

Timothée Poisot

# Introduction à **R** Par la pratique

UQAR — Juin 2012



# ***Table des matières***

<b>Table des matières</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Environnement de travail . . . . .	2
1.2 Types de données . . . . .	4
<b>2 Lecture et écriture des données</b>	<b>17</b>
2.1 Lecture de données . . . . .	17
2.2 Écriture de données . . . . .	22
2.3 Export formaté . . . . .	23
2.4 Mises en application . . . . .	26
2.5 Solution des mises en application . . . . .	27
<b>3 Opérations sur les tables de données</b>	<b>29</b>
3.1 Travail sur les lignes et colonnes . . . . .	29
3.2 Division et traitement par niveau . . . . .	29
3.3 Traitement des données . . . . .	33
3.4 Sélection d'un sous-ensemble . . . . .	35
3.5 Tables de contingence . . . . .	36
3.6 Jonctions de tables de données . . . . .	37
<b>4 Introduction à la programmation</b>	<b>39</b>
4.1 Boucles . . . . .	39
4.2 Tests . . . . .	42
4.3 Fonctions . . . . .	44
4.4 Mise en application . . . . .	48
4.5 Solution des mises en application . . . . .	50
<b>5 Graphiques</b>	<b>51</b>
5.1 Principaux types de visualisations . . . . .	51
5.2 Ajout d'éléments sur un graphique . . . . .	69
5.3 Enregistrement des figures . . . . .	76
5.4 Mise en application . . . . .	77
5.5 Solution des mises en application . . . . .	78
<b>Bibliographie</b>	<b>85</b>



## Introduction

L'objectif de ce cours est de fournir une introduction générale à R [1], un langage qui est en train de devenir un standard en analyse de données et en calcul scientifique. R est souvent présenté dans l'optique d'une utilisation pour les statistiques. Si c'est effectivement une des capacités les plus souvent utilisées de ce langage, ce cours n'abordera pas ce domaine. L'objectif des 6 séances est plutôt de fournir, au travers d'exemples et de mises en application, un aperçu suffisamment vaste et complet de R pour le rendre utilisable dans la plus grande majorité des situations : nous aborderons donc les bases du langage, en commençant par les types d'objets utilisés par R, puis consacrerons une partie importante du temps à lire, écrire, et manipuler des jeux de données. Les deux dernières séances sont consacrées à la programmation et aux graphiques, qui permettront d'aller plus loin par la suite. À chaque séance, vous êtes encouragés à aller lire l'aide des différentes fonctions abordées, pour mieux comprendre les différentes options possibles : nous verrons les options principales, mais la maturité du langage R fait qu'il est impossible de couvrir tous les usages possibles en seulement 6 séances.

Si R a acquis une très grande popularité dans le milieu de la recherche, c'est parce qu'il permet très facilement de rajouter des fonctionnalités. *via* le système de *packages*. En écologie et biologie évolutive, les principaux sont *vegan* (analyse de structure des communautés), *simcol* (simulation de systèmes dynamiques), *ape* (analyses phylogénétiques), *bipartite* (analyse des réseaux de pollinisation), et *emdbook* (modèles statistiques). Chacun de ces *packages* est accompagné d'un livre ou d'une série de publications, et couvrent une variété énorme d'analyses. Depuis plus récemment, le projet *ROpenSci*<sup>1</sup> propose des *packages* permettant d'interagir avec les principales bases de données en écologie et évolution, et notamment de réaliser des analyses bibliométriques. Il est aussi de plus en plus fréquent de voir des articles dans lesquels les auteurs ont écrit un *package* R qui permet de reproduire leurs analyses, ou au moins mis en *supplementary material* des nouvelles fonctions utilisées dans leur travail.

Le cours s'étend sur 3 jours, et se compose de 6 séances couvrant chacune une demie journée. L'objectif est qu'à l'issue des 6 séances, vous soyez capables de comprendre le fonctionnement de R (séance 1), de lire, et écrire des données (séance 2), et de les manipuler (séance 3). La séance 4 comporte une introduction à la programmation, qui vous permettra de réaliser des choses plus poussées. La séance 5 est consacrée aux graphiques dans R, et couvre les outils de base. La séance 6

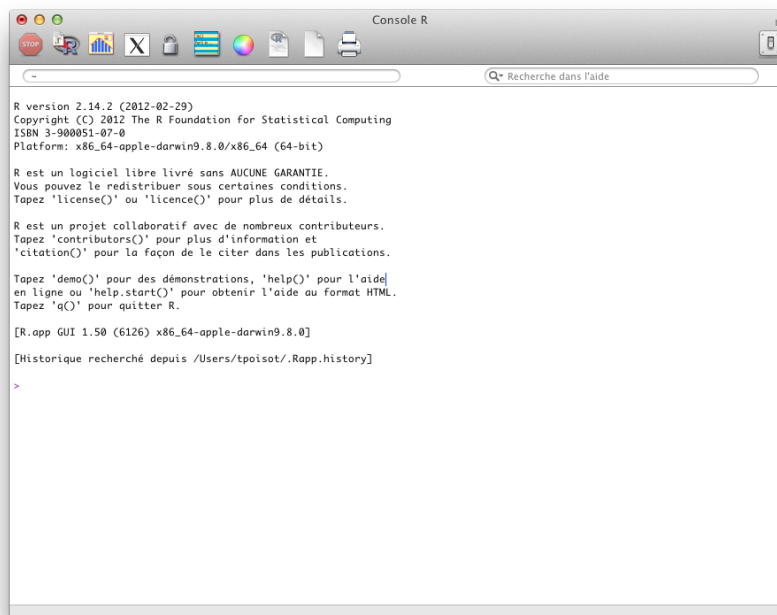
---

1. <http://ropensci.org/>

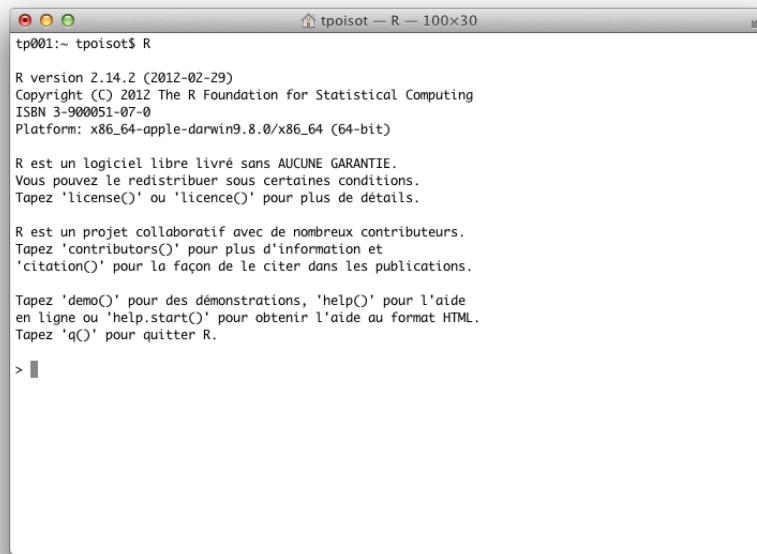
est prévue pour répondre à des questions plus générales. Chaque séance se déroule en deux temps. Premièrement, une introduction assez générale des concepts qui seront utilisés, avec des exemples de commandes et leurs résultats. Deuxièmement, une série de mises en application, qui consistent en un ou plusieurs petits problèmes généraux, mettant en application les notions acquises jusque là.

## 1.1 Environnement de travail

La première étape est de mettre en place notre environnement de travail. Si ce n'est pas déjà fait, commencez par installer R, disponible à <http://www.r-project.org/>. Au lancement de R s'affiche la fenêtre suivante :



On peut aussi lancer R depuis un terminal, en écrivant simplement R :



```
tpoisot:~ tpoisot$ R

R version 2.14.2 (2012-02-29)
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.

> █
```

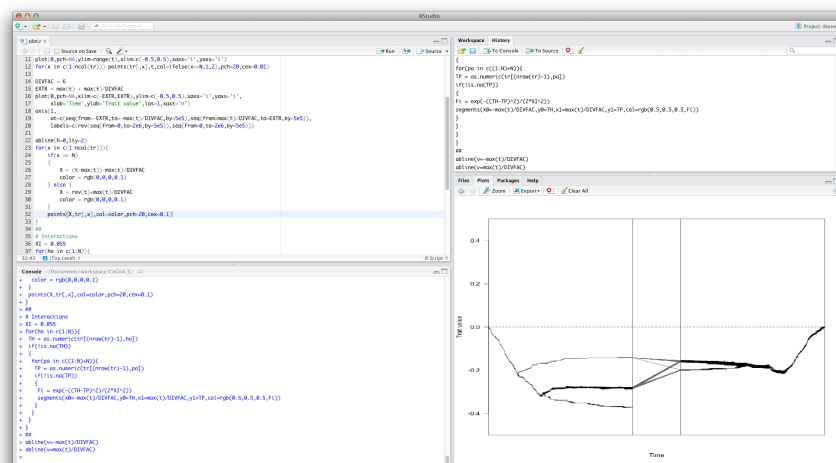
R n'est donc pas un logiciel comme *JMP*, *Statistica* ou *Excel*. Sa philosophie est entièrement différente : plutôt que d'utiliser des boutons et des menus, on communique avec R via des commandes, qu'on entre soit directement dans la console, soit dans des fichiers `.r`. R est accessible depuis de nombreux autres programmes, incluant *Komodo Edit*<sup>2</sup>, *Eclipse*<sup>3</sup>, et *NotePad ++*<sup>4</sup>. De tous, *Eclipse/StatET* est sans doute le plus avancé et le plus complet, mais aussi celui qui possède la courbe d'apprentissage la plus difficile. Apprendre à maîtriser les fonctions de base vous demandera sans doute du temps, mais une fois cette étape franchie, les fonctions les plus avancées fonctionnant sur la même logique, votre progression sera beaucoup plus rapide.

Dans le cadre de ce cours, nous utiliserons *RStudio*, un programme relativement récent qui possède l'ensemble des fonctions qui nous intéressent au jour le jour. *RStudio* est un environnement intégré de travail pour R qui est gratuit, multi-plateforme, et disponible en ligne à <http://rstudio.org/>. Ce programme permet de réaliser, dans le même environnement, la majorité des tâches qu'on peut vouloir effectuer dans R. *RStudio* possède aussi la propriété vraiment intéressante de pouvoir tourner sur un serveur web : une fois un serveur de ce type mis en place, vous pouvez accéder à *RStudio* de n'importe où, à condition d'avoir un navigateur relativement respectueux des standards. *Chrome*, *Firefox* et *Safari* devraient n'avoir aucun problème.

Au lancement de *RStudio*, une fenêtre ressemblant à celle ci-dessous s'affiche :

---

2. <http://www.sciviews.org/SciViews-K/>  
3. <http://www.walware.de/goto/statet>  
4. <http://sourceforge.net/projects/npptor/>



Les différentes sections permettent d'avoir accès simultanément à la console R (en bas à gauche), à la fenêtre des scripts (en haut à gauche), à l'historique des dernières commandes, et aux graphiques (la disposition des fenêtres peut changer de celle présentée, et se règle dans le menu *Options*, puis *Pane layout*). Au cours de la première séance, en même temps que nous découvrirons les bases du langage R, nous survolerons les fonctionnalités de *RStudio*.

## 1.2 Types de données

Cette section présente les types de données comprises par R. C'est sans doute la section la plus importante de l'ensemble du cours, dans la mesure où toute la logique de R repose sur les différents types de données disponibles. Les langages de programmation stockent en général des variables de différents types d'une manière différente. Dans les langages de bas niveau, comme C, ces différences peuvent être relativement restrictives, pour, par exemple, convertir un entier en flottant. R est un langage dit faiblement typé, en ce que les valeurs peuvent facilement changer de type<sup>5</sup>.

Il existe deux niveaux d'organisation pour représenter des valeurs dans un format compris par R : les types de données, et les collections de données. Ce vocabulaire n'est certainement pas le plus précis, mais établit au moins une différence entre les valeurs (les données) qu'on veut regrouper dans des structures plus vastes, et ces structures en elles-mêmes (les collections).

### Types de valeurs

Dans R, stocker n'importe quelle valeur, ou collection de valeurs, dans un objet, se fait de la manière suivante :

```
objet <- valeur
```

On peut aussi utiliser indifféremment les opérateurs `<-` ou `=`. Ma préférence va au dernier, puisque dans une lecture rapide `a <- 2` ( $a = 2$ ) ressemble beaucoup (trop) à `a < -2` ( $a < -2$ ). Cependant,

5. Dans la réalité, R est surtout *très mal* typé, et il est beaucoup trop facile de faire des erreurs de type. Il faut donc une grande discipline de programmation, et surtout bien savoir sur quel type d'objet on travaille



les recommandations sur la présentation de la syntaxe de R vont dans le sens de l'utilisation de <- (entouré de deux espaces), et c'est cette notation qui sera utilisée ici.

### Numériques

R est extrêmement performant dans le stockage de nombres, ce qui ne devrait pas surprendre de la part d'un langage développé pour traiter des problèmes statistiques. Les exemples suivants montrent les différentes manières de déclarer un objet contenant un nombre :

```
a <- 0.1
b <- 1
c <- 1e-06
a
## [1] 0.1

a + b
## [1] 1.1

b
## [1] 1

c
## [1] 1e-06
```

Pour forcer R à afficher une valeur, on peut utiliser la commande `print`. Par exemple,

```
print(a)
## [1] 0.1
```

affichera toujours la valeur contenue dans `a`. Cette commande est importante si vous utilisez des fichiers R *via* la commande `source` (cf. séance suivante) ; dans ce cas, `print` est le seul moyen de forcer l'affichage d'une valeur.

### Chaînes

Les chaînes permettent de stocker des expressions textuelles. Par exemple,

```
texte <- "Hello, world!"
texte
## [1] "Hello, world!"
```

R permet de manipuler les chaînes, même s'il n'est pas le choix le plus recommandé dans ce domaine (PERL ou Python font un bien meilleur travail). On peut mentionner quelques commandes simples :

Pour coller plusieurs chaînes entre elles,

```
paste("Hello", "World!")

## [1] "Hello World!"

paste("Hello", "World!", sep = ", ")

## [1] "Hello, World!"
```

On peut aussi couper des chaînes sur un motif donné, par exemple, pour connaître l'extension d'un fichier en coupant sur un point :

```
filename <- "mon_fichier.dat"
strsplit(filename, "\\.")

## [[1]]
## [1] "mon_fichier" "dat"
##

strsplit(filename, "_")

## [[1]]
## [1] "mon"          "fichier.dat"
##
```

Le fait d'écrire `.` et pas simplement `.` vient du fait que R peut utiliser des expressions régulières pour la découpe des chaînes, par exemple couper au premier chiffre, et que `.` a dans ce contexte un sens différent du caractère `.`. On peut séparer une chaîne de caractères par lettres, de la manière suivante :

```
strsplit("abcdefgh", "")

## [[1]]
## [1] "a" "b" "c" "d" "e" "f" "g" "h"
##
```

R possède aussi deux vecteurs nommés `letters` et `LETTERS`, qui contiennent les 26 lettres en haut et bas de casse :

```
letters

## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

LETTERS

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

### Booléens

Le type booléen (logical) permet de représenter les états *vrai* (TRUE) et *faux* (FALSE). Il est possible d'utiliser des raccourcis, à savoir T et F. Par exemple :

```
vrai <- TRUE  
faux <- FALSE  
vrai  
  
## [1] TRUE  
  
faux  
  
## [1] FALSE  
  
vrai + faux  
  
## [1] 1  
  
vrai * faux  
  
## [1] 0
```

La manière d'utiliser les booléens est vue plus en détail dans la séance 4.

### Facteurs

Les facteurs sont une notion importante dans R. Ils permettent de stocker des données catégorielles, par exemple des traitements expérimentaux. On peut créer un facteur de la manière suivante :

```
DemoFacteur <- factor(c(1, 2, 3, 4, 3, 3, 3, 2, 1))  
DemoFacteur  
  
## [1] 1 2 3 4 3 3 3 2 1  
## Levels: 1 2 3 4
```

Les facteurs possèdent un `level`, c'est-à-dire les différentes catégories :

```
levels(DemoFacteur)  
  
## [1] "1" "2" "3" "4"
```

On peut facilement renommer les niveaux d'un facteur :

```
levels(DemoFacteur) <- c("a", "b", "c", "d")
DemoFacteur

## [1] a b c d c c c b a
## Levels: a b c d
```

## Conversions

R est un langage faiblement typé, dans la mesure où une commande comme

```
paste(99, "luftballons", sep = " ")

## [1] "99 luftballons"
```

est parfaitement valide. Pourtant, elle mélange un type numérique avec un type chaîne, ce qui provoquerait une erreur dans d'autres langages de plus bas niveau. R permet en effet de convertir les données d'un type à l'autre assez facilement :

```
as.numeric("4")

## [1] 4

as.character(2)

## [1] "2"

as.integer(2.4)

## [1] 2

as.logical(1)

## [1] TRUE

as.logical(0)

## [1] FALSE

as.logical(2)

## [1] TRUE

as.numeric(TRUE)

## [1] 1
```

```
as.factor(c(1, 2, 3, 4, 3))

## [1] 1 2 3 4 3
## Levels: 1 2 3 4

as.character(as.factor(c(1, 2, 3, 4, 3)))

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

as.vector(as.factor(c(1, 2, 3, 4, 3)))

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

Cette possibilité de convertir les variables d'un type à l'autre est importante si, par exemple, R importe des colonnes d'un tableau de données comme étant des facteurs :

```
mydata$x <- as.numeric(as.vector(mydata$x))
```

## Collections de valeurs

### Vecteurs et vectorisation

Le vecteur est, avec la matrice, l'objet le plus important de R. R est un langage dit *vectorisé*, c'est-à-dire qui peut traiter plusieurs valeurs regroupées dans un objet unique. Si on utilise une commande très simple, comme

```
2
```

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

on remarque que la sortie est `[1] 2`. L'indicateur `[1]` indique que la valeur retournée est le premier élément d'un vecteur. La puissance de la notation vectorielle est qu'on peut accéder à une partie du vecteur, avec un *indice*. Si on prend l'exemple suivant,

```
a <- 2
```

```
a[1]
```

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

```
a[2]
```

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

, accéder à la position 1 *via l'indice* `[1]`, on récupère la première valeur du vecteur `a`. Voilà une des particularités de R : tout objet est un vecteur ! Accéder à la position `[2]` retourne `NA`, parce que le vecteur `a` ne possède pas de deuxième position.

On peut créer des vecteurs dans R en utilisant la commande `c`.

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

R propose différents raccourcis pour créer rapidement des vecteurs. Par exemple, examinez le comportement des commandes suivantes :

```
seq(from = 0, to = 5, by = 1)
```

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

```
seq(from = 0, to = 10, length = 3)
```

```
## [1] 0 5 10
```

```
c(0:5)
```

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

L'avantage de la vectorisation est que R va automatiser une grande partie des opérations sur les vecteurs. Par exemple, examinez l'effet des commandes suivantes :

```
a <- c(1:10)
```

```
a/2
```

```
## [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
log(a, 10)
```

```
## [1] 0.0000 0.3010 0.4771 0.6021 0.6990 0.7782 0.8451 0.9031 0.9542 1.0000
```

```
sum(a)
```

```
## [1] 55
```

```
a * a
```

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

Avec les vecteurs vient le concept important de *recyclage*. Le recyclage consiste à répéter un vecteur autant de fois que nécessaire pour le rendre compatible avec un autre vecteur dans le cadre d'une opération. Par exemple, les commandes

```
c(1, 2, 3, 4, 5) + c(1, 2)
```

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

et

```
c(1, 2, 3, 4, 5) + c(1, 2, 1, 2, 1)
```

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

sont équivalentes. Le vecteur `c(1, 2)` du premier exemple est *recyclé* pour atteindre la longueur du premier vecteur.

La taille d'un vecteur est obtenue en utilisant la fonction `length` :

```
length(1)
```

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

```
length(c(1:5))
```

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

On peut créer facilement des vecteurs d'une taille désirée avec les fonctions `numeric` et `character` :

```
numeric(3)
```

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

```
character(5)
```

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

## Matrices

Le type matrice est central dans le fonctionnement de R. Une fois le principe des vecteurs compris, le fonctionnement des matrices est assez intuitif. Une matrice est en fait un vecteur à deux dimensions. Si un vecteur est une ligne, une matrice est une série de lignes et de colonnes. Dans R, on peut créer une matrice avec la commande `matrix`, de sorte que

```
test_mat <- matrix(0, ncol = 2, nrow = 4)
```

renvoie une matrice pleine de 0, avec 2 colonnes (`ncol`) et 4 lignes (`nrow`).

