler une fonction prédéfinie qui a une multitude d'arguments d'entrée (c'est le cas notamment des fonctions graphiques que nous verrons dans le chapitre suivant). Dans ce cas, plutôt que de reécrire tous les arguments d'entrée de la fonction qu'on va utiliser, il suffit d'ajouter parmi les arguments d'entrée de notre fonction, le symbole **...** qu'on re-utilisera au moment de l'appel de la fonction prédéfinie. Par exemple, on souhaite donner comme arguments d'entrée les paramètres **a**, **b** et **c** du polynôme $ax^2 + bx + c$ et représenter cette fonction dans un intervalle borné (argument **borne** égal par défaut à $[-10, 10]$) :

```r
myFunc <- function(a, b, c, borne = c(-10, 10), ...) {
  x <- seq(borne[1], borne[2], 0.1)
  plot(x, a*x^2 + b*x + c, ...)
}
```

Dans l'écriture ci-dessus, on retrouve les **...** deux fois : une fois comme argument d'entrée de la fonction qu'on est en train d'écrire et une seconde fois parmi les arguments d'entrée de la fonction *plot()*. Cela signifie que lorsqu'on appelera la fonction *myFunc()*, tous les arguments d'entrée qui sont différents de **a**, **b**, **c** et **borne**, seront automatiquement utilisés dans la fonction *plot()*. Voici un exemple d'utilisation :

```r
myFunc(1, 1, 1, main = "Graphique", type = "l", col = "red")
```



Ici, les arguments **main=**, **type=** et **col=** sont des arguments reconnus par la fonction *plot()*. C'est pourquoi nous n'avons pas eu à les définir comme paramètre d'entrée de la fonction *myFunc()* parce qu'ils sont inclus dans les **...**. Pour connaître tous les arguments d'entrée disponibles, il suffit de faire *help(plot.default)*.

1.3 Variable globale VS Variable locale

Dans la fonction **rate** créée précedemment, les objets **p1**, **p2**, **t1**, **t2**, **d**, **r** et **res** sont toutes des variables locales car elles ont été définies à l'intérieur de la fonction **rate** et ne peuvent être qu'utilisées à l'intérieure de celle-ci. Par exemple, la commande suivante renverra un message d'erreur car dans l'environnement global de **R**, l'objet **d** n'est pas connu.

```r
print(d)
```

```
## Error in print(d): objet 'd' introuvable
```

Maintenant, considérons la fonction suivante (**remarque:** lorsque le code d'une fonction est contenue dans 1 seule ligne, on n'est pas obligé de mettre une accolade ouvrante et fermante):

```
f <- function(x)
  x + a^2
```

On constate que l'objet **a** n'a pas été défini à l'intérieur de la fonction. **R** va donc aller chercher si cet objet existe à l'extérieur. Si cet objet existe en variable globale, il sera utilisé; en revanche, s'il n'existe pas, il y aura un message d'erreur comme le montre cet exemple :

```
f(5)
```

```
## Error in f(5): objet 'a' introuvable
```

A présent, on définit **a** en variable globale :

```
a <- 3
```

On re-applique la fonction *f()* :

```
f(5)
```

```
## [1] 14
```

Si l'objet existe à la fois en variable locale et en variable globale, c'est la variable locale qui sera utilisée en priorité à l'intérieur de la fonction. On reprogramme la fonction et on definit l'objet **a** à l'intérieur :

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

Lorsqu'on applique cette fonction, on voit bien que la valeur de **a** qui a été utilisée est celle qui a été définie à l'intérieur de la fonction :

```
f(5)
```

```
## [1] 9
```

1.4 Comprendre un message d'avertissement ou d'erreur

La plupart du temps, si une fonction retourne un message d'avertissement ou un message d'erreur, ces messages sont suffisament explicites pour qu'on arrive à debugger le problème soi-même. Dans l'exemple suivant, le message est très clair :

```
a <- c("red", "blue")
mean(a)
```

```
## Warning in mean.default(a): argument is not numeric or logical: returning
## NA
```

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

Dans cet exemple, on a reçu un message d'avertissement (Warning message), ce qui est différent d'un message d'erreur. En effet, un message d'avertissement retourne une valeur alors qu'un message d'erreur signifie que l'appel d'une fonction a été interrompue.

Lorsqu'on ne comprend pas pourquoi une fonction a retourné un message d'erreur, on peut utiliser la fonction *traceback()* suivante.

1.4.1 Utiliser la fonction *traceback()* pour debugger une fonction

L'exemple que nous allons prendre est tiré de l'aide de la fonction *traceback()*. On créé ici deux fonctions dont la première fait appel à la seconde :

```
# Première fonction
foo <- function(x) {
  print(1)
  bar(2)
}

# Deuxième fonction
bar <- function(x) {
  x + variable_manquante
}
```

Si on fait :

```
foo(2)
```

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

```
## Error in bar(2): objet 'variable_manquante' introuvable
```

on obtient un message d'erreur. Ce message est très clair et comme les fonctions *foo()* et *bar()* ne comptent que quelques lignes, on peut facilement comprendre que l'objet **variable_manquante** n'a pas été trouvé ni en variable locale, ni en variable globale. Mais dans le cas où les fonctions contiennent un grand nombre de lignes, cela deviendrait difficile de repérer d'où vient le problème.

La fonction *traceback()* permet de retracer l'historique des différentes fonctions appelées qui ont conduit au dernier message d'erreur.

```
traceback()
```

Ici, on a d'abord appellé la fonction *foo()* (étape 1), puis à l'intérieur de celle-ci, on a appelé la fonction *bar()* qui a conduit au bug au niveau de la deuxième ligne.

2 Les structures de contrôle

Quelque soit le langage de programmation, on retrouvera les structures de contrôle **if/else**, **for** et **while**. On en profitera pour voir également des structures de contrôle propre au langage **R**. L'idée de cette section est de donner du vocabulaire en programmation, qui pourra être utilisé soit en-dehors, soit à l'intérieur de nos fonctions. Les exemples présentés ici peuvent être directement exécuter dans la console **R**.

2.1 Les opérateurs de comparaison

Avant de présenter la condition **if/else** dont l'objectif est de faire un choix selon un critère de comparaison, il nous semble important de voir de nouveaux les opérateurs et fonctions qui permettent de faire plusieurs comparaisons à la fois.

2.1.1 Vérifier plusieurs conditions à la fois avec les opérateurs & et |

Ces deux opérateurs permettent de vérifier deux conditions à la fois. Pour les illustrer, on va considèrer les deux scalaires **a** et **b** suivants :

```r
a <- 5
b <- -5
```

2.1.1.1 L'opérateur &

L'opérateur **&** correspond à l'opérateur logique "et". Les règles lorsqu'on teste deux conditions avec **&** sont les suivantes :

- si les deux conditions sont vérifiées (autrement dit, on a **TRUE** et **TRUE**), on obtient la valeur **TRUE**. Par exemple :

```r
(abs(a) > 4) & (abs(b) > 4)
```

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

- en revanche, si au moins une des 2 conditions n'est pas vérifiée (autrement dit, on a soit **FALSE** et **TRUE**, **TRUE** et **FALSE** ou bien **FALSE** et **FALSE**), on obtient la valeur **FALSE**. Par exemple :

```r
(a > 4) & (b > 4) # la 2nde condition n'est pas vérifiée
```

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

```r
(a < 4) & (b < 4) # la 1e condition n'est pas vérifiée
```

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

```r
(a == 4) & (b ==4 ) # aucune condition n'est vérifiée
```

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

2.1.1.2 L'opérateur |

L'opérateur | correspond à l'opérateur logique "ou".