On peut connaître les dimensions d'une matrice de différentes manières :

```
dim(test_mat)
```

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

```
nrow(test_mat)
```

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

```
ncol(test_mat)
```

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

Accéder à une position particulière d'une matrice se fait en deux temps. D'abord, par le numéro de la ligne, ensuite par le numéro de la colonne. Par exemple, on peut fixer l'élément sur la première ligne, deuxième colonne de `test_mat` à 2, avec

```
test_mat[1, 2] <- 2
test_mat

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

Dans certaines situations, on peut souhaiter avoir accès à une ligne ou une colonne en particulier. R permet donc de ne spécifier qu'un numéro de ligne, ou un numéro de colonne :

```
test_mat[, 2]

## [1] 2 0 0 0

test_mat[1, ]

## [1] 0 2
```

Une matrice peut aussi posséder des noms de lignes et de colonnes :

```
colnames(test_mat) <- c("a", "b")
rownames(test_mat) <- c("A", "B", "C", "D")
test_mat

##   a b
## A 0 2
## B 0 0
## C 0 0
## D 0 0

colnames(test_mat)

## [1] "a" "b"
```

Cela permet aussi d'accéder plus facilement à certaines positions de la matrice :

```
test_mat["A", ]

## a b
## 0 2
```



```
test_mat["A", "b"]
```

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

## Listes

Un des derniers types d'objets qu'il faut connaître est les listes. Une liste, dans R, est une manière de stocker de l'information venant de source diverses, pour y accéder facilement. On verra dans la séance 3 que les listes permettent aussi de traiter très rapidement plusieurs jeux de données à la suite.

Pour créer une liste, on peut utiliser différentes méthodes :

```
n_list <- list(a = 1, b = 2, c = 3)
```

```
n_list
```

```
## $a
```

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

```
##
```

```
## $b
```

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

```
##
```

```
## $c
```

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

```
##
```

```
u_list <- list(1, 2, 3)
```

```
u_list
```

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

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

```
##
```

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

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

```
##
```

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

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

```
##
```

Les listes, comme `n_list`, peuvent être nommées :

```
names(n_list)
```

```
## [1] "a" "b" "c"
```

On peut accéder aux éléments des listes de différentes manières. Si la liste est nommée, la notation `liste$nom` est possible ; dans tous les autres cas, `liste[[indice]]` fonctionne.

```
n_list$a
## [1] 1
n_list[[1]]
## [1] 1
```

Les listes peuvent être utilisées dans le contexte de la structure `with`, qui permet d'accéder facilement aux différents éléments. Pour simplifier, `with` permet d'éviter d'écrire `liste$nom` pour n'écrire que `nom`; les deux commandes ci-dessous sont donc équivalentes :

```
n_list$a + n_list$b - n_list$c
## [1] 0
with(n_list, {
  a + b - c
})
## [1] 0
```

Dans certains cas, il peut être intéressant d'aplatir une liste en un vecteur. R propose la fonction `unlist` pour effectuer cette opération :

```
t_list <- list(1, 2, 3, 4, 5, 6)
t_list

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

```
unlist(t_list)
```

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

### *Data frames*

Les *data frames* sont un des objets les plus puissants de R, que nous verrons en détail dans les autres séances : la majorité des données que nous lirons seront stockées de cette manière.



## Lecture et écriture des données

R propose plusieurs manières de lire des données, depuis des fichiers textes ou des tableaux Excel. L'objectif de cette séance est de lire, et mettre en forme des données. Nous aborderons aussi les moyens de sauvegarder ces données sur le disque. Pour la durée de la séance, on suppose que l'ensemble des données qu'on veut lire sont stockées dans le répertoire `./data/`.

### 2.1 Lecture de données

La méthode la plus simple de stocker des données, et la seule que l'on devrait recommander si on veut s'assurer de pouvoir lire les données partout, en tout temps, est d'utiliser des fichiers texte. À la différence d'un fichier produit par *Excel* ou *OpenOffice Calc*, un fichier en texte brut ne contient pas d'autre informations que ce qu'on y a entré. Il est possible de lire dans n'importe quel programme, et son format ne changera *jamais* – sans mentionner que sa lecture ne coûte rien, et qu'il est extrêmement difficile de corrompre un fichier texte.

#### Depuis des fichiers textes

Dans cet exemple, on utilisera les données prises par KACI-CHAOUCH et al. [2] et POISOT et DESDEVICES [3] sur 189 parasites du genre *Lamellodiscus*, parasites de poissons communs en Méditerranée. Ces données correspondent aux relevés morphométriques effectuées sur les parties solides de l'appareil d'attachement. Les données sont classées selon l'espèce du parasite (sppar), et l'espèce de l'hôte sur lequel le parasite a été isolé (sphote).

```
morpho <- read.table("data/lamellodiscus.txt", h = TRUE, sep = "\t")
head(morpho)
```

##	sphote	sppar	para	a	b	c	d	f	g	aa	bb	cc	lm
## 1	Divu	eleg	elegDivu1	2.06	1.93	NA	NA	NA	NA	1.93	1.81	1.16	5.43
## 2	Divu	eleg	elegDivu2	1.93	1.82	1.51	0.41	0.57	0.20	1.77	1.67	1.20	2.35
## 3	Divu	eleg	elegDivu2	1.67	1.56	1.20	0.31	0.36	0.26	1.56	1.51	0.94	1.88
## 4	Disa	eleg	elegDisa1	1.46	1.41	1.04	0.47	0.67	0.31	1.25	1.20	0.83	1.46
## 5	Disa	eleg	elegDisa1	1.30	1.25	0.94	0.36	0.52	0.31	1.14	1.09	0.78	NA

```
## 6 Disa eleg elegDisa1 1.41 1.35 0.99 0.36 0.57 0.36 1.25 1.20 0.83 2.08
##    li
## 1 2.66
## 2 2.66
## 3 1.77
## 4 2.19
## 5 2.03
## 6 2.55
```

La première ligne comporte deux éléments importants : `h = TRUE`, et `sep = 't'`. L'argument `h` est l'abréviation de *header*, à savoir, est-ce que la première ligne donne le nom de la colonne de données. L'argument `sep` indique quel caractère est utilisé pour séparer les colonnes, ici une tabulation. Si les champs avaient été séparés par une espace, il aurait fallu utiliser `sep = ' '`. On peut aussi spécifier le séparateur décimal (`dec='.'`), ainsi que d'autres options, cf. `?read.table`.

Cette fonction renvoie un objet de type *data frame*. La commande `head` (resp. `tail`) donne accès aux premières (resp. dernières) lignes d'un *data frame*.

R donne aussi accès directement à la lecture des fichiers *comma separated value*, *csv*, via la commande `read.csv`. Cette fonction est en réalité un appel à `read.table` avec une partie des arguments par défaut pré-remplis pour lire le format *csv*.

Une fois les données lues, elles sont en général importées sous forme de *data.frame*. On peut voir les noms des colonnes :

```
colnames(morpho)
```

```
## [1] "sphote" "sppar" "para" "a" "b" "c" "d"
## [8] "f" "g" "aa" "bb" "cc" "lm" "li"
```

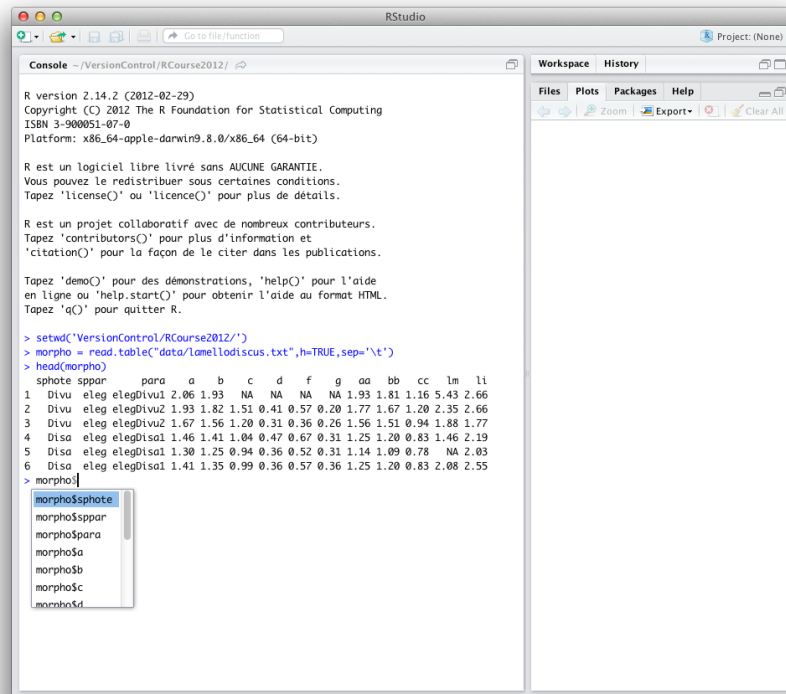
et afficher le contenu d'une des colonnes avec

```
morpho$a
```

```
## [1] 2.060 1.930 1.670 1.460 1.300 1.410 1.980 2.080 2.450 2.240 2.030
## [12] 2.080 1.980 2.140 2.290 2.190 2.350 2.240 2.030 2.840 1.090 1.820
## [23] 2.030 1.090 1.040 1.140 1.040 1.670 1.560 1.300 1.460 1.610 1.040
## [34] 0.990 NA 1.930 1.980 2.030 1.980 1.880 1.930 1.930 2.030 1.980
## [45] 1.930 2.030 2.080 2.080 2.140 2.140 1.250 0.900 0.900 1.700 1.800
## [56] 0.800 1.800 1.400 1.550 2.400 2.300 1.650 2.150 1.750 1.800 1.300
## [67] 1.700 1.700 1.700 1.500 1.400 1.650 1.700 2.240 1.700 1.100 NA
## [78] 1.150 1.250 1.000 1.050 1.700 1.600 1.850 1.850 1.900 1.850 1.900
## [89] 1.200 1.000 1.100 1.800 1.500 1.000 0.950 1.100 1.800 2.100 1.400
## [100] 1.400 1.300 1.350 1.400 1.450 1.000 1.200 1.100 1.100 0.900 1.000
## [111] 1.050 0.900 1.550 1.450 0.850 1.750 1.600 1.600 1.600 1.100 0.800
## [122] 1.400 1.500 1.400 0.850 1.450 0.900 1.450 1.550 0.900 1.700 1.750
## [133] 1.550 1.600 1.800 1.000 1.800 1.600 1.600 2.100 2.000 1.900 1.700
## [144] NA 1.400 1.700 1.800 0.700 1.400 1.500 1.400 1.400 1.300 1.200
## [155] 1.300 0.800 1.450 0.900 1.000 2.334 2.420 2.463 1.256 2.340 2.396
```

```
## [166] 2.323 1.815 1.300 1.826 1.850 1.832 1.772 1.869 1.932 1.905 1.813
## [177] 1.467 1.250 2.028 2.161 2.091 2.046 2.043    NA 2.025 1.555 2.031
## [188] 1.204 1.086
```

Dans *RStudio*, et la plupart des autres plugins permettant d'utiliser R dans un environnement de travail, après avoir entré le nom de la *data frame*, vous aurez accès en appuyant sur tabulation à un menu déroulant, lequel contient les noms des colonnes :



Cette fonctionnalité s'avère très utile quand on a tendance à oublier comment on a nommé ses variables.

La syntaxe `dataframe$nom` peut sembler lourde, et c'est pourquoi R propose une commande nommée `attach`. Avec la syntaxe suivante,

```
attach(morpho)
```

```
## The following object(s) are masked _by_ '.GlobalEnv':
```

```
##
```

```
##      a, b, c
```

```
a
```

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

```
detach(morpho)
```

on peut accéder directement aux noms des colonnes. C’est une des pires pratiques d’écriture de code possible, et il est très fortement déconseillé d’utiliser `attach` :

```
attach(morpho)

## The following object(s) are masked _by_ '.GlobalEnv':
##
##      a, b, c

a[c(1:10)]

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

a[2] <- 33333
a[c(1:10)]

## [1] 1 33333 3 4 5 6 7 8 9 10

detach(morpho)
morpho$a[c(1:10)]

## [1] 2.06 1.93 1.67 1.46 1.30 1.41 1.98 2.08 2.45 2.24
```

Alors qu’on modifie `a`, les modifications ne sont pas rapportées à `morpho$a`. Notez aussi que dans le *caas* ou des colonnes de la *data.frame* sur laquelle on travaille ont le même nom que des variables déjà existantes dans l’environnement de travail, il n’est pas toujours évident de prédire comment R se comporte. Ce n’est pas problématique dans le cas de colonnes nommées `a` ou `b`, mais ça l’est nettement plus dans le cas de `c`, une fonction nécessaire au bon fonctionnement de R.

Si vous souhaitez vraiment vous abstenir de l’appel complet au nom de la *data frame*, la construction `with` fonctionnera aussi bien :

```
with(morpho, {
  mean(a, na.rm = TRUE)
})

## [1] 1.612
```

Notez que `with` ne permet pas non plus de modifier `morpho$a` directement, mais permet au moins de ne pas “contaminer” votre environnement de travail avec différentes variables portant le même nom.

## Depuis des fichiers Excel

Il est fortement déconseillé de stocker des données importantes dans Excel ou Open Office (ou équivalent). Si vous avez à manipuler des données dans ce format, ma recommandation est de les exporter au format texte et de travailler avec sous cette forme. Dans tous les cas, R permet de *lire* ces données.

Pour avoir la possibilité de lire des fichiers *Excel*, il faut installer le *package* `gdata`.



```
install.packages("gdata")
```

Les *packages* sont des collections de code R écrites par la communauté, et rendues disponibles sur le *Comprehensive R Archive Network*<sup>1</sup>. Il existe des *packages* pour à peu près toutes les utilisations possibles.

Pour charger un *package*, il faut utiliser la commande `library` :

```
library(gdata)

## gdata: read.xls support for 'XLS' (Excel 97-2004) files ENABLED.
##
## gdata: read.xls support for 'XLSX' (Excel 2007+) files ENABLED.
##
## Attaching package: 'gdata'
##
## The following object(s) are masked from 'package:stats':
##
##      nobs
##
## The following object(s) are masked from 'package:utils':
##
##      object.size
##
```

Notez bien que l'utilisation de `gdata` demande que `perl`<sup>2</sup> soit installé sur votre machine. Sur une distribution de Linux ou Mac OS, ce sera le cas par défaut – il vous faudra l'installer vous même sur Windows.

Si vous utilisez des fichiers issus de Excel 2007, il faut avant tout chose lancer une fonction de `gdata` qui permet de lire ces fichiers :

```
installXLSXsupport(perl = "perl", verbose = FALSE)
```

Le dossier `data/` contient une feuille Excel au format `xls`, `data/netlists.xls`. Ce fichier comporte, pour une série de réseaux trophiques, leur nom, type, et le nombre d'espèces présentes à chaque niveau. On peut lire ce fichier avec la commande `read.xls` :

```
webs <- read.xls("data/netlists.xls", sheet = 1, h = TRUE)
head(webs)

##      name      type ltl utl
## 1 aishihik_p hostpara   7  29
## 2 AkatoreAtxt  trophic  85  85
## 3 AkatoreBtxt  trophic  58  58
```

---

1. <http://cran.r-project.org/>

2. <http://www.perl.org/>

```
## 4 arr_1_matr plantpoll 87 98
## 5 arr_2_matr plantpoll 42 61
## 6 arr_3_matr plantpoll 41 28
```

La commande `read.xls` prend les mêmes arguments que `read.table`, et rajoute un argument `sheet`, qui permet de déterminer quelle feuille sera importée. Le fonctionnement de cette fonction quand la feuille Excel contient des macros, de la mise en forme, etc, n'est pas prévisible. La meilleure solution reste toujours de convertir les données au format texte, ou au format csv, et de les importer dans R par `read.table`.

## 2.2 Écriture de données

R propose deux manières distinctes de sauvegarder les données. Soit en utilisant un fichier au format de R, soit dans un fichier en texte brut.

Pour sauvegarder un fichier de données au format R, on utilise la fonction `save` :

```
save(morpho, file = "data/morpho_data.Rdata")
```

L'extension `Rdata` vous permet, en double cliquant sur ce fichier, de le charger directement dans R. Vous pouvez aussi charger un fichier de ce type en utilisant la commande `load` :

```
rm(morpho)
ls()
```

```
## [1] "DemoFacteur" "a"          "b"          "c"          "faux"
## [6] "filename"     "n_list"     "t_list"     "test_mat"   "texte"
## [11] "thm"          "txt"        "u_list"     "vecteur"    "vrai"
## [16] "webs"
```

```
load("data/morpho_data.Rdata")
ls()
```

```
## [1] "DemoFacteur" "a"          "b"          "c"          "faux"
## [6] "filename"     "morpho"     "n_list"     "t_list"     "test_mat"
## [11] "texte"        "thm"        "txt"        "u_list"     "vecteur"
## [16] "vrai"         "webs"
```

```
head(morpho)
```

```
##   sphote sppar      para    a    b    c    d    f    g    aa    bb    cc    lm
## 1  Divu  eleg  elegDivu1 2.06 1.93  NA  NA  NA  NA 1.93 1.81 1.16 5.43
## 2  Divu  eleg  elegDivu2 1.93 1.82 1.51 0.41 0.57 0.20 1.77 1.67 1.20 2.35
## 3  Divu  eleg  elegDivu2 1.67 1.56 1.20 0.31 0.36 0.26 1.56 1.51 0.94 1.88
## 4  Disa  eleg  elegDisa1 1.46 1.41 1.04 0.47 0.67 0.31 1.25 1.20 0.83 1.46
## 5  Disa  eleg  elegDisa1 1.30 1.25 0.94 0.36 0.52 0.31 1.14 1.09 0.78  NA
## 6  Disa  eleg  elegDisa1 1.41 1.35 0.99 0.36 0.57 0.36 1.25 1.20 0.83 2.08
```

```
##      li
## 1 2.66
## 2 2.66
## 3 1.77
## 4 2.19
## 5 2.03
## 6 2.55
```

La commande `rm` permet de supprimer un objet de la mémoire de R – ce qui permet de ne plus se soucier de la version de `morpho` déjà chargée depuis le fichier texte, et la commande `ls` fait une liste de tous les objets présents dans l’environnement de travail. On voit que la commande `load` va remettre une copie de `morpho` dans l’environnement de travail.

L’avantage d’utiliser des fichiers `Rdata` est qu’ils sont compressés, donc prennent moins de place que les mêmes données en format texte brut. Si, par contre, vous voulez utiliser ces données dans un autre programme que R, il est important de pouvoir les exporter de manière lisible pour tous.

En parallèle à la lecture avec `read.table`, R propose la fonction `write.table`. Cette fonction permet d’écrire les données au format texte :

```
write.table(morpho, file = "data/morpho_txt.txt", sep = " ", dec = ".",
            col.names = TRUE, row.names = FALSE)
```

Cette ligne va écrire le contenu de `morpho` dans `data/morpho_txt.txt`, en séparant les colonnes avec une espace, en utilisant le point comme séparateur décimal, et avec uniquement les noms des colonnes. La fonction `write.table` admet un argument `append`, qui permet d’ajouter le contenu de l’objet au fichier choisi.

## 2.3 Export formaté

Une fois qu’on a lu le fichier de données, il peut être utile de l’exporter dans un format plus lisible. R propose *via* des *packages* additionnels, la possibilité de mettre en forme facilement des données. Pour ce-faire, nous avons besoin d’installer deux *packages* : `xtable` et `SortableHTMLTables`.

```
install.packages(c("SortableHTMLTables", "xtable"))
```

On charge ces *packages*,

```
library(SortableHTMLTables)
```

```
## Loading required package: testthat
## Loading required package: brew
```

```
library(xtable)
```

Le premier permet de créer un document au format `html` dans lequel on peut dynamiquement trier les colonnes en fonction de leur valeur. Par exemple, avec la base de données `morpho`,

Si vous allez dans le répertoire `data/morphoHTML`, et que vous ouvrez le fichier `output.html` dans n'importe quel navigateur, vous devriez obtenir le résultat suivant :

[illegible]

Le *package* `xtable` possède une fonctionnalité similaire, et permet d'exporter au format  $\text{\LaTeX}$  ou au format `html`. L'utilisation de la commande est très simple :

```
## % latex table generated in R 2.14.2 by xtable 1.7-0 package
## % Sun Jun 17 13:37:56 2012
## \begin{table}[ht]
## \begin{center}
## \begin{tabular}{rllllrrr}
## \hline
## & sphote & sppar & para & a & b & c & \\
## \hline
## 1 & Divu & eleg & elegDivu1 & 2.06 & 1.93 & & \\
## 2 & Divu & eleg & elegDivu2 & 1.93 & 1.82 & 1.51 & \\
## 3 & Divu & eleg & elegDivu2 & 1.67 & 1.56 & 1.20 & \\
## 4 & Disa & eleg & elegDisa1 & 1.46 & 1.41 & 1.04 & \\
## 5 & Disa & eleg & elegDisa1 & 1.30 & 1.25 & 0.94 & \\
## 6 & Disa & eleg & elegDisa1 & 1.41 & 1.35 & 0.99 & \\
## \hline
## \end{tabular}
## \end{center}
## \end{table}
```

L'avantage de `xtable` est que ce *package* contient des méthodes standard pour plusieurs analyses classiques en statistiques. Par exemple, on peut utiliser un objet `lm` au lieu d'un tableau, pour mettre en forme les résultats de manière cohérente :

```
x <- c(1:10)
y <- 2 * x + 1 + rnorm(length(x))
regression <- lm(y ~ x)
RegTable <- xtable(regression)
```

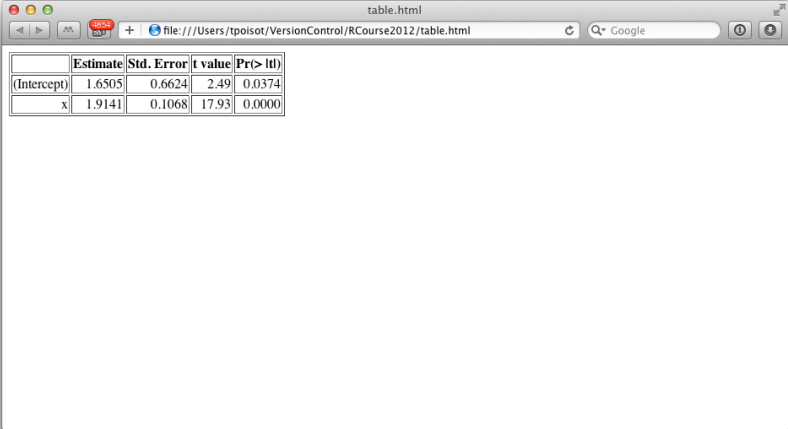
On peut imprimer cet objet de manières différentes, d'abord en html :

```
print(RegTable, type = "html")

## <!-- html table generated in R 2.14.2 by xtable 1.7-0 package -->
## <!-- Sun Jun 17 13:37:56 2012 -->
## <TABLE border=1>
## <TR> <TH> </TH> <TH> Estimate </TH> <TH> Std. Error </TH> <TH> t value </TH> <TH> Pr(> |t|) </TH>
## <TR> <TD align="right"> (Intercept) </TD> <TD align="right"> 1.4411 </TD> <TD align="right"> 0.9
## <TR> <TD align="right"> x </TD> <TD align="right"> 1.8713 </TD> <TD align="right"> 0.1525 </TD>
## </TABLE>

print(RegTable, type = "html", file = "table.html")
```

Ce qui donne le résultat suivant :



	Estimate	Std. Error	t value	Pr(>  t )
(Intercept)	1.6505	0.6624	2.49	0.0374
x	1.9141	0.1068	17.93	0.0000

On peut de la même manière exporter en  $\text{\LaTeX}$  :

```
print(RegTable)

## % latex table generated in R 2.14.2 by xtable 1.7-0 package
## % Sun Jun 17 13:37:56 2012
```

```
## \begin{table}[ht]
## \begin{center}
## \begin{tabular}{rrrrr}
## \hline
## & Estimate & Std. Error & t value & Pr(>|t|) \\
## \hline
## (Intercept) & 1.4411 & 0.9462 & 1.52 & 0.1663 \\
## x & 1.8713 & 0.1525 & 12.27 & 0.0000 \\
## \hline
## \end{tabular}
## \end{center}
## \end{table}

print(RegTable, file = "table.tex")
```

Ce qui donne le résultat suivant :

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	1.4411	0.9462	1.52	0.1663
x	1.8713	0.1525	12.27	0.0000

Cette extension est donc particulièrement puissante pour la rédaction de rapport, ou le maintien à jour de documents<sup>3</sup>.

## 2.4 Mises en application

### Ajout de deux tables à la suite

On dispose de deux séries de mesures différentes, contenues dans deux fichiers texte (`mes1.txt` et `mes2.txt`, tous les deux dans le répertoire `data`), dont les champs sont séparés par une espace. On souhaite regrouper ces deux séries de mesures dans un seul fichier texte, `mes3.txt`, contenu dans le même répertoire. Il existe (au moins) deux méthodes pour le faire.

3. Il faut noter que *knitr*, <http://yihui.name/knitr/>, une extension de R toujours activement en développement, permet de faire exactement la même tâche et bien plus encore – génération de page webs, insertion automatique des figures, etc... Cette extension a été utilisée pour réaliser ce document de cours. *RStudio* permet d'utiliser *knitr* directement, de manière très efficace et relativement simple.

## 2.5 Solution des mises en application

### Ajout de deux tables à la suite

La première étape est de lire les fichiers textes :

```
mes1 <- read.table("data/mes1.txt")
mes2 <- read.table("data/mes2.txt")
dim(mes1)

## [1] 22 21

dim(mes2)

## [1] 54 21
```

On peut maintenant combiner ces deux tableaux en un seul en ajoutant toutes les lignes de mes2 à la fin des lignes de mes1, en utilisant la commande `rbind` (*row bind*, littéralement adjoindre les lignes) :

```
mesTotal <- rbind(mes1, mes2)
dim(mesTotal)

## [1] 76 21
```

On peut vérifier que les dimensions de mesTotal sont bien la somme de celles de mes1 et mes2. Il ne reste maintenant plus qu'à écrire dans un nouveau fichier texte :

```
write.table(mesTotal, file = "data/mes3.txt", sep = " ", dec = ".")
```

La deuxième méthode consiste à ajouter les lignes de mes2 directement à la fin du fichier mes1.txt (faites donc une copie des deux avant de commencer!).

```
write.table(mes2, file = "data/mes1.txt", append = TRUE, sep = " ",
  dec = ".")
```

Notez que dans ce cas, il faut faire attention à ce que les deux fichiers soient formatés de la même manière, sinon la lecture sera quasi impossible après.





## Opérations sur les tables de données

### 3.1 Travail sur les lignes et colonnes

Dans une grande variété de situations, il peut être avantageux de répéter une opération sur toutes les lignes, ou toutes les colonnes. R propose une fonction pour automatiser ce traitement, *via* la fonction `apply`.

```
dat <- matrix(rnorm(100), nrow = 10)
apply(dat, 1, mean)

## [1] -0.03347 -0.36329  0.03966  0.49536 -0.06241 -0.49213  0.34352
## [8]  0.06440 -0.19710  0.25857

apply(dat, 2, var)

## [1] 0.7486 1.5183 0.4915 0.6506 1.0685 1.0710 1.4058 0.6280 0.4975 0.2700
```

### 3.2 Division et traitement par niveau

En utilisant différentes fonctions, on peut traiter facilement un jeu de données par «niveaux» d'un facteur (p.ex. traitement expérimental). En rechargeant les données *Lamellodiscus*, on peut par exemple chercher à connaître la moyenne et la variance de la taille de chaque pièce sclérifiée.

```
morpho <- read.table("data/lamellodiscus.txt", h = TRUE, sep = "\t")
```

L'étape suivante est de diviser les données, en utilisant la fonction `split`. Cette fonction prend une `data.frame`, la divise selon les valeurs de la colonne (ou combinaison de colonnes) choisie, et renvoie les sous-tableaux sous forme de liste.

```

morpho_split <- split(morpho, morpho$sppar)
names(morpho_split)

## [1] "conf" "dipl" "eleg" "erge" "falc" "frat" "furc" "igno" "kech" "morm"
## [11] "neif" "ther" "tome"

morpho_split$conf

##      sphote sppar      para      a      b      c      d      f      g      aa
## 153   Sasa  conf Sasa1conf1 1.300 1.250 0.5500 0.3000 0.3500 0.2000 1.2000
## 188   Disa  conf Disa9conf1 1.204 1.244 0.7498 0.7443 0.6972 0.3702 0.9759
##      bb      cc      lm      li
## 153 1.1000 0.6500 1.300 1.300
## 188 0.9007 0.6713 1.173 1.144

```

On obtient 13 tableaux de données, un pour chaque espèce de parasites. On souhaite éliminer ceux qui ont été observés moins de trois fois au total. Ceci implique de parcourir chaque élément de la liste, et de déterminer sa taille. R propose une fonction `lapply`, littéralement `apply` sur une liste, pour effectuer cette tâche :

```

n_obs <- unlist(lapply(morpho_split, nrow))
n_obs >= 3

## conf dipl eleg erge falc frat furc igno kech morm neif ther
## FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE FALSE
## tome
## TRUE

```

Notons que `lapply` retourne une liste. On peut ensuite utiliser les informations sur le nombre d'observations, `n_obs`, pour choisir quels sous-tableaux garder :

```

morpho_split <- morpho_split[n_obs >= 3]

```

la encore, on remarquera que pour exclure certains éléments d'une liste, on utilise les crochets simples, comme pour un vecteur, et non les crochets doubles. On vérifie maintenant qu'il ne reste plus que des espèces avec plus de 3 observations :

```

unlist(lapply(morpho_split, nrow))

## eleg erge falc frat furc igno kech neif tome
## 59 19 9 6 7 43 30 6 3

```

On veut maintenant calculer la moyenne des éléments de chaque sous-tableau, en ne sélectionnant que les colonnes correspondant aux mesures morphométriques. Ces colonnes sont les 4 et suivantes, soit `c(4:ncol(x))` dans le langage de R, si on travaille sur un objet `x`. Une fois ces colonnes extraites, on peut vérifier qu'on obtient bien une matrice,

```
morpho_split$furc[, c(4:ncol(morpho_split$furc))]
```

```
##      a      b      c      d      f      g      aa      bb      cc      lm      li
## 23  2.030 1.930 1.250 0.5200 0.7300 0.3100 1.880 1.770 1.2000 2.350 2.760
## 140 2.100 1.850 1.500 0.6000 0.7000 0.4000 1.800 1.700 1.2000 1.900 2.700
## 141 2.000 1.900 1.300 0.6000 0.7000 0.3000 1.700 1.600 1.2000 1.800 2.800
## 142 1.900 1.850 1.450 0.6000 0.7500 0.4000 1.850 1.700 1.1000 1.950 2.600
## 175 1.905 1.843 1.305 0.6972 0.7975 0.4873 1.745 1.682 1.1482 2.212 2.621
## 176 1.813 1.691 1.240 0.6617 0.7786 0.5374 1.579 1.477 0.9888 1.693 2.187
## 187 2.031 1.939 1.420 0.7219 0.8022 0.4668 1.832 1.762 1.2685 2.007 2.680
```

, dont on peut calculer la moyenne sur chaque colonne par la fonction `apply`.

```
moy <- function(x) apply(x[, c(4:ncol(x))], 2, mean, na.rm = TRUE)
```

On peut maintenant appliquer cette fonction à nos données divisées en groupes :

```
lapply(morpho_split, moy)
```

```
## $eleg
##      a      b      c      d      f      g      aa      bb      cc      lm
## 1.7070 1.6101 1.1589 0.5191 0.5957 0.3475 1.5249 1.4336 0.9794 1.9543
##      li
## 2.2411
##
## $erge
##      a      b      c      d      f      g      aa      bb      cc      lm
## 2.0589 1.9603 1.3527 0.7425 0.9609 0.5216 1.7363 1.6743 0.9852 2.6796
##      li
## 2.5110
##
## $falc
##      a      b      c      d      f      g      aa      bb      cc      lm
## 1.4151 1.2760 0.6742 0.5014 0.4512 0.2660 1.1984 1.1344 0.7695 1.4033
##      li
## 1.3521
##
## $frat
##      a      b      c      d      f      g      aa      bb      cc      lm
## 1.6333 1.4667 1.0333 0.4833 0.6083 0.2917 1.3333 1.1583 0.9333 1.2000
##      li
## 1.5000
##
## $furc
##      a      b      c      d      f      g      aa      bb      cc      lm
## 1.9684 1.8575 1.3522 0.6287 0.7512 0.4145 1.7694 1.6700 1.1579 1.9873
```

```
##      li
## 2.6212
##
## $igno
##      a      b      c      d      f      g      aa      bb      cc      lm
## 1.1418 1.0682 0.6863 0.4650 0.5244 0.2613 1.0015 0.9159 0.5487 1.6268
##      li
## 1.4836
##
## $kech
##      a      b      c      d      f      g      aa      bb      cc      lm
## 1.8057 1.7346 1.1192 0.5808 0.8611 0.4013 1.5480 1.4304 0.7800 1.8590
##      li
## 1.8269
##
## $neif
##      a      b      c      d      f      g      aa      bb      cc      lm
## 1.0529 0.9435 0.6178 0.4035 0.4394 0.2495 0.8778 0.8019 0.4958 0.7910
##      li
## 1.0076
##
## $tome
##      a      b      c      d      f      g      aa      bb      cc      lm
## 2.2833 2.1833 1.2833 0.8000 1.1500 0.3833 1.8167 1.8333 0.9167 2.8167
##      li
## 3.0833
##
```

On peut aussi convertir facilement cette information en une `data.frame`, que l'on pivote pour avoir les noms des espèces en lignes :

```
t(as.data.frame(lapply(morpho_split, moy)))
```

```
##      a      b      c      d      f      g      aa      bb      cc      lm
## eleg 1.707 1.6101 1.1589 0.5191 0.5957 0.3475 1.5249 1.4336 0.9794 1.954
## erge 2.059 1.9603 1.3527 0.7425 0.9609 0.5216 1.7363 1.6743 0.9852 2.680
## falc 1.415 1.2760 0.6742 0.5014 0.4512 0.2660 1.1984 1.1344 0.7695 1.403
## frat 1.633 1.4667 1.0333 0.4833 0.6083 0.2917 1.3333 1.1583 0.9333 1.200
## furc 1.968 1.8575 1.3522 0.6287 0.7512 0.4145 1.7694 1.6700 1.1579 1.987
## igno 1.142 1.0682 0.6863 0.4650 0.5244 0.2613 1.0015 0.9159 0.5487 1.627
## kech 1.806 1.7346 1.1192 0.5808 0.8611 0.4013 1.5480 1.4304 0.7800 1.859
## neif 1.053 0.9435 0.6178 0.4035 0.4394 0.2495 0.8778 0.8019 0.4958 0.791
## tome 2.283 2.1833 1.2833 0.8000 1.1500 0.3833 1.8167 1.8333 0.9167 2.817
##      li
## eleg 2.241
## erge 2.511
```

```
## falc 1.352
## frat 1.500
## furc 2.621
## igno 1.484
## kech 1.827
## neif 1.008
## tome 3.083
```

### 3.3 Traitement des données

Il existe des moyens de rendre les étapes décrites dans la partie précédente automatique. Par exemple, la fonction `aggregate` permet d'aggréger les données en fonction de deux éléments : une combinaison de facteurs, et une fonction. On peut, en une ligne, connaître la moyenne de chacune des mesures, par hôte et par parasite, avec

```
aggregate(morpho, by = list(hote = morpho$sphot, parasite = morpho$sppar),
          mean, na.rm = TRUE)
```

##	hote	parasite	sphot	sppar	para	a	b	c	d	f
## 1	Disa	conf	NA	NA	NA	1.2035	1.244	0.7498	0.7443	0.6972
## 2	Sasa	conf	NA	NA	NA	1.3000	1.250	0.5500	0.3000	0.3500
## 3	Divu	dipl	NA	NA	NA	1.1000	1.025	0.5250	0.4500	0.4750
## 4	Dian	eleg	NA	NA	NA	1.6500	1.450	0.9750	0.6000	0.6500
## 5	Disa	eleg	NA	NA	NA	1.7402	1.662	1.1857	0.5340	0.6263
## 6	Divu	eleg	NA	NA	NA	1.7600	1.637	1.2436	0.5113	0.5687
## 7	Obme	eleg	NA	NA	NA	1.3917	1.292	0.8667	0.4250	0.4667
## 8	Dipu	erge	NA	NA	NA	2.3865	2.314	1.6667	0.8382	1.1478
## 9	Disa	erge	NA	NA	NA	1.9312	1.828	1.2777	0.6802	0.9088
## 10	Divu	erge	NA	NA	NA	2.1700	2.045	1.1750	0.9250	0.9000
## 11	Disa	falc	NA	NA	NA	1.4428	0.994	0.5997	0.4850	0.4599
## 12	Divu	falc	NA	NA	NA	1.4071	1.370	0.7040	0.5080	0.4483
## 13	Dian	frat	NA	NA	NA	1.6333	1.467	1.0333	0.4833	0.6083
## 14	Disa	furc	NA	NA	NA	1.9684	1.857	1.3522	0.6287	0.7512
## 15	Dipu	igno	NA	NA	NA	1.2776	1.274	0.8402	0.6224	0.6856
## 16	Disa	igno	NA	NA	NA	1.1604	1.117	0.7640	0.4677	0.5664
## 17	Divu	igno	NA	NA	NA	1.2288	1.156	0.7486	0.4943	0.5443
## 18	Limo	igno	NA	NA	NA	1.0200	0.910	0.5050	0.4350	0.4600
## 19	Sasa	igno	NA	NA	NA	1.1250	1.012	0.6625	0.4000	0.4125
## 20	Disa	kech	NA	NA	NA	1.7886	1.833	1.2050	0.6117	1.0317
## 21	Divu	kech	NA	NA	NA	1.8109	1.708	1.0921	0.5706	0.8145
## 22	Disa	morm	NA	NA	NA	1.6000	1.550	1.2500	0.7000	0.8000
## 23	Disa	neif	NA	NA	NA	0.8625	0.750	0.4500	0.3250	0.3500
## 24	Divu	neif	NA	NA	NA	1.8146	1.718	1.2890	0.7173	0.7970
## 25	Dipu	ther	NA	NA	NA	2.3652	2.218	1.5876	0.8393	1.0874

```
## 26 Divu      tome      NA      NA      NA 2.2833 2.183 1.2833 0.8000 1.1500
##           g      aa      bb      cc      lm      li
## 1  0.3702 0.9759 0.9007 0.6713 1.1728 1.144
## 2  0.2000 1.2000 1.1000 0.6500 1.3000 1.300
## 3  0.2250 1.2000 1.0750 0.7000 1.2000 1.575
## 4  0.3000 1.5750 1.4000 0.8250 1.5000 2.050
## 5  0.3587 1.5289 1.4760 0.9987 2.0067 2.342
## 6  0.3307 1.6381 1.5027 1.0667 2.2046 2.272
## 7  0.3417 1.1833 1.0250 0.7000 1.2667 1.530
## 8  0.5814 1.9896 1.8985 1.1497 2.8011 2.639
## 9  0.4777 1.6516 1.5806 0.9307 2.6667 2.406
## 10 0.6650 1.7800 1.8350 1.0100 2.5200 2.940
## 11 0.2461 0.9029 0.8876 0.5781 1.5615 1.297
## 12 0.2740 1.2829 1.2167 0.8333 1.3400 1.374
## 13 0.2917 1.3333 1.1583 0.9333 1.2000 1.500
## 14 0.4145 1.7694 1.6700 1.1579 1.9873 2.621
## 15 0.3492 1.1818 1.1463 0.7165 1.7888 1.774
## 16 0.2548 1.0110 0.9432 0.5483 1.6923 1.497
## 17 0.3086 1.0137 0.9188 0.6050 1.6013 1.550
## 18 0.2400 0.9150 0.7900 0.4700 1.5450 1.410
## 19 0.2125 1.0625 1.0000 0.5500 1.4667 1.363
## 20 0.4483 1.4686 1.5183 0.7950 2.0800 1.893
## 21 0.3847 1.5722 1.4064 0.7759 1.7886 1.806
## 22 0.6500 1.4000 1.3000 0.8000 2.2000 2.000
## 23 0.2000 0.7500 0.6625 0.3750 0.5125 0.800
## 24 0.4476 1.3888 1.3594 0.9788 1.9052 1.838
## 25 0.5883 1.9996 1.9543 1.1350 2.8016 2.812
## 26 0.3833 1.8167 1.8333 0.9167 2.8167 3.083
```