- si au moins une des 2 conditions est vérifiée, on obtient la valeur **TRUE** :

```r
(abs(a) > 4) | (abs(b) > 4) # les deux conditions sont vérifiées
```

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

```r
(a > 4) | (b > 4) # au moins une des deux conditions (la 1e) est vérifiée
```

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

```r
(a < 4) | (b < 4) # au moins une des deux conditions (la 2nde) est vérifiée
```

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

- si aucune des 2 conditions n'est vérifiée, on obtient la valeur **FALSE** :

```r
(a == 4) | (b == 4) # aucune condition n'est vérifiée
```

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

Remarque : on peut également utiliser ces opérateurs sur des vecteurs. Dans ce cas-là, on vérifie que le i-ème élément de la condition de gauche et/ou le i-ème élément de la condition de droite sont vérifiées et on le fait pour tous les éléments du vecteur comparé. Par exemple :

```r
a <- -2:2
a >= 0
```

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

```
a <= 0
```

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

```
(a >= 0) & (a <= 0)
```

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

```
(a >= 0) | (a <= 0)
```

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

2.1.1.3 Vérifier plusieurs conditions

Lorsqu'on teste successivement des conditions à la suite, on effectue des tests 2 par 2, en commençant de la gauche vers la droite. Par exemple :

```
FALSE & TRUE | TRUE
```

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

est équivalent à :

```
(FALSE & TRUE) | TRUE
```

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

mais qui donne un résultat différent de :

```
FALSE & (TRUE | TRUE)
```

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

Conseil : lorsqu'on teste plus de deux conditions, il est très important d'utiliser les parenthèses en fonction des priorités qu'on donne aux conditions.

2.1.1.4 Les opérateurs de court-circuitage && et ||

On considère le scalaire suivant :

```
a <- -6
```

On souhaite vérifier que cet objet est :

- de type **numeric**,
- une valeur positive,
- une valeur paire. Pour ce faire, on va utiliser l'opérateur **a%%2** (modulo) qui consiste à regarder si le reste de la division de **a** par 2 vaut 0 ou non.

Pour ce faire, on utilise la commande :

```
is.numeric(a) & a > 0 & a%%2 == 0
```

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

Pour comprendre quelles ont été les expressions évaluées par **R**, on ajoute la fonction *print()* devant chaque comparaison :

```
print(is.numeric(a)) & print(a > 0) & print(a%%2 == 0)
```

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

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

On voit donc que **R** a évalué les 3 comparaisons les unes après les autres avant d'afficher le résultat final.

Dans cet exemple, on voit qu'à partir de la deuxième comparaison, le résultat sera **FALSE** quelque soit le résultat de la 3ème comparaison. Pourtant, la 3ème comparaison est quand même évaluée. Si on souhaite arrêter le test dès que le test est **FALSE**, on peut utiliser les opérateurs **&&** et ||. Si on reprend l'exemple précédent, en remplaçant **&** par **&&**, le test sera arrêté à la deuxième comparaison, ce qui permet d'éviter de faire le 3ème calcul dont le résultat ne changera pas l'issue du test :

```
print(is.numeric(a)) && print(a > 0) && print(a%%2 == 0)
```

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

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

Si au premier abord cette syntaxe semble être utile dans le but d'éviter des calculs inutiles, elle peut s'avérer très dangereuse lorsqu'on fait des comparaisons sur des vecteurs. Dans l'exmple suivant, on constate que seul le premier élément des vecteurs à comparer a été utilisé :

```
a <- -2:2
a == -2 && a <=0
```

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

Si on n'avait l'opérateur **&**, on aurait obtenu :

```
a == -2 & a <=0
```

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

Si on utilise les opérateurs de court-circuitage, il est donc essentiel de s'être assurer que la comparaison porte sur des scalaires.

2.1.2 La commande %in%

Cette commande s'utilise de la façon suivante : x %in% table

Elle renvoit un vecteur de booléen de la même taille que **x** qui indique si oui (**TRUE**) ou non (**FALSE**) les éléments de x se trouvent dans **table**. Par exemple, on souhaite vérifier que le vecteur **x** ne contient que les modalités **male** ou **female** :

```
x <- c("male", "male", "female", "autre", "male")
x %in% c("male", "female")
```

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

On notera dans l'aide de la fonction *match()* que la commande %in% est définie à partir de la fonction *match()*. La fonction *match(x, table)* permet de savoir si les éléments du vecteur **x** sont bien présents dans **table** et si oui, à quelle place. Par exemple :

```
match(x, c("male", "female"))
```

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

Si une valeur de **x** n'a pas été trouvée dans **table**, on obtient la valeur **NA**. Pour détecter des valeurs manquantes dans un vecteur, on utilise la fonction *is.na()*. Autrement dit, pour retrouver le même résultat qu'avec l'opérateur *%in%*, on fait :

```
!is.na(match(x, c("male", "female")))
```

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

2.1.3 Utiliser les fonctions *all()* et *any()*

Lorsqu'on souhaite vérifier que plusieurs conditions sont toutes égales à **TRUE**, on utilise la fonction *all()*. Par exemple, on souhaite vérifier que tous les éléments du vecteur **x** sont égales à **male** ou **female** :

```
all(x %in% c("male", "female"))
```

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

Pour vérifier s'il existe au moins une valeur **TRUE** dans un vecteur de booléen, on utilise la fonction *any()*. Ici, on souhaite savoir s'il existe au moins une valeur égale à **autre** :

```
any(x %in% "autre")
```

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

2.1.4 La commande !

Parfois, il peut intéressant d'utiliser la négation d'un booléen à l'aide de l'opérateur **!**.

```
!(x %in% "autre")
```

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

```
!any(x %in% "autre")
```

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

Question

On considère le vecteur suivant :

```
x <- c(55, 40, 55, 60, 70, 18)
```

Q1 Tester si **x** contient au moins une valeur négative.

Q2 Tester que toutes les valeurs de **x** sont positives ou nulles.

Réponse

Q1

```
any(x < 0)
```

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

Q2

```
all(x >= 0)
```

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

Remarque: pour répondre à l'une des deux questions, on aurait aussi pu prendre la négation de l'autre.

2.2 L'expression if/else

Dans le paragraphe précédent, on a fait le tour sur les opérateurs de comparaison. L'expression **if** va permettre de vérifier qu'une condition est vérifiée et orienter selon le résultat, vers des instructions spécifiques.

2.2.1 Syntaxe

Pour lire plus facilement le code, il est conseillé de respecter quelques règles de styles lorsqu'on utilise des structures de contrôle (**if/else**, **for**, **while**) :

- laisser un espace après le mot clé,
- l'accolade ouvrante se trouve à la fin de la ligne contenant le mot clé, l'accolande fermante se trouve sur une nouvelle ligne,
- mettre deux espaces en début de ligne (indentation) à partir de la seconde ligne jusqu'à la fin de la condition,
- si des conditions sont imbriquées, la nouvelle condition devrait se trouver sur la même ligne que l'accolade fermante de la première condition.

La syntaxe pour la structure de contrôle **if/else** est :

```
if (<condition(s)>) {
  <instruction 1>
} else {
  <instruction 2>
}
```

Si la (les) condition(s) renvoie la valeur **TRUE**, l'instruction 1 sera exécutée. Dans le cas contraire, ce sera l'instruction 2 qui sera exécutée.

Remarque : le **else** n'est pas obligatoire, et dans ce cas aucune instruction ne sera exécutée si la condition est rejetée. On peut simplement avoir :

```
if (<condition(s)>) {
  <instruction 1>
}
```

2.2.2 Un premier exemple

On cherche à savoir si une valeur **x** simulée selon une loi normale $\mathcal{N}(0, 1)$, est supérieure à 1.96. D'abord, on simule la valeur **x** :

```
x <- rnorm(1)
```

On utilise ensuite la condition **if/else** et selon le résultat de la condition, on affiche un résultat plutôt qu'un autre :

```
if (x > 1.96) {
  print(paste(round(x, 2), "is an extreme value if N(0, 1)"))
} else {
  print(paste(round(x, 2), "is not an extreme value if N(0, 1)"))
}
```

```
## [1] "0.71 is not an extreme value if N(0, 1)"
```

2.2.3 Un second exemple

On cherche à savoir si dans un vecteur **x** de taille $n = 25$ simulé selon une loi normale $\mathcal{N}(0, 1)$, il existe au moins une valeur qui en valeur absolue soit supérieure à 1.96. On simule d'abord le vecteur gaussien :

```
x <- rnorm(25)
```

On vérifie ensuite la condition recherchée :

```r
if (any(abs(x) > 1.96)) {
  print("Il existe au moins une valeur extreme")
} else {
  print("Il n y a pas de valeurs extremes")
}
```

```
## [1] "Il existe au moins une valeur extreme"
```

Pourquoi avons-nous écrit *if(any(abs(x)>1.96))* et pas simplement *if(abs(x)>1.96)* ?

Si on fait juste :

```r
abs(x) > 1.96
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE
```

on obtient un vecteur de booléen de la même taille que **x** avec la valeur **TRUE** pour les valeurs de **x** qui sont supérieures à 1.96 en valeur absolue et **FALSE** sinon. Si on applique l'expression **if** à un vecteur de booléen, on obtient le message d'avertissement suivant :

```r
if (abs(x) > 1.96) {
  print("Il existe au moins une valeur extrême")
}
```

```
## Warning in if (abs(x) > 1.96) {: la condition a une longueur > 1 et seul le
## premier élément est utilisé
```

Autrement dit, la condition n'a porté que sur la première valeur du vecteur de booléen. C'est pourquoi nous avons utilisé la fonction *any()* qui prend comme argument d'entrée un vecteur de booléen et renvoit la valeur **TRUE** si la valeur **TRUE** apparaît au moins une fois dans le vecteur de booléen.

```r
any(abs(x) > 1.96)
```

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

2.2.4 Un troisième exemple

Dans cet exemple, nous allons voir comment utiliser l'expression **if/else** à l'intérieur d'une fonction. Par ailleurs, nous allons imbriquer plusieurs expressions **if/else** entre elles.

Objectif : créer la fonction *prix_apres_taxe* qui aura les deux arguments d'entrée suivants :

- **prix_HT**, un objet de type **numeric**,
- **type**, un **character** qui prend trois valeurs possibles dans "normal", "intermediaire", "reduit".

La fonction retournera le prix final après application des taux suivants sur le **prix_HT**, à savoir 20% sur le **type** "normal", 10% sur le **type** "intermediaire" et 5.5% sur le **type** "reduit". Voici le code qui permet de répondre à ce problème :

```r
prix_apres_taxe <- function(prix_HT, type) {
# verification
  stopifnot(is.numeric(prix_HT), type %in% c("normal", "intermediaire", "reduit"))
# calcul du prix selon le type
  if (type == "normal"){
    prix <- prix_HT*1.2
  } else {
    if(type == "intermediaire") {
```

```
      prix <- prix_HT*1.1
    } else {
      prix <- prix_HT*1.055
    }
  }
# le resultat à retourner
  return(prix)
}
```

Exemple d'application :

On vérifie que l'étape de vérification fonctionne bien :

```
prix_apres_taxe(1000, "TVA_reduit")
```

```
## Error in prix_apres_taxe(1000, "TVA_reduit"): type %in% c("normal", "intermediaire", "reduit") is no
```

On applique le même prix H.T. à la fonction pour les différents types possibles :

```
prix_apres_taxe(1000, "normal")
```

```
## [1] 1200
```

```
prix_apres_taxe(1000, "intermediaire")
```

```
## [1] 1100
```

```
prix_apres_taxe(1000, "reduit")
```

```
## [1] 1055
```

2.3 L'expression for

L'expression **for** permet l'exécution répétitive d'instructions un nombre fini de fois.

2.3.1 Syntaxe de for

```
for (<ind> in <vecteur>) {
  <instructions>
}
```

ind est la variable de boucle qui va prendre successivement toutes les valeurs de **vecteur**. **vecteur** prend généralement la forme *1:n*, mais il peut aussi être un vecteur de n'importe quel type. Les instructions seront répétées jusqu'à ce que **ind** aura pris successivement toutes les valeurs de **vecteur**. Le nombre de boucles est donc la taille de **vecteur**.

2.3.2 Exemples

1er exemple :

Dans ce premier exemple, on voit que **i** prend tour à tour les valeurs du vecteur $(1, 2, 3, 4, 5)$:

```
for (i in 1:5) {
  print(i)
}
```

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

On peut remplacer le vecteur $(1, 2, 3, 4, 5)$ par un vecteur de **character** :

```
for (j in c("debutant", "experimente", "professionnel")) {
  print(j)
}
```

```
## [1] "debutant"
## [1] "experimente"
## [1] "professionnel"
```

ou un vecteur de booléen :

```
for (k in c(T, T, T, F)) {
  print(k)
}
```

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

2ème exemple :

On souhaite répéter 5 fois l'instruction suivante :

- simuler 100 valeurs distribuées selon une loi gaussienne centrée et réduite
- compter à chaque fois le pourcentage de valeurs positives

```
for (k in 1:5) {
  x <- rnorm(100)
  print(length(which(x > 0))/length(x))
}
```

```
## [1] 0.54
## [1] 0.44
## [1] 0.51
## [1] 0.5
## [1] 0.55
```

Remarque : on rappelle que la fonction *which()* renvoie les indices des éléments d'un vecteur qui respectent une condition (ici la positivité). En appliquant la fonction *length()* au résultat de *which()*, on obtient donc bien le nombre d'éléments positifs.

3ème exemple :

Les boucles peuvent s'emboîter entre elles. Voici un petit exemple, où on définit une matrice de taille 3×2 qui contient des élements simulées selon une loi normale centrée réduite :

```
mat <- matrix(rnorm(6), 3, 2)
```

On va écrire un petit programme qui permet de dire si un élément de la matrice est positif ou négatif :

```
for (i in 1:nrow(mat)) {
  for (j in 1:ncol(mat)) {
    cat("ligne ", i, ", colonne ", j, " : ", mat[i, j] >= 0, "\n", sep = "")
```

```
  }
}
```

```
## ligne 1, colonne 1 : FALSE
## ligne 1, colonne 2 : FALSE
## ligne 2, colonne 1 : FALSE
## ligne 2, colonne 2 : TRUE
## ligne 3, colonne 1 : TRUE
## ligne 3, colonne 2 : TRUE
```

Remarque : on a utilisé la fonction *cat()* qui permet de concatèner et afficher des objets de différents types. Cela revient (pratiquement) à utiliser la fonction *paste()* (pour concaténer) puis *print()*.

2.3.3 Le mot clé break

Il est possible d'interrompre l'expression **for** en cours de route. Pour cela, le programme doit repérer le mot clé **break**. C'est pourquoi le mot-clé **break** est lié à l'expression **if** de telle sorte que **break** ne sera lu que lorsqu'une certaine condition aura été vérifiée. Ceci est intéressant pour éviter d'avoir à répéter toutes les instructions alors qu'on soupçonne qu'il y a un problème.