Cette ligne signifie, en clair, pour chaque niveau de `sphot` et pour chaque niveau de `sppar`, calculer la moyenne de toutes les colonnes de `morpho`. On peut récupérer la moyenne de `aa` uniquement, avec

```
aggregate(morpho$aa, by = list(hote = morpho$sphot, parasite = morpho$sppar),
          mean)
```

```
##      hote parasite      x
## 1  Disa      conf 0.9759
## 2  Sasa      conf 1.2000
## 3  Divu     dipl 1.2000
## 4  Dian     eleg 1.5750
## 5  Disa     eleg 1.5289
## 6  Divu     eleg 1.6381
## 7  Obme     eleg 1.1833
## 8  Dipu     erge 1.9896
## 9  Disa     erge 1.6516
## 10 Divu     erge 1.7800
```

```
## 11 Disa      falc 0.9029
## 12 Divu      falc 1.2829
## 13 Dian      frat 1.3333
## 14 Disa      furc 1.7694
## 15 Dipu      igno 1.1818
## 16 Disa      igno      NA
## 17 Divu      igno 1.0137
## 18 Limo      igno 0.9150
## 19 Sasa      igno      NA
## 20 Disa      kech 1.4686
## 21 Divu      kech 1.5722
## 22 Disa      morm 1.4000
## 23 Disa      neif      NA
## 24 Divu      neif 1.3888
## 25 Dipu      ther 1.9996
## 26 Divu      tome 1.8167
```

### 3.4 Sélection d'un sous-ensemble

R permet de facilement sélectionner un sous-ensemble des données sur lequel travailler, avec la fonction `subset`. Cette fonction sélectionne une partie des lignes d'une *data frame* selon une série de conditions. Par exemple, dans les données *morpho*, on veut uniquement travailler sur les mesures de *L. elegans* (`sppar == 'eleg'`) trouvés sur *D. vulgaris* (`sppar == 'divu'`), pour lesquelles la valeur *cc* est inférieure ou égale à 1 (`cc <= 1`).

```
E1DvSmall <- subset(morpho, (morpho$sppar == "eleg") & (morpho$sphote ==
  "Divu") & (morpho$cc <= 1))
E1DvSmall
```

```
##      sphote sppar      para      a      b      c      d      f      g      aa      bb      cc
## 3      Divu  eleg  elegDivu2 1.67 1.56 1.20 0.31 0.36 0.26 1.56 1.51 0.94
## 59      Divu  eleg  Divu1eleg1 1.55 1.40 1.20 0.50 0.50 0.30 1.60 1.60 0.90
## 69      Divu  eleg  Divu1eleg3 1.70 1.60 1.20 0.50 0.60 0.35 1.60 1.45 1.00
## 72      Divu  eleg  Divu1eleg5 1.65 1.30 1.10 0.55 0.55 0.35 1.50 1.40 0.90
## 73      Divu  eleg  Divu1eleg6 1.70 1.80 1.10 0.65 0.65 0.25 1.55 1.40 1.00
## 85      Divu  eleg  Divu2eleg3 1.85 1.50 1.25 0.50 0.55 0.40 1.65 1.55 1.00
## 97      Divu  eleg  Divu7eleg1 1.80 1.65 1.20 0.55 0.55 0.40 1.60 1.50 1.00
## 132      Divu  eleg  Divu9eleg1 1.75 1.70 1.25 0.50 0.65 0.35 1.65 1.50 1.00
##      lm      li
## 3      1.88 1.77
## 59      NA      NA
## 69      2.10 2.40
## 72      2.10 2.40
## 73      1.50 2.10
```

```
## 85    NA    NA
## 97  1.90  2.30
## 132 1.90  2.10
```

Pour récupérer uniquement les valeurs de  $a$ , on dispose des deux options suivantes :

```
EldvSmall$a
```

```
## [1] 1.67 1.55 1.70 1.65 1.70 1.85 1.80 1.75
```

```
subset(morpho$a, (morpho$sppar == "eleg") & (morpho$sphote == "Divu") &
        (morpho$cc <= 1))
```

```
## [1] 1.67 1.55 1.70 1.65 1.70 1.85 1.80 1.75
```

### 3.5 Tables de contingence

R permet très facilement de regrouper les données sous forme de table. Par exemple, on veut représenter la moyenne des mesures de la dimension  $a$  :

```
moyA <- aggregate(morpho, by = list(hote = morpho$sphot, parasite = morpho$sppar),
                  mean, na.rm = TRUE)
round(xtabs(a ~ hote + parasite, moyA), 1)
```

```
##      parasite
## hote  conf dipl eleg erge falc frat furc igno kech morm neif ther tome
## Dian  0.0  0.0  1.6  0.0  0.0  1.6  0.0  0.0  0.0  0.0  0.0  0.0  0.0
## Dipu  0.0  0.0  0.0  2.4  0.0  0.0  0.0  1.3  0.0  0.0  0.0  2.4  0.0
## Disa  1.2  0.0  1.7  1.9  1.4  0.0  2.0  1.2  1.8  1.6  0.9  0.0  0.0
## Divu  0.0  1.1  1.8  2.2  1.4  0.0  0.0  1.2  1.8  0.0  1.8  0.0  2.3
## Limo  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
## Obme  0.0  0.0  1.4  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
## Sasa  1.3  0.0  0.0  0.0  0.0  0.0  0.0  1.1  0.0  0.0  0.0  0.0  0.0
```

La fonction `xtabs` permet de prendre une colonne du *data frame*, et de créer un tableau à double entrées avec l'ensemble de ses valeurs. On peut s'en servir pour créer une table de contingence, en omettant de dire quelle valeur doit se trouver dans le tableau :

```
Conting <- xtabs(~sphote + sppar, morpho)
Conting
```

```
##      sppar
## sphote conf dipl eleg erge falc frat furc igno kech morm neif ther tome
## Dian    0    0    2    0    0    6    0    0    0    0    0    0    0
## Dipu    0    0    0    4    0    0    0    2    0    0    0    2    0
```



```
## Disa 1 0 35 13 2 0 7 18 7 1 5 0 0
## Divu 0 2 16 2 7 0 0 8 23 0 1 0 3
## Limo 0 0 0 0 0 0 0 10 0 0 0 0 0
## Obme 0 0 6 0 0 0 0 0 0 0 0 0 0
## Sasa 1 0 0 0 0 0 0 5 0 0 0 0 0
```

```
sum(Conting)
```

```
## [1] 189
```

Si aucune colonne n'est spécifiée, R fournit le nombre d'observatins. On peut vérifier que le total est de 189, on dispose donc, en une commande, du nombre d'échantillons pris pour chaque condition. Notez aussi que les objets retournés par `xtabs` peuvent aller directement dans un certain nombre de tests statistiques, notamment

```
chisq.test(Conting)
```

```
##
## Pearson's Chi-squared test
##
## data: Conting
## X-squared = 335.7, df = 72, p-value < 2.2e-16
##
```

### 3.6 Jonctions de tables de données

On souhaite regrouper dans un `data.frame` unique deux séries de données disponibles dans ANDREWS et HERZBERG [4] : le nombre de peaux de Lynx enregistrées par la compagnie de la Baie d'Hudson de 1857 à 1911 (`data/lynxunits.dat`), et le prix de ces peaux (`data/lynxprices.dat`). Il y a plusieurs problèmes avec ces données. D'une part, les années ne sont pas entrées dans le même ordre, et il n'est donc pas possible de faire un `cbind`. D'autre part, la colonne correspondant aux années est nommée `year` dans un cas, et `Year` dans l'autre.

R propose une fonction nommée `merge`, qui permet d'aggréger très facilement des données. On commence par les charger :

```
units <- read.table("data/lynxunits.dat", h = T)
prices <- read.table("data/lynxprices.dat", h = T)
head(units)
```

```
## year units
## 1 1871 15686
## 2 1881 15386
## 3 1865 5437
## 4 1909 9704
## 5 1870 37447
```

```
## 6 1910 3410
```

```
head(prices)
```

```
##   Year prices
## 1 1890 13.75
## 2 1863 12.25
## 3 1906 26.83
## 4 1859  9.50
## 5 1894 11.50
## 6 1878  7.42
```

La commande `merge` demande les deux tables de données, et deux arguments `by.x` et `by.y`, qui indique quelles colonnes doivent être utilisées pour le regroupement. Notez au passage que si les deux tables de données ont des colonnes avec le même nom, par défaut, R réunira les données selon ces colonnes.

```
dataset <- merge(units, prices, by.x = "year", by.y = "Year")
head(dataset)
```

```
##   year units prices
## 1 1857 23362 12.75
## 2 1858 31642  8.08
## 3 1859 33757  9.50
## 4 1860 23226 10.00
## 5 1861 15178  8.08
## 6 1862  7272  8.50
```

Notez au passage que la fonction `merge` va trier automatiquement les colonnes sur lesquelles elle opère le regroupement : les années sont maintenant en ordre. Avec ce qui a été vu dans la séance précédente, on peut écrire les données dans un fichier texte :

```
write.table(dataset, file = "data/lynxdata.txt", row.names = FALSE)
```

## Introduction à la programmation

Dans les séances précédentes, nous avons utilisé des fichiers `.R` pour sauvegarder des listes d'instructions. Nous avons aussi chargé et manipulé des jeux de données. Il est souvent nécessaire d'automatiser tout ou partie de ce processus, ce qui implique de faire appel à de la programmation.

L'objectif de cette séance est de se familiariser avec les principaux concepts qu'on utilise pour concevoir un programme. La première partie couvre les bases en algorithmie, c.-à-d. les boucles et les tests. La deuxième partie concerne les fonctions, leur définition dans R, et leur utilisation. À l'issue de cette séance, vous serez en mesure de vous attaquer à pratiquement tous les problèmes nécessitant de manipuler des données. Tout programme que vous aurez à écrire ne sera qu'une combinaison plus ou moins complexe des éléments abordés dans les séances précédentes.

### 4.1 Boucles

Les boucles permettent parcourir une liste, ou de répéter une série d'instructions, dans des conditions bien définies ; c'est une des structures de base de l'algorithmique. R propose deux types de boucles, les boucles `for` et les boucles `while`. En français, on peut les résumer par «pour chaque» et «tant que».

#### Boucles de type *for*

Une boucle `for` permet de répéter un bloc d'instructions un nombre prédéfini de fois, ou d'exécuter des commandes pour chaque élément d'un tableau de données. La syntaxe de base est la suivante :

```
for (step in c(1:10)) cat(step)
```

```
## 12345678910
```

En clair, pour chaque valeur entre 1 et 10, qu'on nomme `step` (mais qu'on aurait pu nommer n'importe comment), on affiche (`cat`) la valeur de `step`. On peut bien sûr spécifier plusieurs instructions qui doivent être exécutées à chaque *itération* (étapes de la boucle) en utilisant les accolades :

```
for (step in c(1:3)) {
  cat(step)
  print(summary(rnorm(100, mean = step)))
}

## 1   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -0.932  0.128   0.902   0.886   1.690   2.870
## 2   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.325  1.460   1.980   2.030   2.580   4.440
## 3   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -0.034  2.150   2.950   2.870   3.610   5.750
```

Cette commande affiche le numéro de l'itération en cours (`cat(step)`), puis affiche les informations de base (`summary`) sur une distribution normale (`rnorm`) centrée sur `step`. Les boucles `for` peuvent contenir des instructions aussi longues que souhaité.

Une autre application des boucles `for` est de parcourir un objet. Par exemple, on peut souhaiter, pour chaque élément d'un objet, afficher sa valeur. R permet de réaliser ce genre d'opérations, avec la syntaxe suivante :

```
vect <- c("a", "b", "c", "d")
for (val in vect) cat(val)

## abcd
```

Pour chaque élément du vecteur `vect`, que l'on nomme `val` pour pouvoir y accéder pendant les itérations, R va afficher la valeur que l'élément contient.

Les boucles `for` sont donc une façon explicite de faire la même chose que ce qui a été traité dans la séance précédente, avec un inconvénient majeur : elles sont extrêmement demandantes en temps de calcul. Dans la majorité des cas, il est préférable d'avoir recours autant que possible aux fonctions `*apply`, qui sont bien plus optimisées que des boucles.

## Boucles de type *while*

Les boucles de type `while`, littéralement *pendant*, permettent de répéter une série d'instructions tant qu'une condition n'a pas été atteinte. Pour cette raison, il faut bien prendre en compte le fait que mal utilisées, ces boucles peuvent ne jamais stopper. Il faut donc faire particulièrement attention à la condition qui est évaluée à chaque itération.

Un exemple simple d'utilisation d'une boucle `while` est le calcul d'une factorielle. On veut calculer  $n!$ , ce qui se fait simplement en multipliant l'ensemble des  $1 \leq k \leq n$ .

```
n <- 5
k <- n
while (k > 1) {
  k <- k - 1
  n <- n * k
}
```

```
print(n)
```

```
## [1] 120
```

On remarquera que dans la parenthèse après `while` se trouve un test logique ; les tests sont abordés dans la partie suivante.

### Sortir d'une boucle et sauter des étapes

Lors de l'exécution d'une boucle, on peut ne pas vouloir exécuter toutes les instructions. R possède des structures de contrôle pour effectuer ce type d'opérations. Les principales qu'il faut connaître sont `break` et `next`. L'instruction `break` permet de stopper l'exécution de la boucle, c'est-à-dire de sortir de la boucle comme si la condition de sortie était remplie, ou le nombre maximal d'itérations atteint. La boucle suivante va afficher les valeurs de `i` tant qu'elles sont inférieures à 3, et afficher `bye!` puis stopper la boucle sinon.

```
for (i in c(1:5)) {  
  if (i < 3) {  
    cat(i)  
  } else {  
    cat(" bye!")  
    break  
  }  
}
```

```
## 12 bye!
```

Une autre structure de contrôle intéressant est `next`, qui permet de sauter une itération si la valeur de l'itérateur ne nous plaît pas. Cette structure est particulièrement utile quand on réalise un nombre important d'opérations à chaque itération, et qu'on ne veut pas perdre de temps à traiter des valeurs qui ne nous intéressent pas. On peut utiliser `next` pour avoir, par exemple, une liste de tous les nombres pairs entre 1 et 10 :

```
is.even <- function(x) x%%2 == 0  
for (i in c(1:10)) {  
  if (is.even(i)) {  
    cat(i)  
    cat(" is even\n")  
  } else {  
    next  
  }  
}
```

```
## 2 is even  
## 4 is even  
## 6 is even
```

```
## 8 is even
## 10 is even
```

## 4.2 Tests

Cette partie est consacrée à l'utilisation des tests. On a vu dans les séances précédentes l'existence de variables de type booléen, qui prennent les valeurs TRUE ou FALSE. Des variables de ce type sont utilisées dans le cadre d'expressions conditionnelles, c'est-à-dire quand on souhaite effectuer différentes instructions en fonction de la valeur d'une condition.

### Expressions conditionnelles

La structure de base d'une expression conditionnelle est la suivante :

```
if (condition) {
  instruction(1)
} else {
  instruction(2)
}
```

Une notation comme

```
if (condition) instruction
```

est aussi acceptable, de même que

```
val <- ifelse(condition, valeur_if, valeur_else)
```

Cette dernière notation permet de gagner du temps quand on veut que la valeur d'une variable dépende d'une condition. Par exemple,  $(n \geq 2) \vee (n = 0)$  s'écrit  $(n \leq 2) \text{ or } (n == 0)$ , ou encore  $(n \leq 2) \mid (n == 0)$ . Les différents opérateurs logiques sont regroupés dans le tableau 4.1.

L'argument condition prend la forme d'un test logique, qui peut être d'une complexité aussi grande que l'on veut. Pour certains des opérateurs, il existe deux variantes ( $\mid$  et  $\mid\mid$ ,  $\&$  et  $\&\&$ ). On peut comprendre pourquoi avec les exemples suivants :

```
a <- c(1, 2, 3, 4, 5, 6)
b <- c(1, 2, 3, 4, 4, 6)
d <- c(3, 2, 3, 4, 5, 4)
(a == b)

## [1] TRUE TRUE TRUE TRUE FALSE TRUE

(a == d)

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

Opérateur	Signification	Version 1	Version 2
$\vee$	ou (au moins une des deux)	or	,
$\wedge$	et (les deux)	and	&, &&
$\neg$	non	not	!
$\oplus$	ou conditionnel (seulement une des deux)	xor(a, b)	
$\in$	est compris dans	%in%	
=	égalité		==
$\leq$	inférieur ou égal		<=
$\geq$	supérieur ou égal		>=
>	supérieur		>
<	inférieur		<

TABLE 4.1: Différents opérateurs logiques disponibles dans R.

```

(b == d)

## [1] FALSE TRUE TRUE TRUE FALSE FALSE

(a == d) & (a == b)

## [1] FALSE TRUE TRUE TRUE FALSE FALSE

(a == d) && (a == b)

## [1] FALSE

(a == d) | (a == b)

## [1] TRUE TRUE TRUE TRUE TRUE TRUE

(a == d) || (a == b)

## [1] TRUE

all.equal(a, b)

## [1] "Mean relative difference: 0.2"

```

L'opérateur répété une seule fois travaille par élément : les éléments des vecteurs sont comparés deux à deux (et on applique du recyclage). L'opérateur répété deux fois ne prend en compte que les premières valeurs des vecteurs (un message d'avis sera émis si le vecteur a une taille supérieure à 1).

## Manipulation des valeurs booléennes

Comme tous les autres types de données de R, les valeurs booléennes peuvent être manipulées dans des additions.

```
TRUE + TRUE
```

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

```
FALSE + TRUE
```

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

```
TRUE * TRUE
```

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

Cette possibilité de travailler sur les valeurs booléennes comme si  $\text{TRUE} = 1$  et  $\text{FALSE} = 0$  permet des raccourcis intéressants. Par exemple, on peut facilement connaître le nombre d'éléments d'un vecteur qui répondent à une condition particulière, ici  $x \in (-0.5; 1]$  :

```
a <- rnorm(100)
sum(a[(a < 1) & (a >= 0.5)])
```

```
## [1] 14.37
```

```
sum((a < 1) & (a >= 0.5))
```

```
## [1] 20
```

Dans l'exemple précédent, la première ligne fait la somme de  $a$  aux indices satisfaisant la condition fixée. La deuxième ligne fait la somme de ces indices, c.-à-d. d'un vecteur de booléens.

## 4.3 Fonctions

### Généralités

L'utilisation des fonctions va permettre de gagner du temps dans la programmation. Comprendre le principe des fonctions dépasse de beaucoup le cadre de R, et mérite qu'on s'y arrête. Qu'est-ce qu'une fonction ? Une série d'instructions qui vont, à partir d'arguments, renvoyer un résultat. En quoi est-ce différent des scripts que nous avons utilisé jusqu'ici ? Écrire une fonction revient en quelque sorte à 'expliquer' le code une fois, et R se charge ensuite de redonner la bonne valeur aux arguments.

Le parallèle le plus évident est celui des fonctions mathématiques : si  $f(x) = x^2$ , on peut calculer  $f(x)$  pour tout  $x$ , parce qu'on sait quoi faire. Ça devient donc très avantageux si on doit calculer  $f(x)$  un grand nombre de fois. En écrivant uniquement des scripts, pour calculer la valeur de beaucoup de  $x^2$ , on aurait écrit :



```
1^2
2^2
3^2
4^2
5^2
```

ou encore

```
for (i in c(1:5)) i^2
```

Si il faut revenir sur ce code plus tard, et transformer tous les  $^2$  en  $^3$ , la première solution implique de tout corriger manuellement. En utilisant une fonction, la logique est différente :

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

## [1] 4
```

Peut importe le nombre de fois ou on devra effectuer l'opération contenue dans `f`, si on veut la modifier, elle sera toujours stockée au même endroit. Dans la pratique, la majorité des instructions utilisées jusqu'à présent sont des fonctions. Par exemple, `print`, `read.table`, et `apply` sont des fonctions rendues disponibles par R. Écrire une instruction comme `read.table('file.txt', h=TRUE)`, c'est faire appel à une fonction, en lui spécifiant certains *arguments*. L'objectif des parties suivantes est de comprendre comment les fonctions fonctionnent (!), afin de pouvoir en écrire de nouvelles.

Les fonctions existent comme un espace «à part» dans R : ce qui se déroule dans une fonction, reste dans une fonction. Prenons le cas du code suivant :

```
a <- 2
b <- 3
f <- function(x) {
  y <- x + a
  z <- y
  return(z)
}
f(b)

## [1] 5
```

La fonction `f` peut aller chercher la valeur de `a` dans l'environnement global, mais tout ce qui est défini au sein de `f` est inaccessible. D'ailleurs, tout ce qui est créé dans la fonction est détruit – retiré de la mémoire – une fois que la dernière instruction est exécutée. Voyez la section sur la fonction `return` pour plus de détails. Cette notion de quel objet est accessible est extrêmement importante à maîtriser.

Dans R, les objets existent dans deux «mondes», l'environnement global, et l'intérieur de chaque fonction. L'intérieur de chaque fonction peut avoir accès aux objets et aux variables de l'environnement global, même si cette pratique est à éviter pour différentes raisons (une variable globale peut

être modifiée entre deux appels à la fonction, notamment). En revanche, l'environnement global n'a pas accès aux objets créés ou modifiés à l'intérieur d'une fonction. La communication avec les fonctions se fait ...