Pour illustrer ce concept, on considère le vecteur de **character** suivant :

```
x <- c("normal", "normal", "normal", "normal", "rrrrron", "rrrrron", "rrrrrron", "normal")
```

On souhaite effectuer l'opération suivante autant de fois que la taille du vecteur **x** :

- simuler 100 valeurs distribuées selon une loi gaussienne centrée et réduite et retourner la moyenne de ce vecteur,
- arrêter le processus si **x** ne prend pas la valeur "normal".

```
for (k in seq_along(x)) {
  if(x[k] != "normal") {
    cat("L'opérateur a du s'endormir. Nous allons interrompre le processus. \n")
    cat("Rapport d'erreur : le processus s'est interrompu à la boucle", k)
    break
  }
  print(mean(rnorm(100)))
}
```

```
## [1] -0.01355148
## [1] -0.08366064
## [1] -0.1307399
## [1] -0.009050641
## L'opérateur a du s'endormir. Nous allons interrompre le processus.
## Rapport d'erreur : le processus s'est interrompu à la boucle 5
```

Remarque 1 : la fonction *seq_along()* permet de définir la suite d'entier 1 jusqu'à n, où n est la taille du vecteur mis comme argument d'entrée.

2.3.4 Le mot clé next

Plutôt que d'interrompre une boucle, on peut sauter une répétition si une certaine condition n'a pas été vérifiée. C'est ce que permet de faire le mot clé **next** à l'intérieur d'une boucle. Si on reprend l'exemple précédent :

```r
for (k in seq_along(x)) {
  if(x[k] != "normal") {
    cat("Le processus s'est interrompu à la boucle", k, "\n")
    next
  }
  print(mean(rnorm(100)))
}
```

```
## [1] 0.07421056
## [1] 0.01387475
## [1] -0.06762082
## [1] 0.1379142
## Le processus s'est interrompu à la boucle 5
## Le processus s'est interrompu à la boucle 6
## Le processus s'est interrompu à la boucle 7
## [1] 0.08781641
```

2.4 L'expression while

L'expression **while** permet également l'exécution répétitive d'instructions. A la différence de **for**, on ne connaît pas a priori le nombre de fois où on va répéter les instructions, mais en revanche on va donner une condition d'arrêt. L'inconvénient est que si la condition d'arrêt n'arrive pas, le programme peut tourner en boucles sans jamais s'interrompre...

2.4.1 Syntaxe de while

```r
<initialisation paramètre(s)>
while (<Test sur paramètre(s)>) {
  <instructions>
  <Mise à jour paramètre(s)>
}
```

2.4.2 Exemple

Exemple 1 : tant que **i** ne dépasse pas la valeur 6 on ajoute la valeur 1 à **i**

```r
i <- 1  # initialisation
while (i < 6) { # condition d'arret
  print(i)
  i <- i + 1
}
```

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

Exemple 2 : on veut savoir au bout de combien de tirage d'un nombre simulé selon une gaussienne centrée et réduite, on dépasse la valeur 1.96.

```r
eps <- rnorm(1) # 1er tirage
k <- 1          # nombre de simulation nécessaire
while (eps < 1.96) {  # vérification
  eps <- rnorm(1) # nouveau tirage
  k <- k + 1      # mise à jour du nombre de simulation
}
cat("La valeur 1.96 a été dépassé après", k, "tirages")
```

```
## La valeur 1.96 a été dépassé après 19 tirages
```

2.4.3 Le mot clé break

Comme pour la boucle **for**, l'instruction **break** permet d'interrompre à tout moment un calcul lancé avec **while**, dès lors qu'un critère d'arrêt a été défini avec l'expression **if**.

Exemple : on souhaite savoir au bout de combien de temps et combien de simulation d'une loi normale centrée et réduite, la valeur simulée dépassera 4.26489. En théorie, la probabilité de dépasser 4.26489 vaut 0.00001. Autrement dit, on devrait tirer une valeur supérieure à 4.26489, 1 fois tous les 100000 tirages. Selon les machines, cela peut demander du temps avant de tirer une valeur supérieur à 4.26489. Par précaution, on souhaite arrêter le calcul dès lors que le temps de calcul dépassera 1 seconde. Pour répondre au problème, nous allons mettre en place le programme suivant.

```r
ptm <- proc.time()  # initialisation d'un temps de référence (voir explications ci-dessous)
eps <- rnorm(1)     # 1er tirage
temps_ecoule <- (proc.time() - ptm)[3] # mise à jour du temps de calcul
k <- 1   # paramètre indiquant le nombre de tirage nécessaire pour tirer la valeur seuil

# début boucle
while(eps < 4.26489) { # 1ère vérification : valeur seuil dépassée ?
 eps <- rnorm(1)       # nouveau tirage
 k <- k + 1            # mise à jour du nombre de simulation
 temps_ecoule <- (proc.time() - ptm)[3] # mise à jour du temps de calcul
 if(temps_ecoule > 1)  # 2ème vérification : temps calcul > 1 ?
   break
}
# fin boucle et affichage du résulat

if(temps_ecoule > 1) {
  cat("Le calcul a été interrompu (temps dépassé) après", k, "simulations")
} else {
  cat("La valeur 4.26489 a été atteinte au bout de :", k, "simulations en",
              temps_ecoule, "secondes.")
}
```

```
## La valeur 4.26489 a été atteinte au bout de : 111851 simulations en 0.827 secondes.
```

Explications :

- la fonction *proc.time()* renvoie un vecteur avec 3 valeurs numériques : la 3ème valeur (celle qui nous intéresse ici) affiche le temps en seconde depuis que la session **R** est ouverte. En exécutant à l'intérieur de chaque boucle cette fonction et en ayant pris un temps de référence au début du programme, cela nous permet de mettre à jour le temps de calcul qui s'est écoulé depuis le début du programme. A noter que les 1ère et 2ème valeurs renvoyées par la fonction *proc.time()* renvoient les temps de calcul réellement utilisés par notre machine depuis que la session **R** est ouverte.

2.4.4 Syntaxe alternative de while avec repeat

Dans la syntaxe avec **while**, d'abord on teste quelque chose et ensuite on agit. On peut vouloir d'abord agir et ensuite tester. Pour cela, on utilise la commande **repeat**. C'est le mot clé **break** que nous avons déjà vu précédemment qui va permettre d'interrompre la boucle.

```
<initialisation paramètre(s)>
 repeat {
   <Mise à jour paramètre(s)>
   <instructions>
   if (<Test sur paramètre(s)>)
     break
}
```

Exemple : on reprend l'exemple où on veut savoir au bout de combien de tirage d'un nombre simulé selon une gaussienne centrée et réduite, on dépasse la valeur 1.96.

```
k <- 0  # initialisation
repeat {
  eps <- rnorm(1) # mise à jour du paramètre testé
  k <- k + 1
  if (eps >= 1.96)
    break
}
cat("La valeur 1.96 a été dépassé après", k, "tirages")
```

```
## La valeur 1.96 a été dépassé après 96 tirages
```

La différence avec l'expression **while** est qu'il n'est pas nécessaire de faire une première simulation en-dehors de la boucle. De plus, la condition d'arrêt se trouve à la fin de l'expression.

2.5 La fonction ifelse()

2.5.1 Syntaxe

La syntaxe de la fonction **ifelse()** est la suivante :

```
ifelse(test, <resultat si oui>, <resultat si non>)
```

- Le premier argument d'entrée de cette fonction est un vecteur de booléen de taille n qui est en général le résultat d'une condition testée.
- Les deuxième et troisième arguments d'entrée peuvent être des scalaires ou des vecteurs qui seront mis automatiquement à la taille n (procédé qui ce fait comme nous l'avons vu dans le chapitre sur les vecteurs).
- La fonction **ifelse()** renvoit un vecteur de taille n dont les composantes seront celles de **yes** lorsque **test** vaut **TRUE** et celles de **no** lorsque **test** vaut **FALSE**.

Exemple 1 : pour créer une variable qualitative à partir d'une variable quantitative.

```
(x <- rnorm(10))
```

```
##  [1] -1.59133724 -0.19231284  1.30676910  0.02998976  0.39796621
##  [6] -0.13873421  0.75356886 -3.07169260  1.15171392  0.65761544
```

```
(signe <- ifelse(x > 0, "+", "-"))
```

```
##  [1] "-" "-" "+" "+" "+" "-" "+" "-" "+" "+"
```

2.5.2 Lignes de codes équivalentes

Dans l'exemple précédent, la fonction **ifelse()** est équivalente aux lignes de code suivantes :

2.5.2.1 Syntaxe 1

```
signe <- character(10)
signe[x > 0] <- "+"
signe[!(x > 0)] <- "-"
print(signe)
```

```
##  [1] "-" "-" "+" "+" "+" "-" "+" "-" "+" "+"
```

2.5.2.2 Syntaxe 2

```
signe <- character(10)
for (k in 1:10) {
  if (x[k] > 0)
    signe[k] <- "+"
  else
    signe[k] <- "-"
}
print(signe)
```

```
##  [1] "-" "-" "+" "+" "+" "-" "+" "-" "+" "+"
```

Quelle syntaxe utilisée dans ce genre de problème ? Il vaut mieux privilégier l'usage de la fonction *ifelse()* ou la syntaxe 1, plutôt que d'utiliser la syntaxe 2 qui sera plus coûteuse en temps de calcul.

2.5.3 Exercice

L'objectif est de créer une fonction **simul** qui prend comme arguments d'entrée un nombre **n** et un nombre **B**. A l'intérieur de la fonction, on répète **B** fois l'opération suivante :

- simulation d'un vecteur **x** de taille **n** selon une loi normale $\mathcal{N}(0, 1)$
- on vérifie si oui ou non il existe au moins un élément de **x** supérieur en valeur absolue à 1.96
- On comptabilise sur les **B** boucles le pourcentages de boucles où le phénomène s'est produit.

Une fois la fonction exécutée, on utilisera des boucles pour comparer les résultats de sortie en fonction de **n** (qui prendra les valeurs 10, 50, 100 et 200) et **B** (qui prendra les valeurs 10, 50 et 100).

Code de la fonction

```
simul <- function(n, B) {
# vérification
  stopifnot(is.numeric(n), is.numeric(B))

# initialisation
  res = 0 # le phénomène s'est produit 0 fois au début

  for (b in 1:B) {
    x <- rnorm(n) # simulation d'un vecteur x de taille 10
     if(any(abs(x) > 1.96)) {
       res <- res + 1   # on ajoute 1 à chaque fois que la condition est vérifiée
     }
```

```
  }
 print(paste("Le phénomène s'est produit", res/B*100, "% de fois"))
 }  # fin de la fonction
```

Test sur la fonction

```
for (n in c(10, 50, 100, 200)) {
  for (B in c(10, 50, 100)) {
    print(paste("n =", n, "et B =", B))
    simul(n, B)
  }
}
```

```
## [1] "n = 10 et B = 10"
## [1] "Le phénomène s'est produit 40 % de fois"
## [1] "n = 10 et B = 50"
## [1] "Le phénomène s'est produit 38 % de fois"
## [1] "n = 10 et B = 100"
## [1] "Le phénomène s'est produit 35 % de fois"
## [1] "n = 50 et B = 10"
## [1] "Le phénomène s'est produit 100 % de fois"
## [1] "n = 50 et B = 50"
## [1] "Le phénomène s'est produit 88 % de fois"
## [1] "n = 50 et B = 100"
## [1] "Le phénomène s'est produit 93 % de fois"
## [1] "n = 100 et B = 10"
## [1] "Le phénomène s'est produit 100 % de fois"
## [1] "n = 100 et B = 50"
## [1] "Le phénomène s'est produit 100 % de fois"
## [1] "n = 100 et B = 100"
## [1] "Le phénomène s'est produit 100 % de fois"
## [1] "n = 200 et B = 10"
## [1] "Le phénomène s'est produit 100 % de fois"
## [1] "n = 200 et B = 50"
## [1] "Le phénomène s'est produit 100 % de fois"
## [1] "n = 200 et B = 100"
## [1] "Le phénomène s'est produit 100 % de fois"
```

3 Utiliser la famille des fonctions *apply()*

En terme de clarté dans le code et en temps calcul, on a beaucoup à gagner à utiliser une de ces fonctions plutôt que de faire des boucles. On a déjà vu des exemples d'utilisation de ces fonctions dans le chapitre précédent, mais comme elles sont très importantes dans le langage **R**, on prend le temps ici de les revoir et de les appronfondir.

3.1 La fonction *apply()*

La syntaxe de la fonction *apply()* est la suivante :

```
apply(X, MARGIN, FUN, ...)
```

où **X** est un tableau de donnée de type **array** (typiquement un objet **matrix** ou **data.frame**), **MARGIN** est la(les) dimension(s) sur laquelle (lesquelles) on va appliquer la fonction **FUN**. Si n est le nombre de

composantes de **MARGIN**, alors la fonction renvoie un vecteur de taille n.

Exemple 1 : le jeu de données **iris** est un **data.frame** qui est un objet à deux dimensions.

```
dim(iris)
```

```
## [1] 150   5
```

L'espace des individus est représenté en lignes (dimension 1 de taille 150) et l'espace des variables est représenté en colonne (dimension 2 de taille 5). Ici, on va s'intéresser uniquement aux variables quantitatives (de type **numeric**) car il est difficile d'utiliser une même fonction sur des variables quantitatives et qualitatives en même temps. C'est pourquoi on créé l'objet **iris2** qui ne contient que les variables quantitatives :

```
iris2 <- iris[, 1:4]
```

L'objet **iris2** a deux dimensions. On a donc 2 possibilités pour renseigner l'argument **MARGIN** : **MARGIN=1**, **MARGIN=2**. On va utiliser la fonction *mean()* comme argument **FUN**. Les résultats sont les suivants :

```
apply(iris2, 1, mean)
```

```
##   [1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400 2.700
##  [12] 2.500 2.325 2.125 2.800 3.000 2.750 2.575 2.875 2.675 2.675 2.675
##  [23] 2.350 2.650 2.575 2.450 2.600 2.600 2.550 2.425 2.425 2.675 2.725
##  [34] 2.825 2.425 2.400 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675
##  [45] 2.800 2.375 2.675 2.350 2.675 2.475 4.075 3.900 4.100 3.275 3.850
##  [56] 3.575 3.975 2.900 3.850 3.300 2.875 3.650 3.300 3.775 3.350 3.900
##  [67] 3.650 3.400 3.600 3.275 3.925 3.550 3.800 3.700 3.725 3.850 3.950
##  [78] 4.100 3.725 3.200 3.200 3.150 3.400 3.850 3.600 3.875 4.000 3.575
##  [89] 3.500 3.325 3.425 3.775 3.400 2.900 3.450 3.525 3.525 3.675 2.925
## [100] 3.475 4.525 3.875 4.525 4.150 4.375 4.825 3.400 4.575 4.200 4.850
## [111] 4.200 4.075 4.350 3.800 4.025 4.300 4.200 5.100 4.875 3.675 4.525
## [122] 3.825 4.800 3.925 4.450 4.550 3.900 3.950 4.225 4.400 4.550 5.025
## [133] 4.250 3.925 3.925 4.775 4.425 4.200 3.900 4.375 4.450 4.350 3.875
## [144] 4.550 4.550 4.300 3.925 4.175 4.325 3.950
```

```
apply(iris2, 2, mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

Pour **MARGIN=1**, on a appliqué la fonction *mean()* sur toutes les composantes de la 1ère dimension (i.e. les 150 individus). Pour **MARGIN=2**, on a appliqué la fonction *mean()* sur toutes les composantes de la 2ème dimension (i.e. les 4 variables).

Remarque : on aurait pu faire exactement le même calcul en utilisant une boucle **for**. Par exemple :