Le fait qu'on puisse passer des objets d'un environnement à l'autre peut entraîner un comportement assez surprenant. Dans l'exemple suivant :

```
a <- 2
print(a)

## [1] 2

f <- function(a) {
  a <- a + 1
  return(a)
}
print(f(a))

## [1] 3

print(a)

## [1] 2
```

L'appel à la fonction `f` devrait modifier `a`, puisque la seule instruction de cette fonction est `a = a+1`. Or, quand on appelle cette fonction, puis qu'on affiche la valeur de `a`, elle n'a pas changé. Ce comportement vient du fait que l'objet `a` qui existe dans la fonction n'est pas celui qui existe dans l'environnement global. Par conséquent, le `a` de la fonction peut être modifié, sans que cela n'affecte le `a` de l'environnement de travail.

Cet exemple illustre aussi pourquoi le nom des variables est important. On sait que R est capable d'aller chercher, depuis une fonction, des variables de l'environnement global. Quelle version de `a` faut-il aller chercher ? Pour éviter les erreurs liées au fait que plusieurs variables aient le même nom, on essaie de donner un nom unique à toutes les variables. C'est sans doute plus long à écrire – même si cet argument n'est pas valable avec un éditeur qui auto-complète le code –, mais ça évite surtout les erreurs à l'exécution.

## Déclarer une fonction

Comme illustré dans les exemples précédents, la déclaration d'une fonction se fait par

```
nom_de_la_fonction <- function(argument1, argument2) {
  instructions
  return(sortie)
}
```

Les éléments les plus importants sont les arguments et l'instruction `return`.

### La commande *return*

La commande `return` est en général la dernière ligne d'une fonction : ce qui se passe après est ignoré. Cette commande va renvoyer ce qui se trouve entre ses parenthèses dans l'environnement global de R : c'est un des points de communication entre l'environnement global et l'environnement de la fonction. La commande `return` ne peut prendre qu'un seul argument, il faut donc regrouper les variables sous forme de liste ou autres si vous en avez plusieurs.

Prenons un exemple simple :

```
somme_1 <- function(a, b) {  
  S <- a + b  
}  
somme_2 <- function(a, b) {  
  S <- a + b  
  return(S)  
}  
  
a <- somme_1(2, 3)  
a  
  
## [1] 5  
  
b <- somme_2(2, 3)  
b  
  
## [1] 5
```

Quelques précisions. On peut tout à fait nommer des variables `a` et `b` dans l'environnement global, même si les fonctions `somme_1` et `somme_2` utilisent des variables avec ces noms, grâce au fait que le *scope* de ces variables n'est pas le même. Il est toutefois déconseillé de le faire. Dans l'exemple précédent, la variable `S` n'existe pas, à aucun moment, dans l'environnement global de R. Le seul moyen de récupérer ce qui a été renvoyé par les fonctions est de les assigner à une variable de l'environnement global.

Lorsque que la fonction ne se compose que d'une seule ligne, le résultat de cette dernière ligne est renvoyé et il n'y a pas besoin de spécifier `return` :

```
multipl <- function(a, b) a * b  
multipl(2, 3)  
  
## [1] 6
```

### Arguments

Les arguments sont des variables que l'on peut échanger entre R et l'intérieur d'une fonction. Dans l'exemple précédent, pour faire une somme, il faut additionner deux nombres :  $f(a, b) = a + b$  s'écrit `f = function(a, b) a + b`. Les éléments entre parenthèse sont les arguments de la fonction.

Typiquement, un argument est noté par `nom = valeur`, où `nom` est l'identifiant de cet argument *dans la fonction*, et `valeur` est la valeur par défaut.

Les exemples suivants permettent de comprendre le fonctionnement des arguments et des valeurs par défaut.

```
somme <- function(a = 1, b = 1) a + b
somme()

## [1] 2

somme(a = 2)

## [1] 3

somme(b = 3)

## [1] 4

somme(1, 2)

## [1] 3
```

Il existe un argument particulièrement intéressant dans R : `...`. Cet argument contient tout objet passé à une fonction, pour lequel il n'y pas pas de nom d'argument correspondant. Par exemple :

```
print_message <- function(msg = "Hello", ...) {
  print(msg)
  print(...)
}
print_message(msg = "Hello", "world!")

## [1] "Hello"
## [1] "world!"
```

L'utilisation de cet argument *catch all* demande un peu de réflexion et beaucoup de pratique, mais s'avère particulièrement utile quand le format exact des arguments qui seront donnés à une fonction n'est pas connu d'avance.

## 4.4 Mise en application

### Test par permutation

Lorsque les données deviennent de la normalité, on peut préférer réaliser un test paramétrétique avec des permutations plutôt qu'un test non paramétrique. La plupart des programmes de statistique n'offrent pas cette possibilité qui demande pourtant très peu d'efforts pour être implémentée dans R. Dans cette mise en application, on veut effectuer un test t, pour comparer deux distributions, disponibles dans un fichier `s4-data.txt`.

Le principe d'un test par permutations est simple. La première étape est d'effectuer le test sur l'échantillon non permuté, pour obtenir la valeur de la statistique ( $T$ ). Dans le cas du test  $t$ , R propose la fonction `t.test`, et un rapide survol de `?t.test` vous donnera les arguments nécessaires et la manière de récupérer la statistique. Commencent ensuite les permutations à proprement parler. Pour un nombre  $n$  d'itérations choisies (en général 9999), on mélange l'ensemble des valeurs des deux distributions. On reconstruit ensuite, en tirant au hasard dans le *pool* de valeurs ainsi formées, deux distributions de taille égale. Cette étape peut, par exemple, prendre la forme d'une fonction `resample`, qui prendrait une `data.frame` avec deux colonnes (la valeur, et le groupe d'origine) en argument. R propose la fonction `sample`, qui permettra de mélanger la colonne correspondant au groupe (ce qui recréera automatiquement les deux distributions – économisons nous !). Une fois les deux distributions reconstruites, on calcule la nouvelle statistique  $T'$ . Si la valeur de  $T'$  est inférieure ou égale à la valeur de  $T$ , on incrémente une variable  $N$  de 1. Sinon, la valeur de  $N$  reste la même.

Le calcul de la *p-value* se fait de la manière suivante :

$$p = \frac{N}{n + 1} \quad (4.1)$$

À partir de ces informations, et des informations données dans l'introduction de cette séance, vous devez être en mesure de programmer sans difficultés une fonction `t.test.permut`, qui permet de réaliser un test  $t$  par permutations. En bonus, vous pouvez ajouter des arguments qui permettent de contrôler le nombre de permutations qui doivent être réalisées.

## 4.5 Solution des mises en application

### Test par permutation

On commence par écrire une fonction `resample`, qui mélange l'attribution des valeurs à un des deux groupes.

```
resample <- function(df) {
  df$group <- sample(df$group)
  return(df)
}
```

Puis on écrit la fonction `t.test.permut`, qui effectue les permutations à proprement parler

```
t.test.permut <- function(df, n = 9999) {
  baseStat <- t.test(value ~ group, df)$statistic
  N <- 0
  for (repl in c(1:n)) if (t.test(value ~ group, resample(df))$statistic <=
    baseStat)
    N <- N + 1
  return(N/(n + 1))
}
```

On peut générer un jeu de données de test :

```
value <- c(rnorm(100, mean = 0), rnorm(100, mean = 1))
group <- c(rep("a", 100), rep("b", 100))
test_df <- as.data.frame(cbind(value = value, group = group))
test_df$value <- as.numeric(as.vector(test_df$value))
```

puis vérifier que la *p-value* est inférieure à 0.05 :

```
print(t.test.permut(test_df, n = 9))

## [1] 0
```

Pour éviter les problèmes de lenteur des boucles `for`, on peut écrire la même fonction basée sur la fonction `replicate` :

```
t.test.permut <- function(df, n = 9999) {
  baseStat <- t.test(value ~ group, df)$statistic
  return(sum(replicate(n, t.test(value ~ group, resample(df))$statistic <=
    baseStat))/(n + 1))
}
```

La différence sur 10000 réplicats est assez minime (3 secondes environ). Mais dans un contexte où il faut analyser plusieurs fois des jeux de données, ces petits écarts finissent par faire une différence importante.

## Graphiques

R propose un moteur graphique d'une très grande puissance, qui possède en plus une flexibilité importante. Il est possible, avec un effort minimal, de produire à peu près tous les types de visualisations possibles directement dans R. L'objectif de cette séance est de vous familiariser avec les commandes de base disponibles dans R par défaut. Les personnes à la recherche de solutions toutes faites pour visualiser des données complexes peuvent aller voir la documentation des *packages* *ggplot2* (intuitif à utiliser, assez lent) ou *lattice* (utilisation complexe, assez rapide).

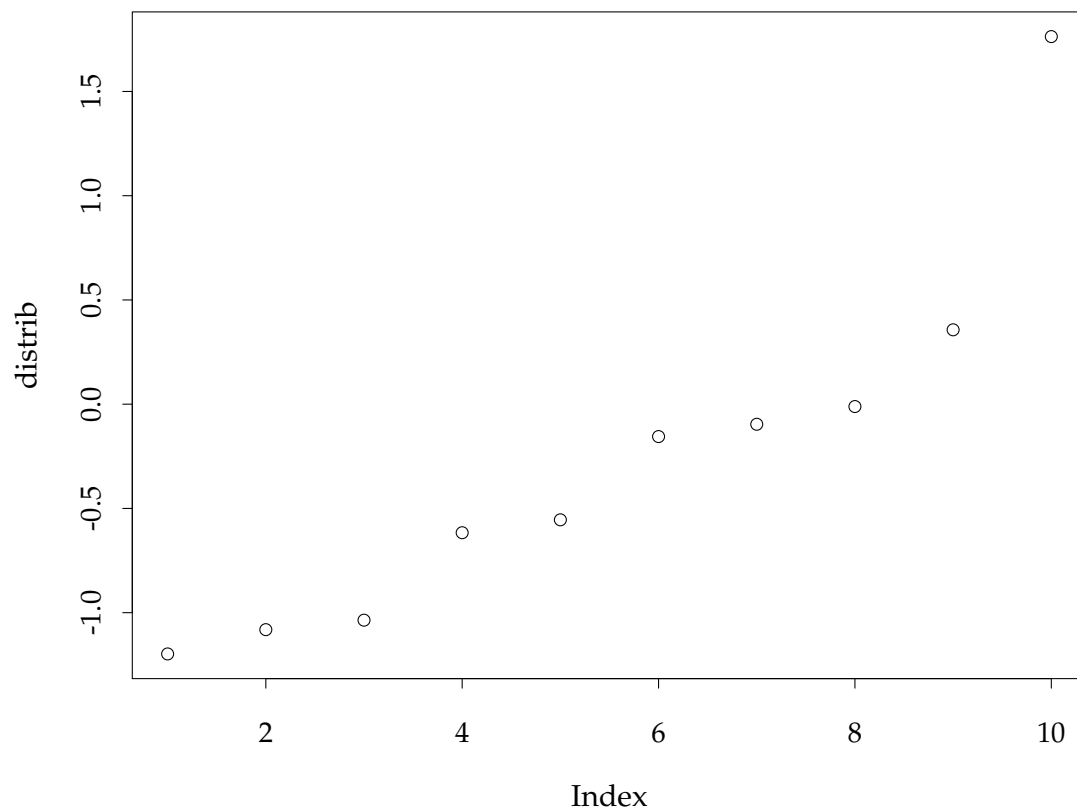
Les ouvrages de référence pour la visualisation des données sont, en ce qui concerne R, sont le livre de MURRELL [5], et de manière plus générale, les travaux d'Edward Tufte, dont un aperçu est donné par IVERSEN [6].

### 5.1 Principaux types de visualisations

#### Nuages de points

La manière la plus simple de représenter un objet dans R est d'utiliser la fonction `plot`. Par exemple,

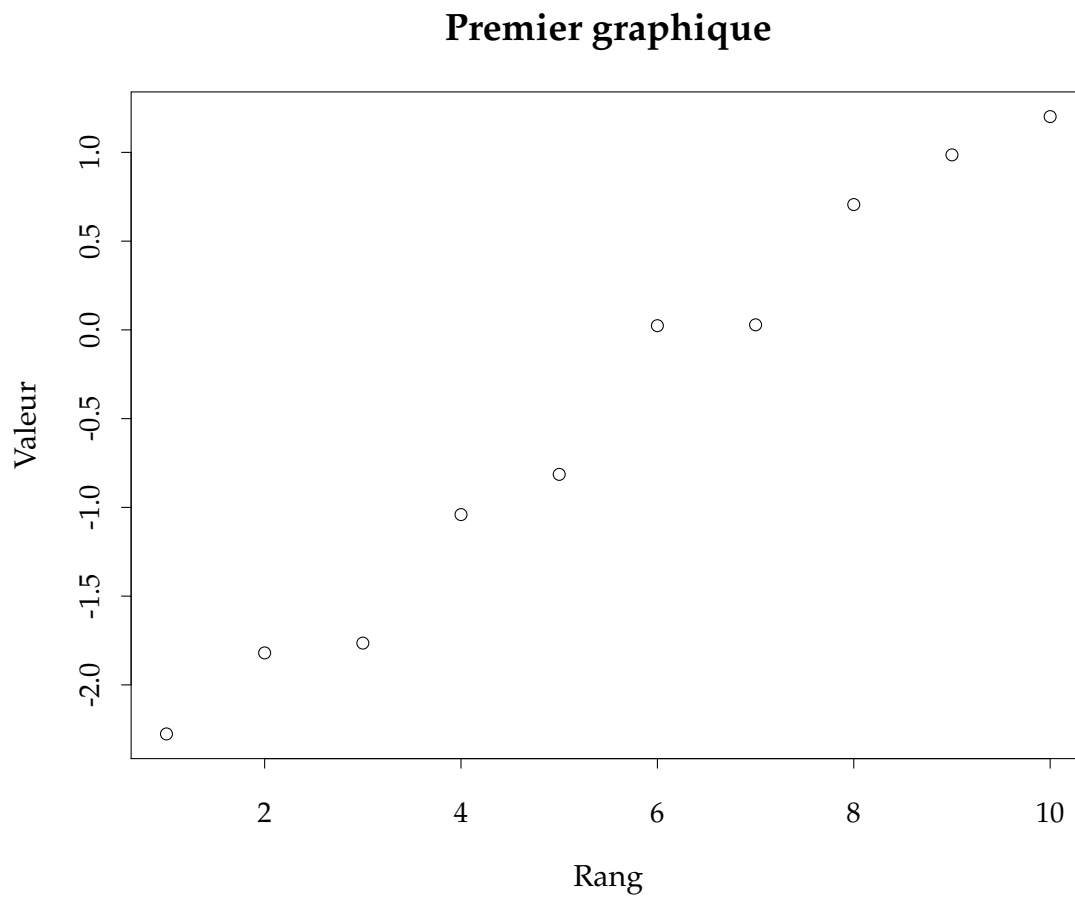
```
distrib <- sort(rnorm(10))  
plot(distrib)
```



R va prendre en charge le calcul d'une grande partie des paramètres nécessaires à la visualisation, comme par exemple les limites des différents axes, et l'espacement entre les valeurs sur les axes. Il est possible de manuellement spécifier l'ensemble de ces paramètres. Par exemple, on peut vouloir changer les étiquettes des axes x et y par quelque chose de plus explicite.

```
distrib <- sort(rnorm(10))  
plot(distrib, xlab = "Rang", ylab = "Valeur", main = "Premier graphique")
```

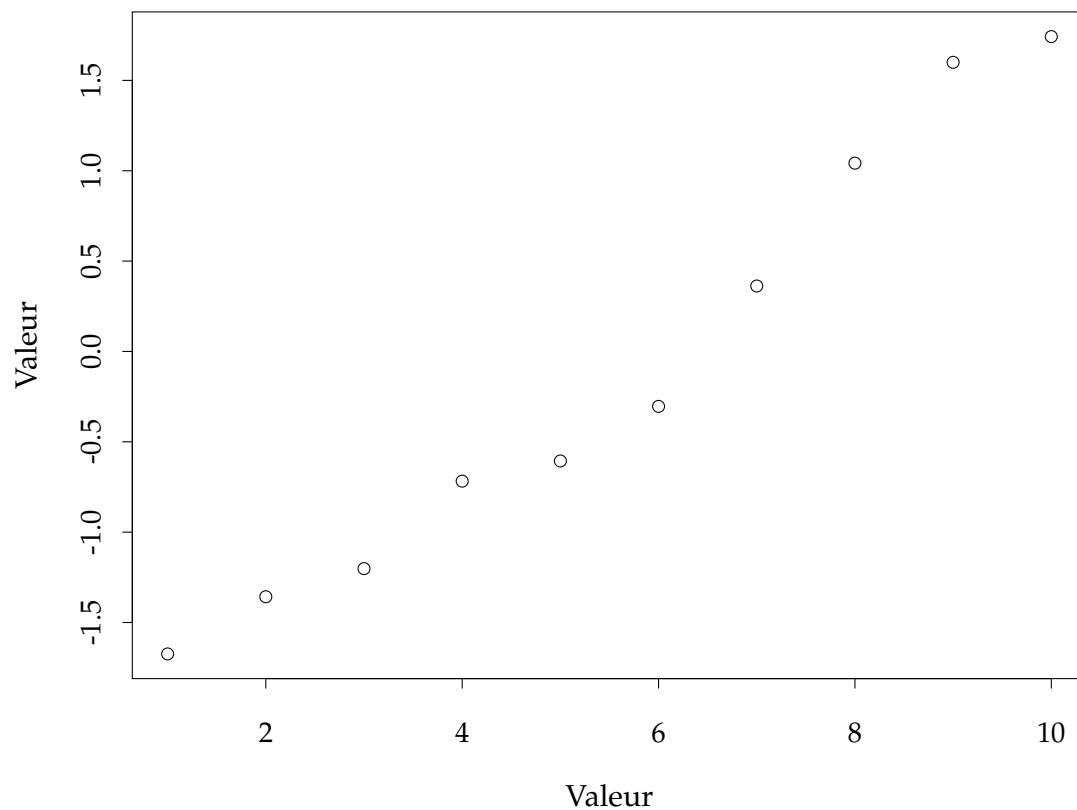




Dans la majorité des cas, R va renseigner des valeurs par défaut pour les étiquettes des axes. Dans le cas où il n'y a pas de valeur par défaut, on peut aussi spécifier les étiquettes par la commande `title` :

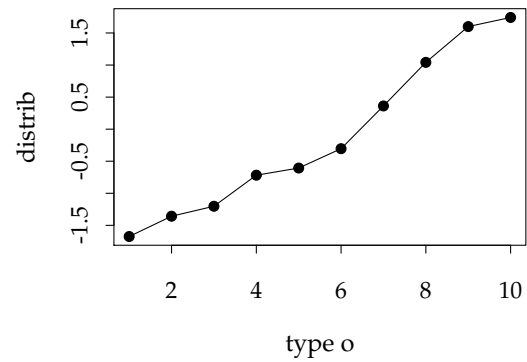
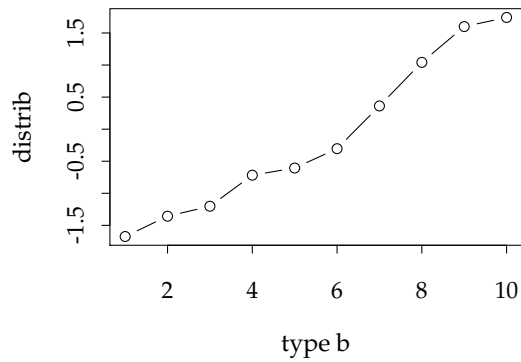
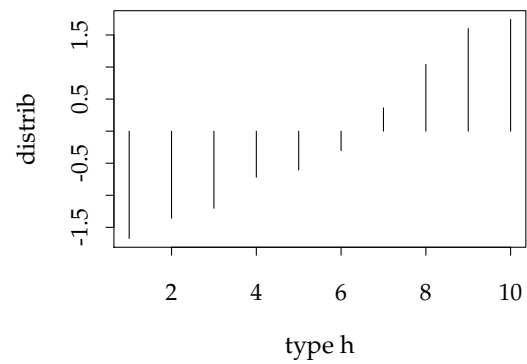
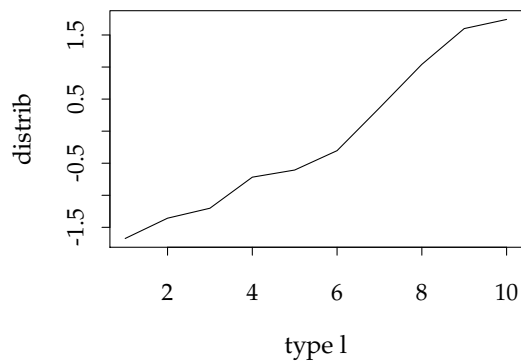
```
distrib <- sort(rnorm(10))  
plot(distrib, xlab = "", ylab = "", main = "")  
title("Premier graphique", xlab = "Valeur", ylab = "Valeur")
```