```
res <- numeric(ncol(iris2))
for (k in 1:ncol(iris2))
  res[k] <- mean(iris2[,k])
print(res)
```

```
## [1] 5.843333 3.057333 3.758000 1.199333
```

On voit donc qu'on a intérêt à utiliser la fonction *apply()*, plus élégante et moins coûteuse en lignes de codes.

Exemple 2 : le jeu de données **iris3** (prédéfini sous **R**) est un objet de type **array**. Il s'agit du jeu de données **iris**, mais qui a été disposé sous la forme d'un tableau à 3 dimensions (il faut donc essayer d'imaginer des données stockées dans un cube) :

```
dim(iris3)
```

```
## [1] 50  4  3
```

La première dimension est la dimension des individus (il y en a 50), la seconde est la dimension des variables (il y en a 4) et la troisième dimension est la dimension des espèces (il y en a 3). Pour visualiser ce à quoi ressemble un tel objet, il faudrait imaginer un tableau sous forme d'un cube. Comme **R** représente difficilement la 3D, lorsqu'on affiche l'objet *print(iris3)*, **R** nous renvoie 3 tableaux : 1 tableau à 2 dimensions (individus (dimension 1) $\times$ variables (dimension 2)) par espèce (dimension 3). Comme l'objet a 3 dimensions, on a donc 3 possibilités pour renseigner l'argument **MARGIN** : **MARGIN=1**, **MARGIN=2** et **MARGIN=3**, auxquelles viennent s'ajouter les croisements possibles entre dimensions : **MARGIN=c(1,2)**, **MARGIN=c(1,3)** et **MARGIN=c(2,3)**. Bien entendu, parmi toutes ces combinaisons possibles, toutes ne sont pas intéressantes... Ici on souhaiterait connaître la moyenne des variables en fonction des espèces. On s'intéresse donc à la fois à la dimension des variabes (la dimension 2) et celle des espèces (la dimension 3). Pour effectuer la moyenne des variables en fonction des espèces, on fait donc :

```
apply(iris3, c(2, 3), mean)
```

```
##          Setosa Versicolor Virginica
## Sepal L.  5.006      5.936     6.588
## Sepal W.  3.428      2.770     2.974
## Petal L.  1.462      4.260     5.552
## Petal W.  0.246      1.326     2.026
```

Remarque : comme la taille de (dimension 2 $\times$ dimension 3) est (4×3), la fonction *apply()* renvoie bien un objet de dimension 4×3. Si on avait utilisé les boucles **for**, il aurait fallu utilisé le code suivant :

```
res <- matrix(0, dim(iris3)[2], dim(iris3)[3])
for (i in 1:nrow(res))
  for (j in 1:ncol(res))
    res[i, j] <- mean(iris3[, i, j])
```

3.2 Les fonctions *lapply()* et *sapply()*

La syntaxe de la fonction lapply() est la suivante :

```
lapply(X, FUN, ...)
```

X est un objet de type **list** (*data.frame* inclus) et renvoie un objet de classe **list** de la même taille que **X**. Contrairement à la fonction *apply()*, il n'est pas nécessaire de préciser la dimension de l'objet sur laquelle on va appliquer la fonction, puisqu'en quelque sorte il n'y qu'une seule dimension (le nombre d'éléments de la **list**). On rappelle qu'un *data.frame* est équivalent à une **list** dont les éléments sont les variables du **data.frame** prise une à une. C'est pourquoi

```
length(iris2)
```

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

renvoie la valeur 4. Ainsi, lorsqu'on applique la fonction *lapply()* sur un **data.frame**, on applique la fonction **FUN** sur chacune des variables. Par exemple :

```
lapply(iris2, mean)
```

```
## $Sepal.Length
## [1] 5.843333
##
## $Sepal.Width
## [1] 3.057333
```

```
##
## $Petal.Length
## [1] 3.758
##
## $Petal.Width
## [1] 1.199333
```

Si on avait utiliser la boucle **for** pour faire le calcul précédent, on aurait fait :

```
res <- vector("list", length(iris2))
for (k in 1:length(res))
  res[[k]] <- mean(iris2[[k]])
```

Un objet de type **list** n'est pas nécessairement commode à manipuler si on veut ensuite faire du calcul. Du coup, on peut utiliser la fonction *sapply()* (il s'agit d'une version arrangée de la fonction *lapply()*) qui retourne un vecteur ou une **matrix** au lieu d'un objet **list** :

```
sapply(iris2, mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

3.3 Créer sa propre fonction et la coupler avec *lapply()*

Jusqu'à présent, on a utilisé des fonction prédéfinies (comme la fonction *mean()*) à la place de l'argument **FUN**. On peut si l'on veut, utiliser une fonction programmée par nous-mêmes. Reprenons le jeu de données **iris2**. On souhaite transformer toutes les variables quantitatives en variables qualitatives. Pour cela, on écrit la fonction **f** qui prend comme argument d'entrée une variable x et retourne la variable transformée en 2 classes selon que les valeurs sont au-dessus ou au-dessous de la médiane :

```
f <- function(x) {
  res <- ifelse(x > median(x), "++", "--")
  return(res)
}
```

Pour appliquer cette fonction à chaque variable de **iris2**, il nous reste plus qu'à appliquer la fonction **sapply()** à l'objet **iris2** en précisant l'argument **FUN=f**.

```
iris2b <- sapply(iris2, f)
head(iris2b, 10)
```

```
##       Sepal.Length Sepal.Width Petal.Length Petal.Width
##  [1,] "--"         "++"        "--"         "--"
##  [2,] "--"         "--"        "--"         "--"
##  [3,] "--"         "++"        "--"         "--"
##  [4,] "--"         "++"        "--"         "--"
##  [5,] "--"         "++"        "--"         "--"
##  [6,] "--"         "++"        "--"         "--"
##  [7,] "--"         "++"        "--"         "--"
##  [8,] "--"         "++"        "--"         "--"
##  [9,] "--"         "--"        "--"         "--"
## [10,] "--"         "++"        "--"         "--"
```

3.4 La fonction *tapply()*

La syntaxe de la fonction est la suivante :

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

Cette fois-ci, **X** est un vecteur et **INDEX** est une **list** de **factor**.

Exemple 1 : on souhaite calculer la moyenne par espèce de la variable **Sepal.Length** du jeu de données **iris**.

```
tapply(iris$Sepal.Length, list(iris$Species), mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

Exemple 2 : on souhaite faire le même calcul que précédemment, mais pour chacune des variables du jeu de données **iris2**. Pour cela, il suffit d'utiliser la fonction *sapply()* et prendre comme argument **FUN**, la fonction *tapply()* telle qu'utilisée ci-dessus. On a vu que la fonction *tapply()* a 3 paramètres d'entrée. Dans le **FUN** de *sapply()*, on n'est pas obligé de préciser le 1er argument qui correspond à la variable renvoyée par la fonction *sapply()*. Les deux autres arguments doivent être précisés après le **FUN**. Voici le résultat :

```
sapply(iris2, tapply, list(iris$Species), mean)
```

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

4 Exercices

On considère ici le jeu de données **diamants** que nous avons déjà vu dans les chapitres précédents :

```
load(file("http://www.thibault.laurent.free.fr/cours/Ressource/diamants.RData"))
```

On affiche les premières lignes de ce jeu de données :

```
head(diamants)
```

```
##   carat       cut color clarity depth table price    x    y    z
## 1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
## 2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
## 3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
## 4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
## 5  0.31      Good     J     SI2  63.3    58   335 4.34 4.35 2.75
## 6  0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96 2.48
```

```
str(diamants)
```

```
## 'data.frame':    53940 obs. of  10 variables:
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

L'exercice consiste à programmer deux fonctions réalisant respectivement l'analyse univarié d'une variable quantitative en fonction d'une variable qualitative (fonction *univarie()*) et l'étude de la liaison entre 2 variables quantitatives en fonction d'une variable qualitative (fonction *bivarie()*).

Important : les fonctions que nous allons créer devront pouvoir s'appliquer sur n'importe quel jeu de données, pas uniquement **diamants**.

4.1 Fonction *univarie()*

La fonction *univarie()* aura pour arguments d'entrée :

- **table_df**, un objet de type **data.frame**,
- **nom_var_quanti**, un objet de type **character**, le nom de la variable quantitative étudiée (celle-ci devant se trouver dans **table**),
- **nom_var_quali**, un objet de type **character**, le nom de la variable catégorielle (celle-ci devant se trouver dans **table**).

A partir de ces 3 arguments, retourner un **data.frame** contenant le nom des modalités de **nom_var_quali** en ligne et l'effectif, la moyenne, la médiane, l'écart-type, le minimum et le maximum de la variable **nom_var_quanti** en colonnes. En outre, elle devra réaliser le graphique des boîtes à moustaches parallèles.

Indications :

Un algorithme pour la construction de cette fonction est :

- 1. Vérifier que **nom_var_quanti** et **nom_var_quali** sont bien des noms de variables de **table_df**.
- 2. Vérifier que la variable catégorielle est bien de type **factor** ou **character**
- 3. Préparer la table de résultats. On pourra utiliser la fonction *tapply()*
- 4. Effectuer le graphique
- 5. Faire "sortir" les résultats. Pour information, les boites à moustaches parallèles se font avec la fonction *boxplot()*. On pourra utiliser les `...` pour utiliser les paramètres optionnels de *boxplot()*

Solution :

On propose la solution suivante pour répondre à l'exercice. Bien évidemment, il existe d'autres façons pour y arriver, mais le but est d'essayer d'avoir un code aussi clair et simple que possible. Remarque : on a utilisé les fonctions *stopifnot()* et *tapply()* plutôt que les structures (**if/else** et **for**).

```
univarie <- function(table_df, nom_var_quanti, nom_var_quali, ...) {
# Vérification 1
  stopifnot(nom_var_quanti %in% colnames(table_df), nom_var_quali %in% colnames(table_df))

# Vérification 2
  stopifnot(is.factor(table_df[, nom_var_quali]) | is.character(table_df[, nom_var_quali]))

# affectation
  vecteur_numeric <- table_df[, nom_var_quanti]
  vecteur_quali <- table_df[, nom_var_quali]

# Création du data.frame
 resultat <- data.frame(n = tapply(vecteur_numeric, vecteur_quali, length),
            moyenne = tapply(vecteur_numeric, vecteur_quali, mean),
            mediane = tapply(vecteur_numeric, vecteur_quali, median),
            ecart.type = tapply(vecteur_numeric, vecteur_quali, sd),
```

```
            minimum = tapply(vecteur_numeric, vecteur_quali, min),
            maximum = tapply(vecteur_numeric, vecteur_quali, max))

# Réalisation du graphique
 boxplot(vecteur_numeric ~ vecteur_quali, ...)

# sortie
 return(resultat)
}
```

Explication : dans la fonction *boxplot()*, on a utilisé la symbole $Y \sim X$. Cette syntaxe est appelée **formula** et signifie que la variable Y est expliquée par la variable X. Quand cette syntaxe est utilisée à l'intérieure de la fonction *boxplot()*, **R** comprend que le graphique sera une boîte à moustache parallèle où Y sera un vecteur de valeurs **numeric** et **X** un vecteur de **factor** (ou **character**).

Remarque : il faut faire attention à distinguer **nom_var_quanti** et **nom_var_quali** qui sont juste des objets de type **character** contenant le nom des variables, avec les objets que nous avons créés à l'intérieur de la fonction **vecteur_numeric** et **vecteur_quali** qui sont bien des vecteurs contenant les valeurs observées.

Application :

On vérifie d'abord que les conditions d'arrêts fonctionnent bien :

```
univarie(diamants, "var1", "var2")
```

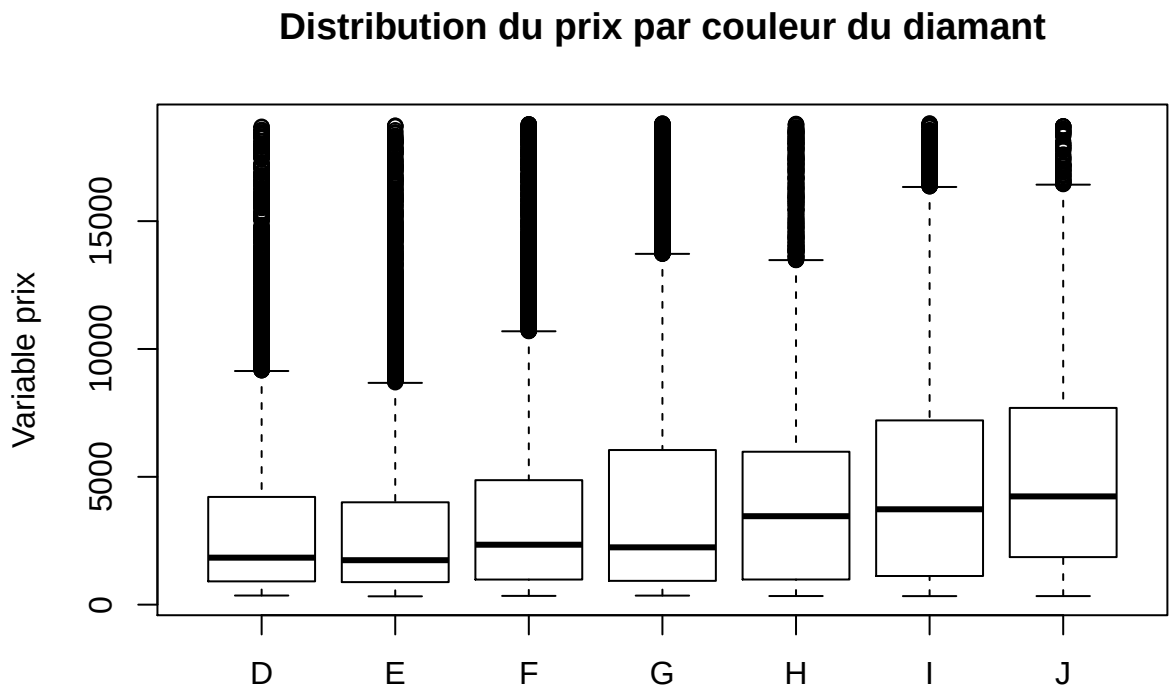
```
## Error in univarie(diamants, "var1", "var2"): nom_var_quanti %in% colnames(table_df) is not TRUE
```

```
univarie(diamants, "crim", "cadr")
```

```
## Error in univarie(diamants, "crim", "cadr"): nom_var_quanti %in% colnames(table_df) is not TRUE
```