### Premier graphique

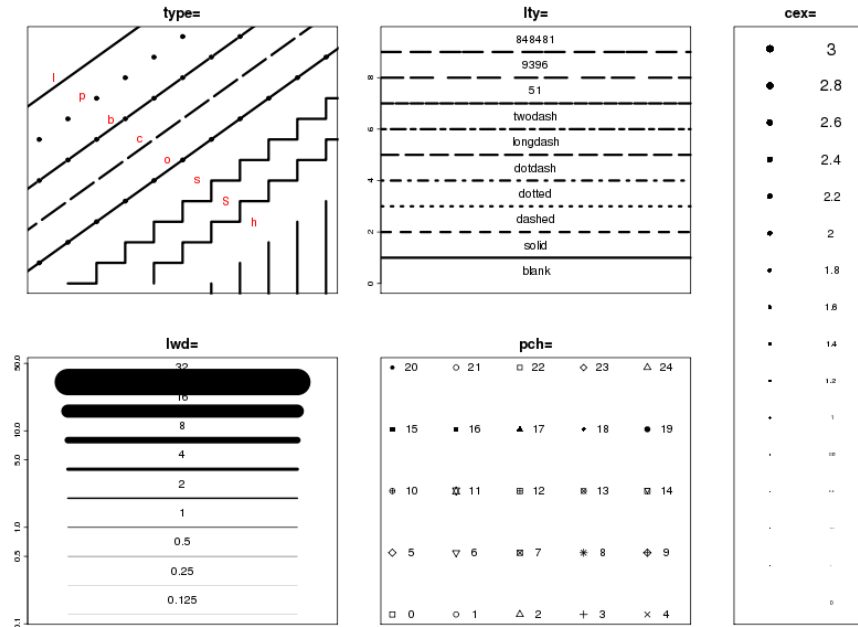


R permet aussi de choisir comment visualiser ces données, en changeant la valeur de l'argument `type` :

```
par(mfcol = c(2, 2))  
plot(distrib, type = "l", xlab = "type l")  
plot(distrib, type = "b", xlab = "type b")  
plot(distrib, type = "h", xlab = "type h")  
plot(distrib, type = "o", pch = 19, xlab = "type o")
```



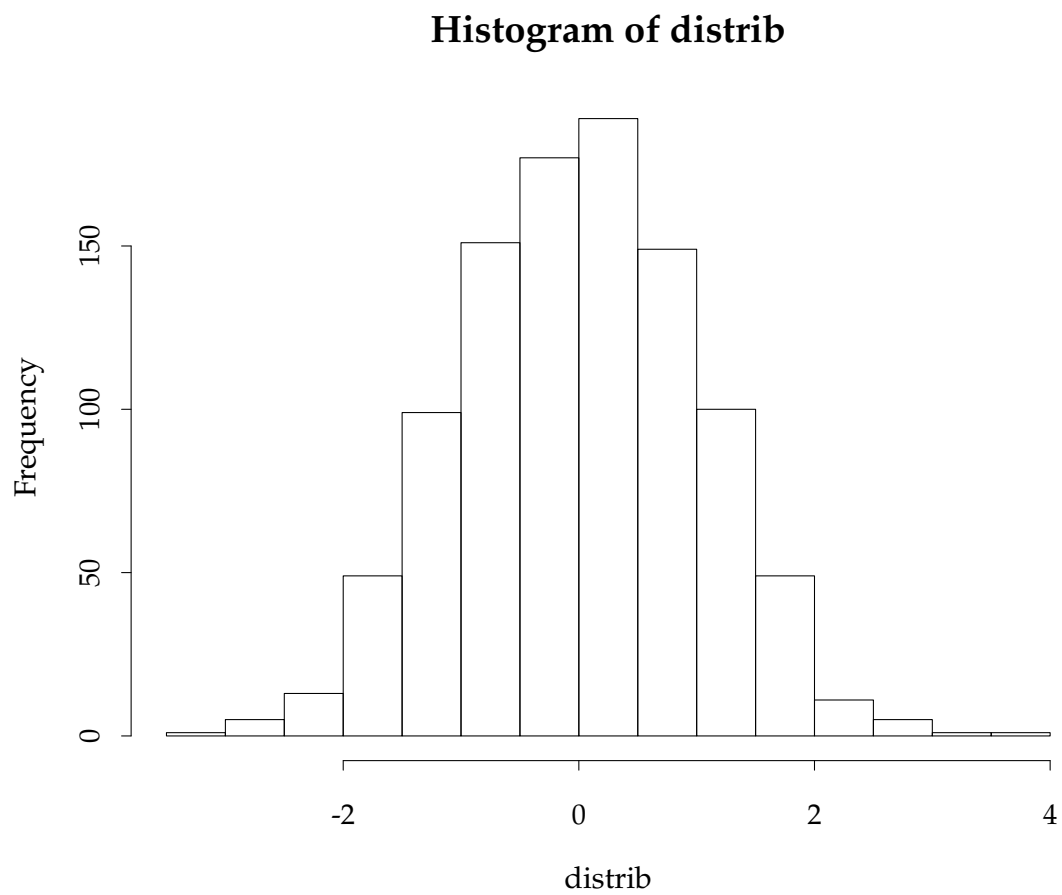
Le type de symbole, le type de ligne, l'épaisseur, *etc*, peuvent être modifiés en ajustant la valeur de différents paramètres. Pour mémoire, un résumé des différentes valeurs possibles et de leurs effets :



## Histogrammes

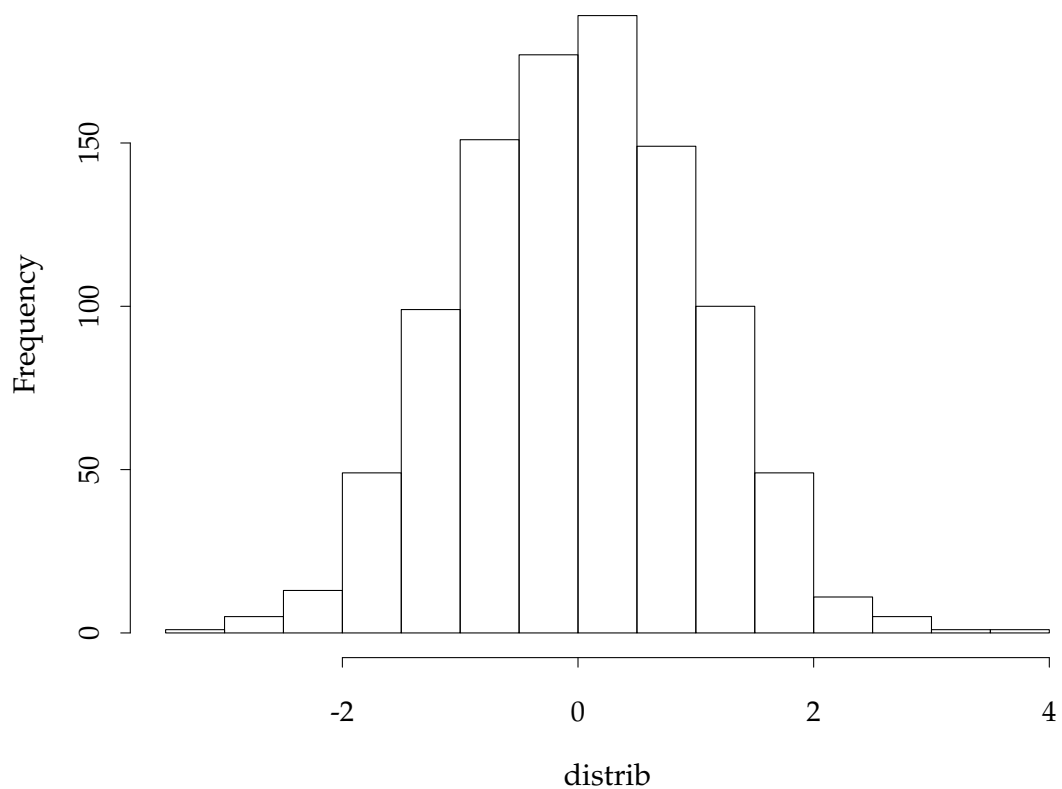
R offre la possibilité de représenter facilement des distributions, *via* des commandes particulières. La plus simple d'utilisation est `hist`, qui permet de représenter un histogramme.

```
distrib <- rnorm(1000)
hist(distrib)
```



Par défaut, R attribue un titre à ces graphiques ; on peut supprimer ce titre en mettant l'argument `main` à une valeur nulle.

```
hist(distrib, main = "")
```

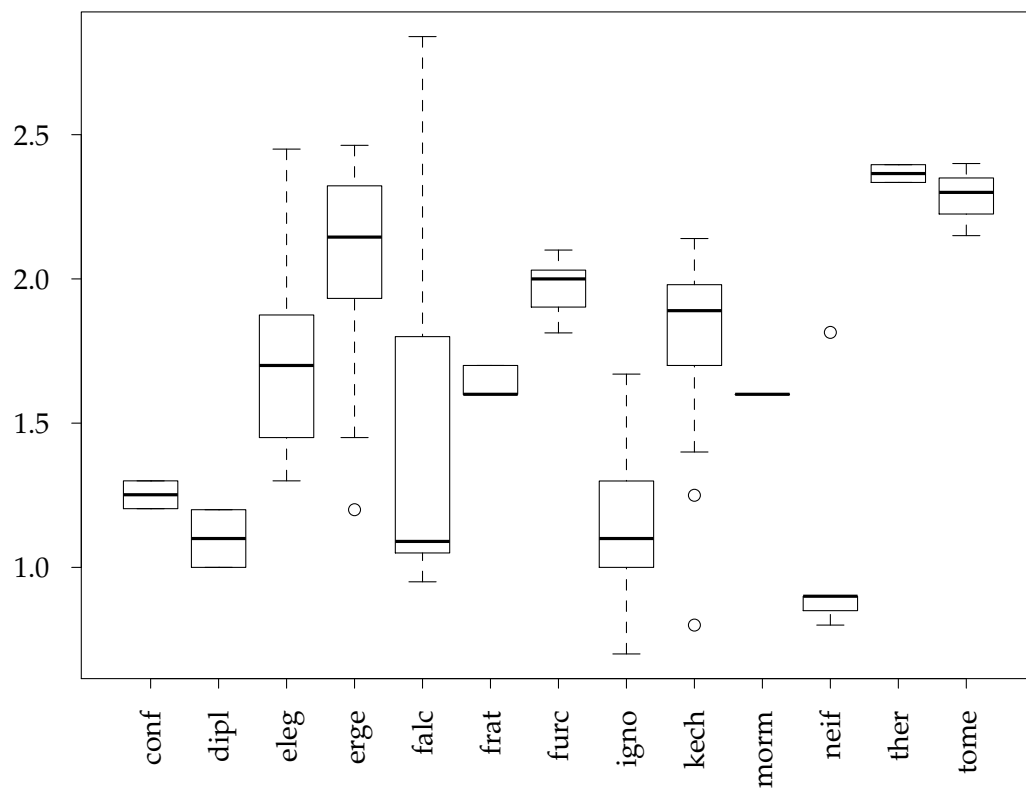


On peut choisir la couleur des barres, *via* l'argument `col`. Comme dans un grand nombre de situations dans lesquelles on fait appel à des vecteurs, R utilise le recyclage : si le nombre de barres de l'histogramme est plus grand que le nombre de couleurs fournies, les couleurs des barres sont alternées.

```
hist(distrib, main = "", col = c(1:8))
```



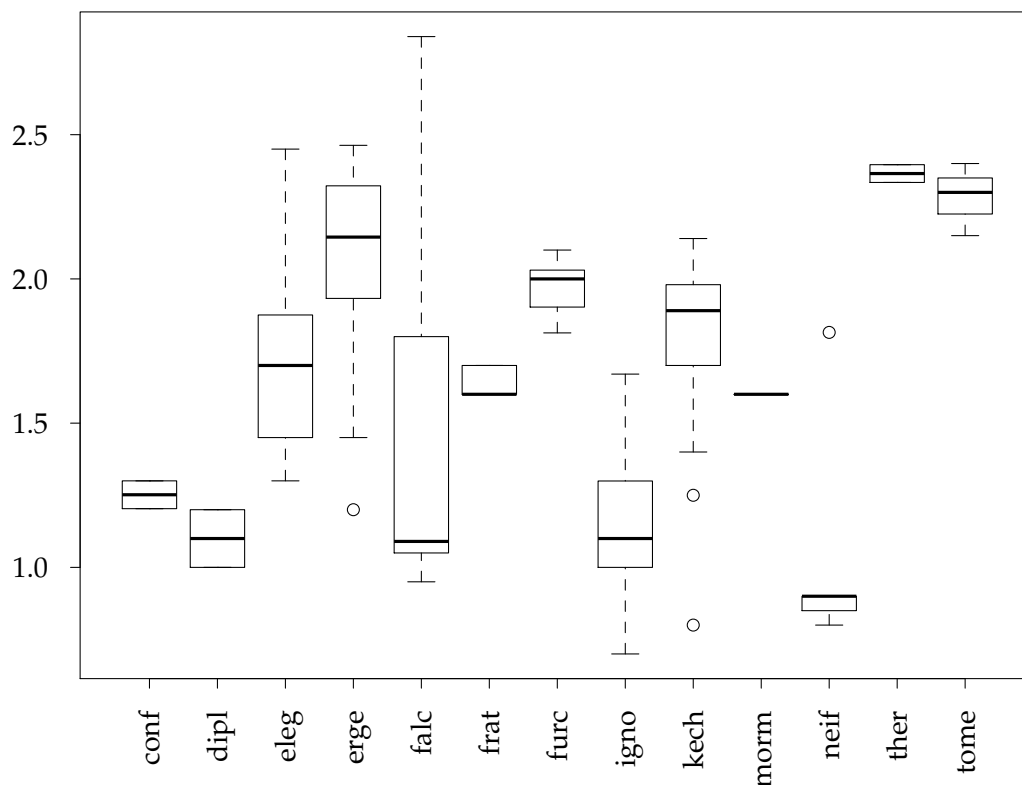
```
boxplot(a ~ sppar, morpho, las = 2)
```



On notera que l'argument `las (?par)` permet de choisir comment les étiquettes des axes sont affichées. Dans le cas des *boxplots*, avoir les étiquettes perpendiculaires permet qu'elles soient toutes affichées. La deuxième façon de spécifier les données est de les passer sous forme de liste. Dans ce cas, on peut aboutir au même résultat que la figure précédente avec :

```
boxplot(split(morpho$a, morpho$sppar), las = 2)
```





## Diagrammes en barres

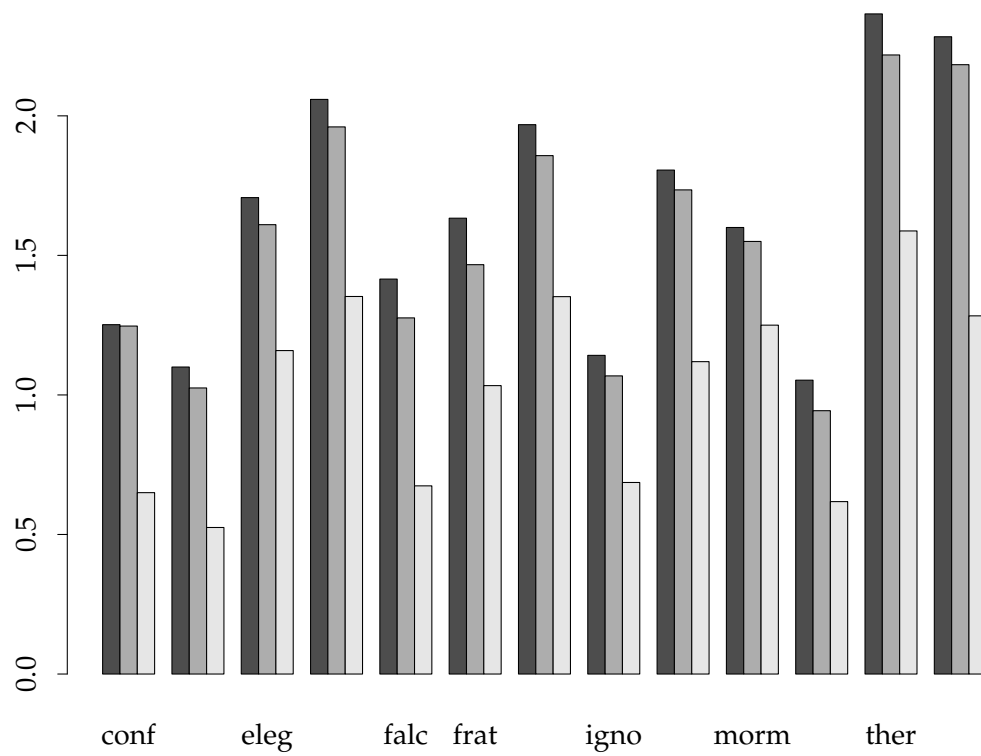
Pour créer des diagrammes en barre, R dispose de la fonction `barplot`. Dans le cas d'une seule série de données, la syntaxe est simplement :

```
Data <- aggregate(morpho$a, by = list(para = morpho$sppar), mean,
  na.rm = TRUE)
barplot(X$x, names.arg = X$para)
```

```
## Error: objet 'X' introuvable
```

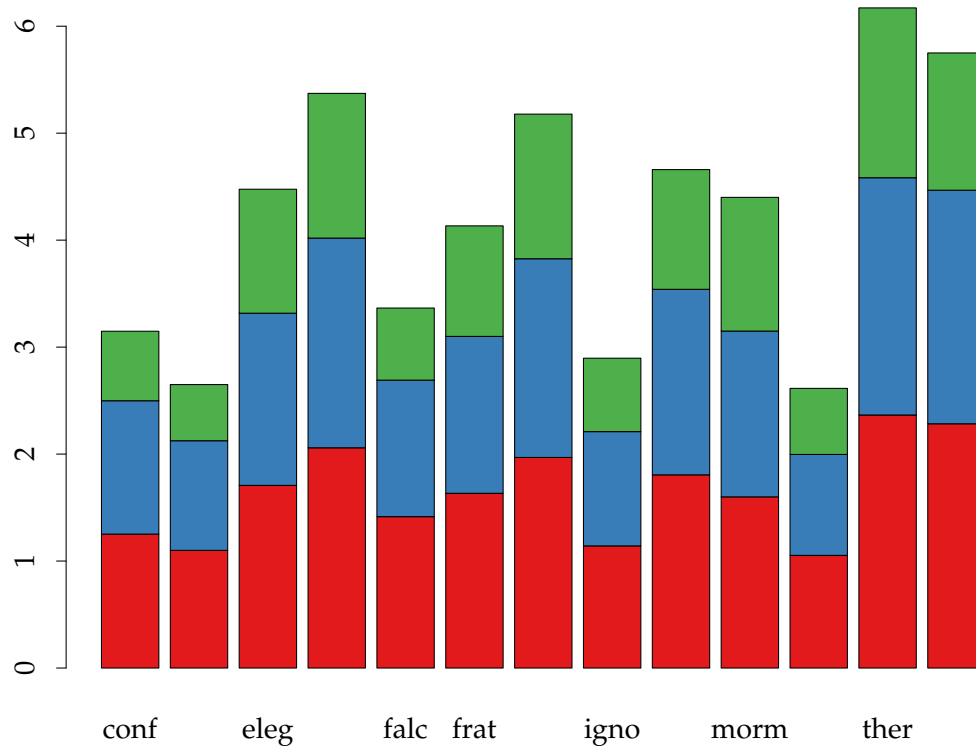
On peut aussi représenter plusieurs séries en même temps, en fournissant une matrice dans laquelle chaque colonne stocke l'information pour les différentes valeurs d'un même groupe. On peut représenter ces différentes valeurs soit en les mettant les unes à côté des autres :

```
X <- aggregate(morpho, by = list(para = morpho$sppar), mean, na.rm = TRUE)
Mat <- t(matrix(c(X$a, X$b, X$c), ncol = 3, byrow = FALSE))
barplot(Mat, names.arg = X$para, beside = TRUE)
```



, soit en les superposant :

```
barplot(Mat, names.arg = X$para, beside = FALSE, col = c(1, 2, 3))
```



On peut modifier la valeur de l'argument `col` pour spécifier la couleur des différentes barres ou fractions de barre.