On teste ensuite sur le jeu de données **diamants** :

```
univarie(diamants, nom_var_quanti = "price", nom_var_quali = "color",
  main = "Distribution du prix par couleur du diamant",
  ylab = "Variable prix")
```



```
##       n  moyenne mediane ecart.type minimum maximum
## D  6775 3169.954  1838.0   3356.591     357   18693
## E  9797 3076.752  1739.0   3344.159     326   18731
## F  9542 3724.886  2343.5   3784.992     342   18791
## G 11292 3999.136  2242.0   4051.103     354   18818
## H  8304 4486.669  3460.0   4215.944     337   18803
## I  5422 5091.875  3730.0   4722.388     334   18823
## J  2808 5323.818  4234.0   4438.187     335   18710
```

Quand on écrit une fonction, on essaye de faire en sorte qu'elle puise s'utiliser dans plusieurs cas de figure. Ici, on a codé la fonction pour qu'elle puisse être utiliser sur n'importe quel jeu de données. On pourra tester la fonction *univarie()* sur le jeu de données **iris** :

```
univarie(iris, nom_var_quanti = "Sepal.Length", nom_var_quali = "Species",
  main = "Distribution de la variable pétale par espèce", ylab = "Variable pétale")
```

4.2 Fonction *bivarie()*

La fonction *bivarie()* aura pour arguments d'entrée :

- **table_df**, un objet de type **data.frame**,
- **nom_vars_quanti**, un vecteur de **character** de taille 2 contenant le nom de deux variables quantitatives
- **nom_var_quali**, un objet de type **character**, le nom de la variable catégorielle (celle-ci devant se trouver dans **table_df**).

A partir de ces 3 arguments, retourner un **data.frame** contenant le nom des modalités de la variable portant le nom **nom_var_quali** en ligne et l'effectif ainsi que le coefficient de corrélation des variables portant le nom **nom_vars_quanti** en colonnes. En outre, elle devra réaliser le graphique pour chaque modalité de la variable catégorielle, le nuage de points des 2 variables (la première étant représentée sur l'axe des abscisses la seconde sur l'axe des ordonnées). On rappelle que la formule de corrélation linéaire de Pearson s'écrit :

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x}).(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}.\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} = \frac{(\sum_{i=1}^{n} x_i y_i) - \bar{x}\bar{y}}{\sqrt{(\sum_{i=1}^{n} x_i^2) - \bar{x}^2}.\sqrt{(\sum_{i=1}^{n} y_i^2) - \bar{y}^2}}$$

Indications:

Un algorithme pour la construction de cette fonction est :

- 1. Vérifier que **nom_vars_quanti** est un vecteur de taille 2 correspondant à des noms de colonnes de **table**.
- 2. Vérifier que **nom_var_quali** est le nom d'une variable de **table_df** de type **factor** ou **character**.
- 3. Identifier les modalités de la variable portant le nom **nom_var_quali**, faire une boucle **for** pour calculer le coefficient de corrélation des variables portant le nom **nom_vars_quanti** et ajouter un nuage de points au graphique à chaque étape de la boucle. Pour information, les nuages de points se font avec la fonction *plot()*. On pourra utiliser les **...** pour utiliser les paramètres optionnels de *plot()*
- 4. Faire "sortir" les résultats.

Solution :

Dans cet exemple, il n'est pas trivial de faire appel à la famille des fonctions *apply()* dans la mesure où on fait un calcul sur 2 variables en même temps. Du coup, on passe par l'utilisation de la boucle **for**. Il faut noter

qu'on a pris le soin de déterminer à l'avance le nombre de calcul qu'on allait faire (qui dépend du nombre de modalités) et on a donc mis la matrice de résultats dans la bonne dimension avant de faire la boucle.

```
bivarie <- function(table_df, nom_vars_quanti, nom_var_quali, ...) {
 # Vérification 1
  stopifnot(length(nom_vars_quanti) == 2, all(nom_vars_quanti %in% colnames(table_df)))

 # Vérification 2
  stopifnot(nom_var_quali %in% colnames(table_df),
            is.factor(table_df[, nom_var_quali]) | is.character(table_df[, nom_var_quali]))

 # on identifie les modalités et leur nombre
  # remarque : cela dépend du type de la variable quali (character ou factor)
  if (is.factor(table_df[, nom_var_quali])) {
    modalites <- levels(table_df[, nom_var_quali])
  } else {
    modalites <- unique(table_df[, nom_var_quali])
  }

 n.modalites <- length(modalites)

 # on prépare la matrice contenant les effectifs
 # et les coeff. de corrélation par modalité
 res <- matrix(0, n.modalites, 2)

 # options pour représenter les graphiques
 # nous les verrons en détails dans le chapitre suivant
 op <- par(mfrow = n2mfrow(n.modalites),
           mar = c(2, 3, 2, 2), oma = c(1, 1, 0, 0), mgp = c(2, .4, 0),
           cex.main = .75, cex.axis = .8, cex = .8, cex.lab = .8)

 # on boucle
 for(k in 1:n.modalites) {
   # on identifie les indices associés à la modalité k
   ind.k <- which(table_df[, nom_var_quali] == modalites[k])
   # on remplit les effectifs
   res[k, 1] <- length(ind.k)
   # on calcule le coefficient de corrélation
   res[k, 2] <- cor(table_df[ind.k, nom_vars_quanti[1]], table_df[ind.k, nom_vars_quanti[2]])
   # on dessine le nuage de points correspondants
   plot(table_df[ind.k, nom_vars_quanti[1]], table_df[ind.k, nom_vars_quanti[2]],
        main = paste("Modalité", modalites[k], "(r =", round(res[k, 2], 2),")"),
        xlim = range(table_df[, nom_vars_quanti[1]]),
        ylim = range(table_df[, nom_vars_quanti[2]]),
        ...)
  }

 # on retourne le résultat sous forme de data.frame
 return(data.frame(mod = modalites, n = res[,1], cor = res[,2]))
}
```

Application

On vérifie d'abord que les conditions d'arrêts fonctionnent bien :

```
bivarie(diamants, c("var1", "var2"), "color")
```

```
## Error in bivarie(diamants, c("var1", "var2"), "color"): all(nom_vars_quanti %in% colnames(table_df))
```

```
bivarie(diamants, nom_vars_quanti = "price", nom_var_quali = "color")
```

```
## Error in bivarie(diamants, nom_vars_quanti = "price", nom_var_quali = "color"): length(nom_vars_quant
```

On teste ensuite sur le jeu de données **diamants** :

```
bivarie(diamants, nom_vars_quanti = c("carat", "price"), nom_var_quali = "cut",
  xlab = "diametre du diamant", ylab = "Prix")
```

```
##         mod     n       cor
## 1      Fair  1610 0.8592985
## 2      Good  4906 0.9224716
## 3 Very Good 12082 0.9263704
## 4   Premium 13791 0.9250047
## 5     Ideal 21551 0.9311760
```