### Images en trois dimensions

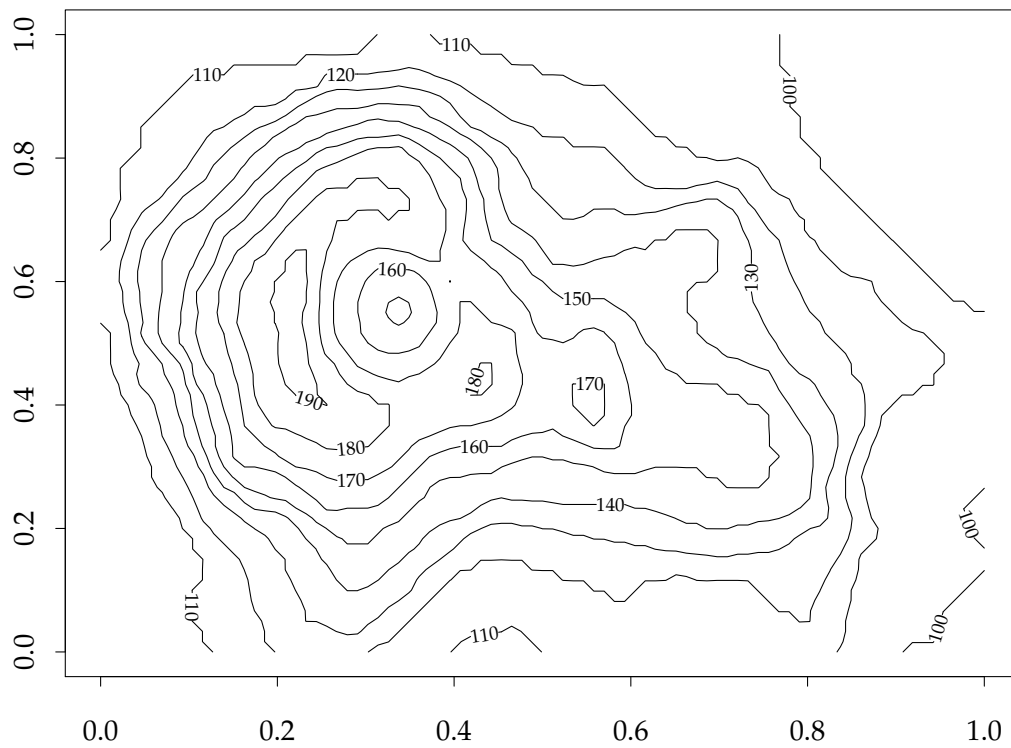
R permet de réaliser assez facilement des graphiques en trois dimensions. On dispose de trois modes de visualisation principaux : les contours, les surfaces, et les perspectives. Dans l'ensemble des cas, il faut que les données soient sous forme de matrice. R propose par défaut le jeu de données `volcano`, que l'on charge avec

```
data(volcano)
```

Ce jeu de données contient une matrice, dans laquelle chaque colonne est l'altitude d'un relevé de la surface du volcan Maunga Whau. Ce jeu de données est standard dans R, et est utilisé pour tester les visualisations en trois dimensions.

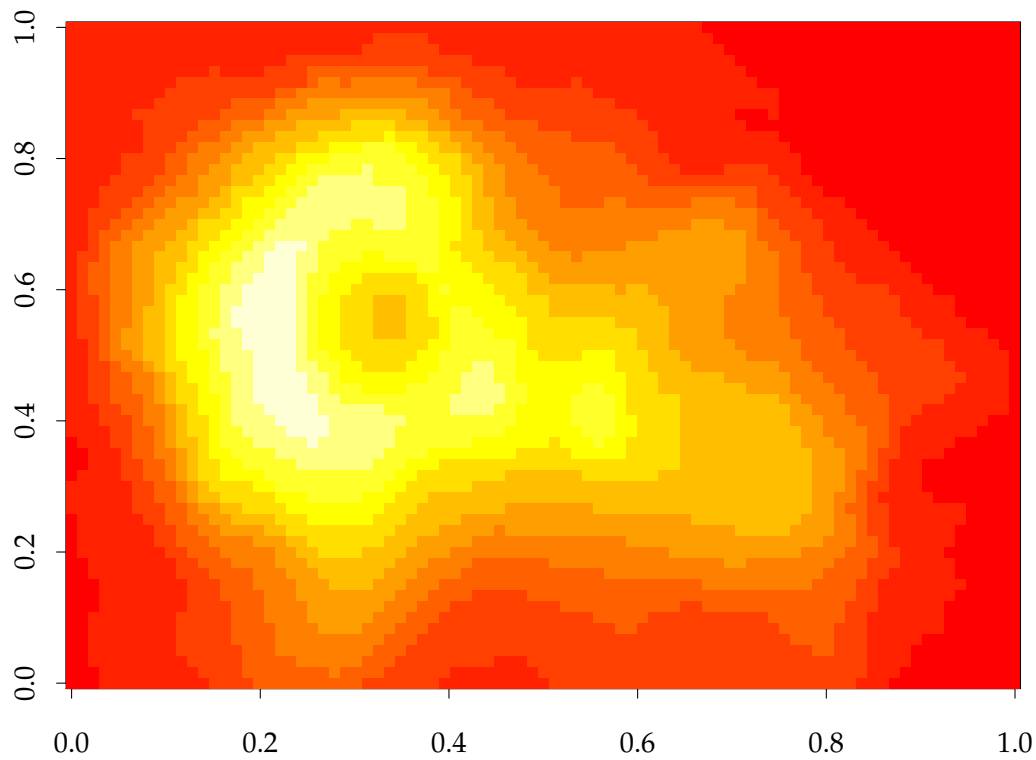
On peut commencer par représenter les contours, avec la commande `contour` :

```
contour(volcano)
```



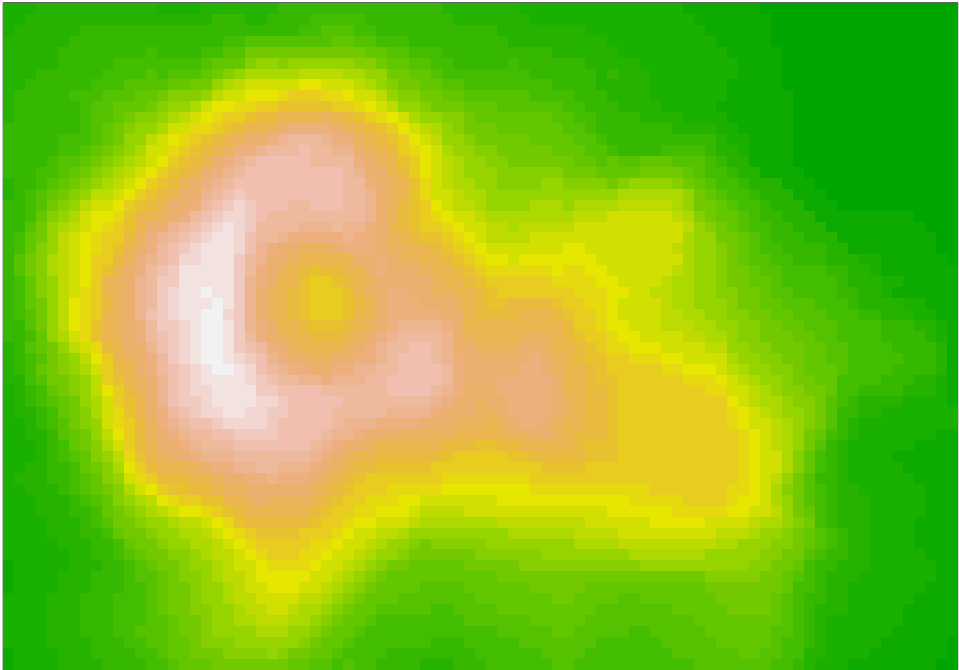
Chaque ligne relie une série de points qui ont la même valeur. Alternativement, on peut créer une image de la surface, avec la fonction `image` :

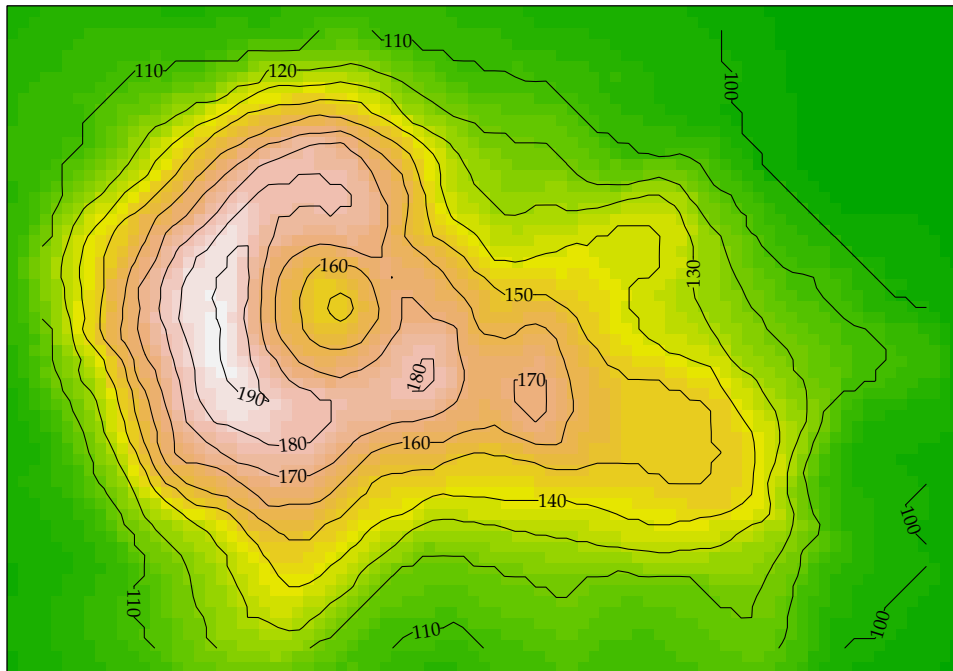
```
image(volcano)
```



Dans certains cas, on peut vouloir superposer les deux informations. Il faut commencer par représenter le fond de l'image, puis ajouter le contour.

```
image(volcano, col = terrain.colors(30), xaxt = "n", yaxt = "n")  
par(new = TRUE)  
contour(volcano, xaxt = "n", yaxt = "n")
```

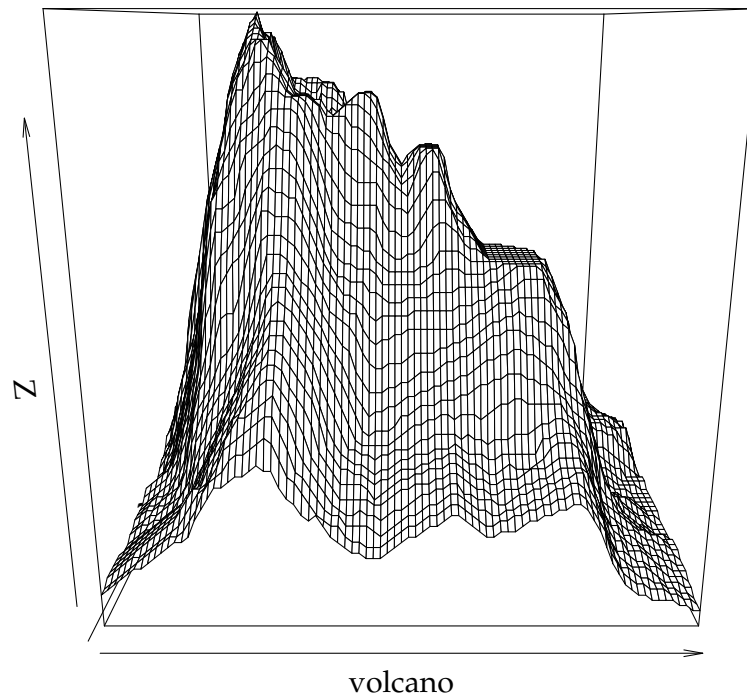




La commande `par(new=TRUE)` indique à R qu'il faut commencer un nouveau graphique *par dessus* le précédent. C'est une des manières possibles pour superposer différentes visualisations. Par souci d'esthétique, on a supprimé les axes des deux images, en utilisant les commandes `yaxt = 'none'` et `xaxt = 'none'`.

Enfin, la dernière visualisation est en trois dimensions :

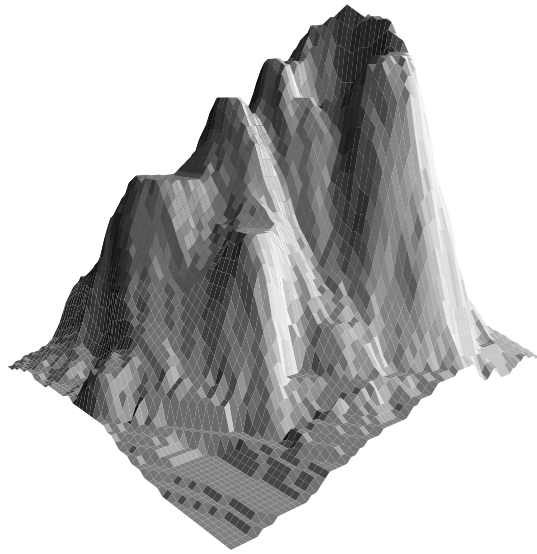
```
persp(volcano)
```



On peut obtenir une version plus aboutie en spécifiant la rotation, l'éclairage, et en supprimant les axes :

```
persp(volcano, box = FALSE, phi = 8, theta = 140, shade = TRUE, border = NA)
```





Notez aussi que l'aide de `persp` possède un exemple qui explique comment colore chaque polygone en fonction de son élévation.

## 5.2 Ajout d'éléments sur un graphique

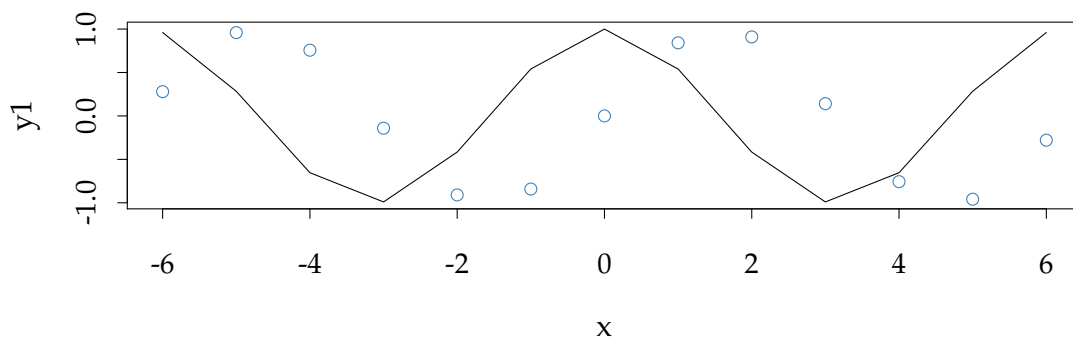
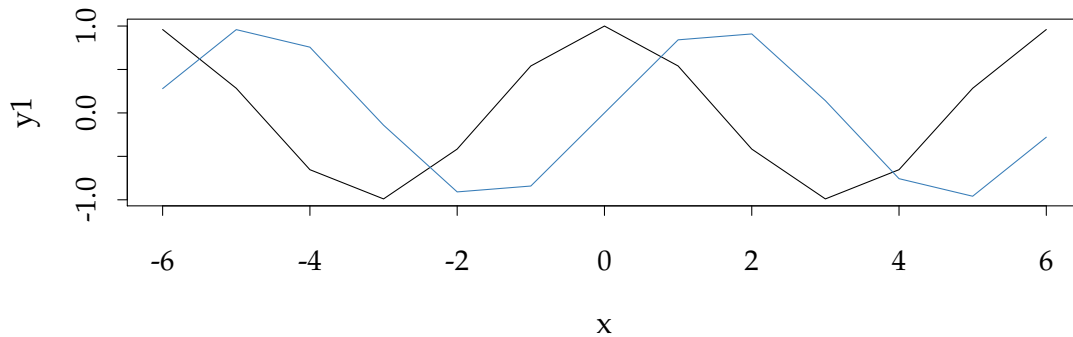
Dans un grand nombre de situations, il est nécessaire de faire apparaître plus d'un élément sur un graphique. Cette section couvre les utilisations les plus fréquentes.

### Autres séries de données

Une fois un graphique affiché à l'écran, on peut ajouter d'autres séries de données assez simplement.

```
x <- seq(from = -6, to = 6, length = 350)
y1 <- cos(x)
y2 <- sin(x)
par(mfcol = c(2, 1))
plot(x, y1, type = "l")
lines(x, y2, col = 2)
```

```
plot(x, y1, type = "l")
points(x, y2, col = 2)
```

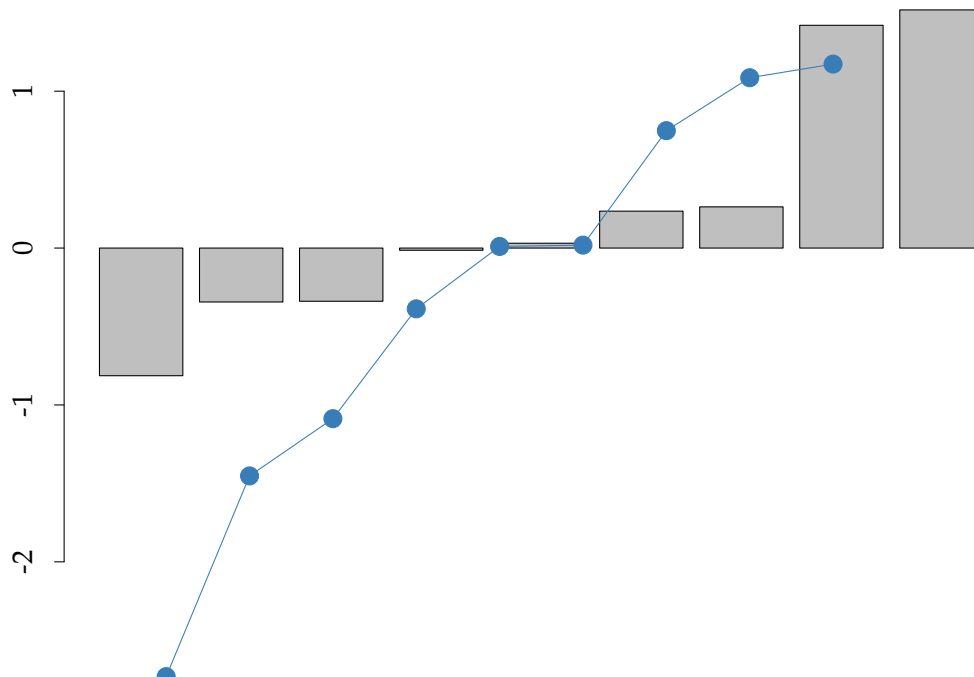


Les commandes `lines` et `points` se comportent exactement de la même manière que `plot` : vous pouvez omettre la série `x`, et vous pouvez passer des arguments `type`.

On peut ajouter des données de cette manière sur n'importe quel type de graphique :

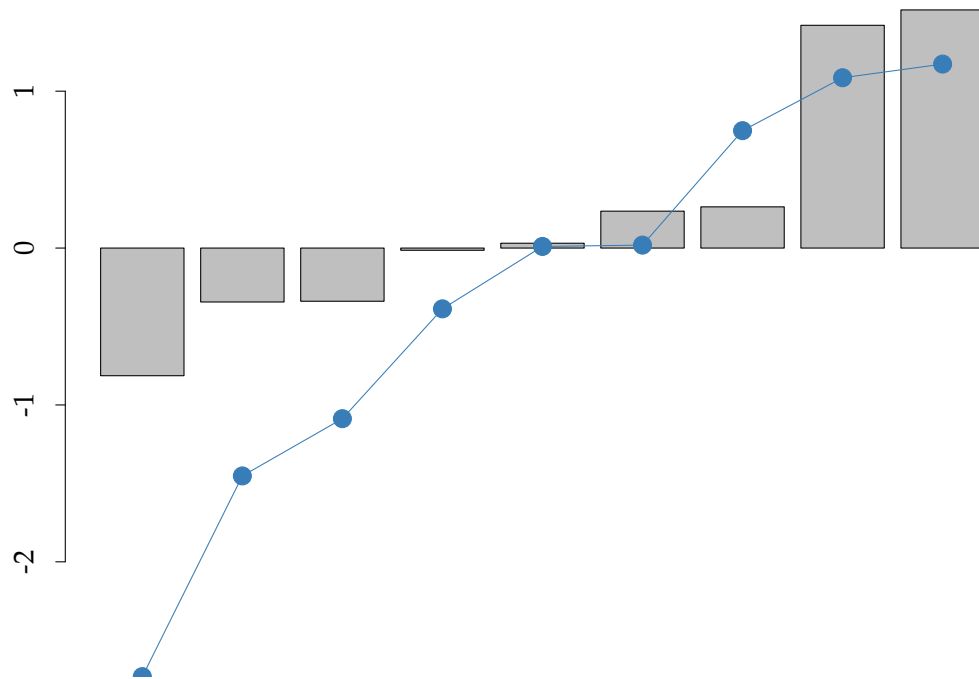
```
x <- sort(rnorm(9))
x2 <- sort(rnorm(9))

barplot(x, ylim = range(c(x, x2)))
lines(x2, col = 2, type = "o", pch = 19, cex = 1.5)
```



Comme dans cet exemple, il faut cependant prendre en compte le fait que la position par défaut des éléments sur les différents axes n'est pas toujours intuitive. Il faut donc connaître la position des éléments. La fonction `barplot` renvoie cette information, et on peut donc refaire le graphique précédent :

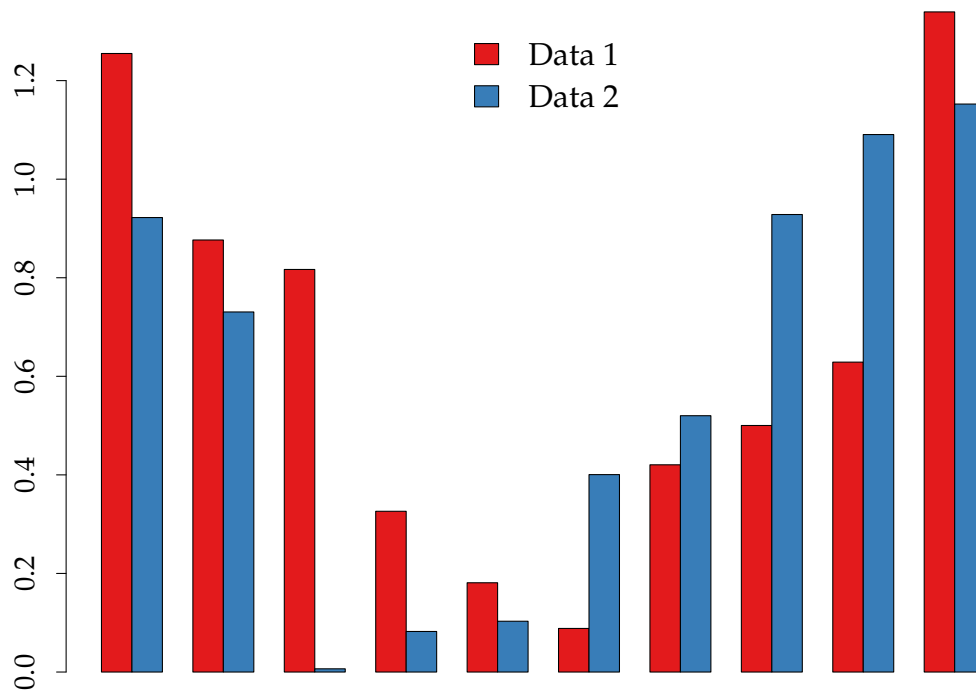
```
Xpos <- barplot(x, ylim = range(c(x, x2)))  
lines(Xpos, x2, col = 2, type = "o", pch = 19, cex = 1.5)
```



## Légendes

L'ajout de légendes est un élément important dans la conception d'un graphique efficace. R propose une fonction `legend` très puissante et intuitive pour créer ces légendes. Par exemple :

```
Heights <- matrix(c(abs(sort(rnorm(10)))), abs(sort(rnorm(10)))),
  nrow = 2, byrow = TRUE)
barplot(Heights, beside = TRUE, col = c(1, 2))
legend("top", legend = c("Data 1", "Data 2"), fill = c(1, 2), bty = "n")
```



Pour disposer une légende, R a besoin de trois séries d'informations distinctes :

1. La position de la légende sur le graphique, donnée soit par des positions x et y, soit par un mot-clé choisi dans "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" et "center"
2. Le texte de la légende, contenu dans l'argument legend, sous forme de vecteur
3. Les informations sur le formatage des séries de données, par exemple type de lignes, types de points, etc...

On peut donc spécifier des légendes assez complexes, comme dans l'exemple suivant :

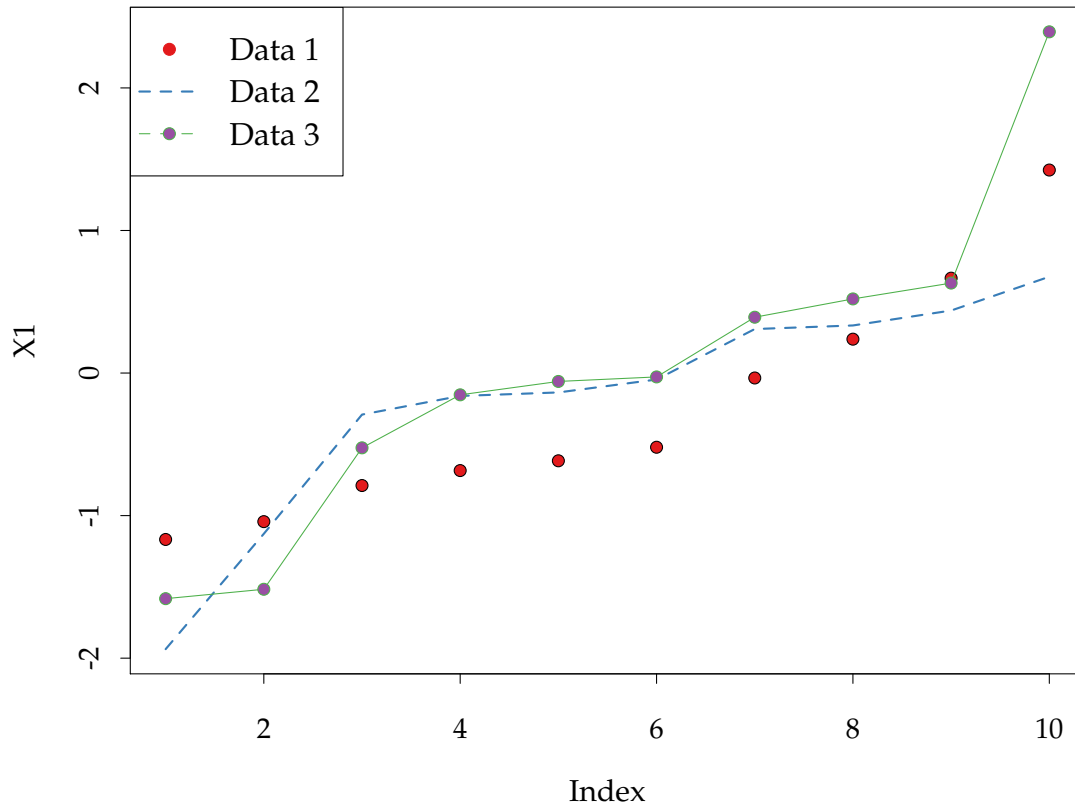
```
X1 <- sort(rnorm(10))
X2 <- sort(rnorm(10))
X3 <- sort(rnorm(10))

Range <- range(c(X1, X2, X3))

plot(X1, pch = 21, bg = 1, ylim = Range)
lines(X2, col = 2, lty = 2, lwd = 2)
```

```
lines(X3, pch = 21, bg = 4, col = 3, type = "o")

legend("topleft", legend = c("Data 1", "Data 2", "Data 3"), lty = c(NA,
  2, 2), lwd = c(NA, 2, 1), pch = c(21, NA, 21), col = c(1, 2, 3), pt.bg = c(1,
  NA, 4))
```



## Axes

Dans certains cas, on veut pouvoir ajouter plusieurs axes sur un unique graphique. Par exemple, dans le cas des données de la séance 3, on peut s'intéresser à la variation du prix des peaux de lynx en fonction du nombre de captures.

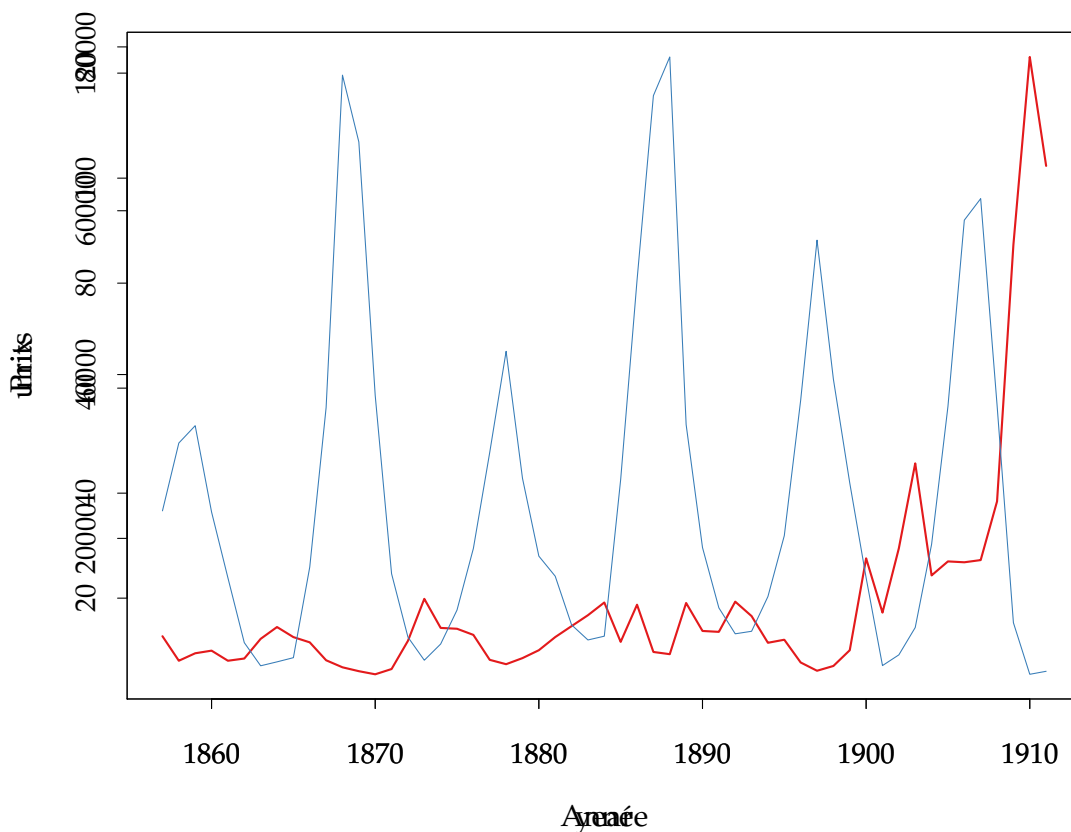
```
lynx <- read.table(file = "data/lynxdata.txt", h = TRUE)
summary(lynx)
```

```
##      year      units      prices
##  Min.   :1857   Min.    : 3410   Min.    :  5.50
##  1st Qu.:1870   1st Qu.: 8506   1st Qu.:  8.96
```

```
## Median :1884   Median :18774   Median : 12.58
## Mean    :1884   Mean    :25054   Mean    : 19.54
## 3rd Qu.:1898   3rd Qu.:36044   3rd Qu.: 19.12
## Max.    :1911   Max.    :78773   Max.    :123.08
```

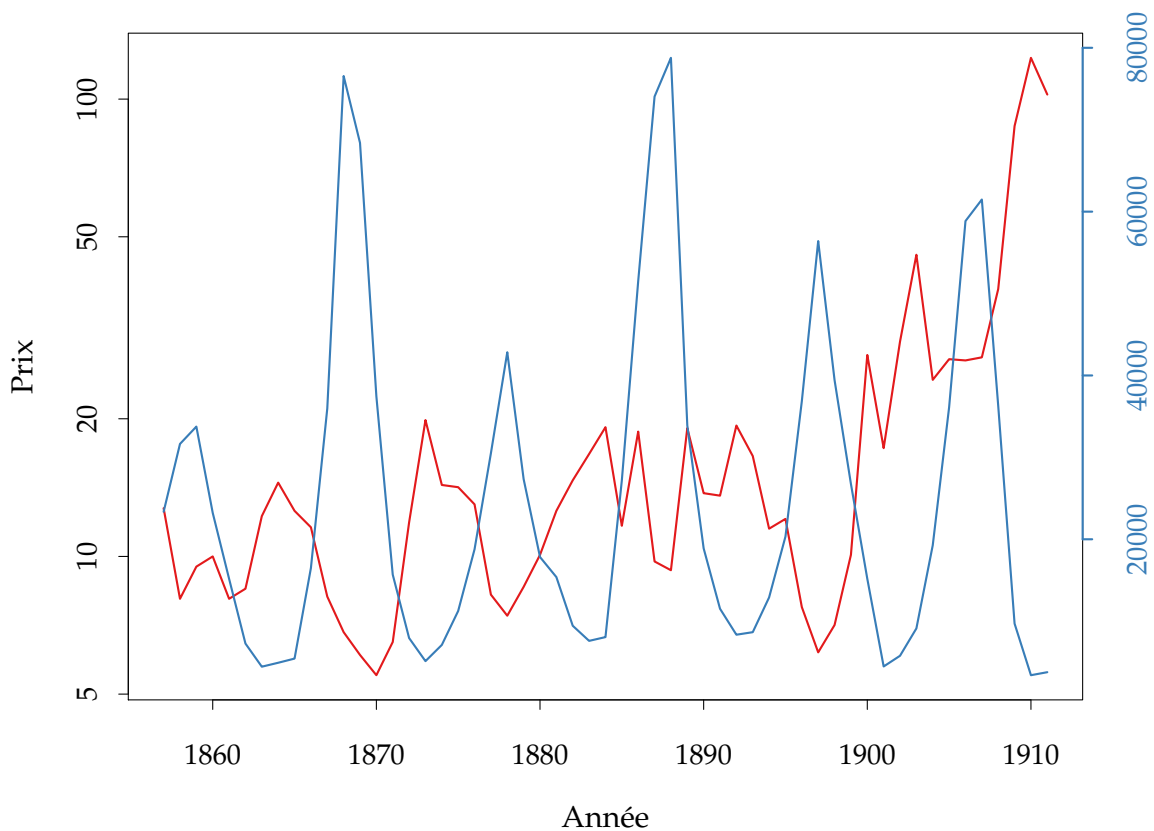
L'application de la fonction `summary` nous montre que les différentes données ont des valeurs qui diffèrent de plusieurs ordres de grandeur. On va donc utiliser deux axes pour les représenter :

```
with(lynx, {
  plot(year, prices, type = "l", xlab = "Année", ylab = "Prix", lwd = 2,
        col = 1)
  par(new = TRUE)
  plot(year, units, type = "l", col = 2)
})
```



En utilisant simplement un nouveau plot, les deux axes  $y$  se superposent. Il faut donc supprimer les axes du deuxième graphique (ce qu'on a déjà vu dans les paragraphes précédents, mais qu'on peut faire en plus court avec l'argument `axes=FALSE`), et ajouter un nouvel axe avec la commande `axis` :

```
with(lynx, {
  plot(year, prices, type = "l", xlab = "Année", ylab = "Prix", lwd = 2,
        col = 1, log = "y")
  par(new = TRUE)
  plot(year, units, type = "l", col = 2, axes = FALSE, xlab = "", ylab = "",
        lwd = 2)
  axis(4, col = 2, col.ticks = 2, col.axis = 2, lwd = 2)
})
```



Notez aussi qu'on a ajouté l'option `log='y'` à la première commande, pour refléter le fait que les prix ont été multipliés par 10 pendant la période étudiée.

### 5.3 Enregistrement des figures

R permet non seulement d'afficher les graphiques dans une fenêtre à part, mais aussi de les enregistrer dans différents formats. La structure générale du code permettant d'enregistrer une figure est toujours la même :



```
open_device(file = "file.extension")
plot(my_data)
dev.off()
```

La commande `open_device` peut prendre plusieurs formes selon le type de fichier désiré en sortie. Les plus communes sont sans doute pdf, png, et tiff. Reportez vous à l'aide de chacune de ces fonctions pour comprendre les arguments, et essayez d'exporter dans ces formats un des graphiques réalisés pendant cette partie.

La commande `dev.off()` est extrêmement importante : elle permet de fermer le périphérique graphique actif. Sans cette commande, le fichier n'est pas fini d'écrire, et il ne pourra pas être lu à la fermeture de R. Dans RStudio, quel est l'effet de la commande `dev.off()` après qu'un graphique ait été affiché ?

## 5.4 Mise en application

### Diagramme en barres

Dans cette mise en application, on veut créer une visualisation qui met en avant les valeurs extrêmes d'une distribution, en utilisant un diagramme en barre. Spécifiquement, on souhaite que les barres correspondant à des valeurs plus petites, ou plus grandes, que des valeurs fixées, soient colorées différemment. En vous aidant de ce qui a été vu jusqu'ici, et de l'aide de la fonction `barplot`, produisez cette visualisation.

### Dynamique de populations

VEILLEUX [7] a mesuré, deux fois par jour, la dynamique de population du prédateur *Dinimum nasutum* et de sa proie *Paramecium aurelia*. Les données sont disponibles dans le fichier `data/predprey.txt`. Lisez le contenu de ce fichier de données, et représentez la dynamique des populations en respectant les contraintes suivantes, qui sont plus où moins celles imposées pour la préparation des figures dans une publication :

1. Les deux populations doivent être représentées sur le même axe. Si la taille de l'axe par défaut, la commande `ylim=c(min,max)` devrait vous aider.
2. Pour différentes raisons, vous ne pouvez pas imprimer en couleur. Utilisez soit différents types de points, soit différentes lignes.
3. Légendez votre figure, incluant les titres des axes.

## 5.5 Solution des mises en application

### Diagramme en barres

On souhaite représenter une diagramme en barres, et colorer les barres qui sont au dessous ou au dessus de valeurs données. On commence par choisir la série de données qui nous intéresse – on peut la trier en ordre croissant, pour faciliter la lecture du graphique :

```
test_data <- c(1, 4, 3, 12, -2, -6, -1, 1)
test_data <- sort(test_data)
```

Pour colorer chaque barre de manière indépendante, on va créer un vecteur colors, qui contiendra une valeur par barre :

```
colors <- rep(2, length(test_data))
```

Chaque barre sera donc de la couleur 2, qui dépend de la palette actuelle. Pour changer la couleur de chaque barre, il y a deux approches. La première consiste à écrire une boucle :

```
for (co_idx in c(1:length(test_data))) {
  if (test_data[co_idx] > 10)
    colors[co_idx] <- 3
  if (test_data[co_idx] < -5)
    colors[co_idx] <- 1
}
```

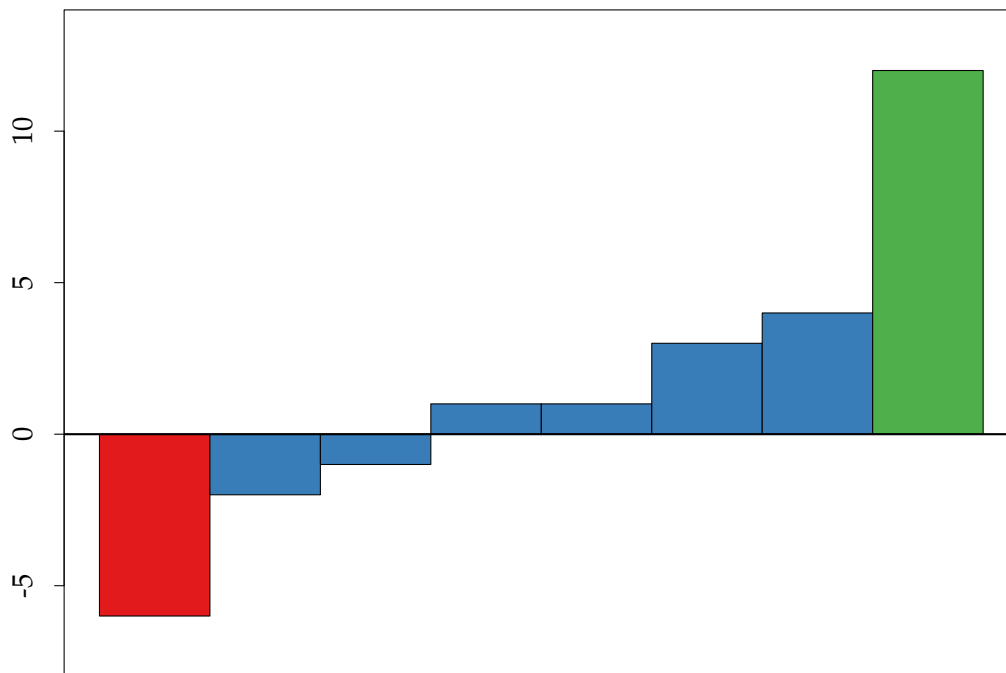
Une méthode plus élégante tire parti de la vectorisation :

```
colors[test_data > 10] <- 3
colors[test_data < -5] <- 1
colors
```

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

On peut ensuite afficher le graphique avec les barres de chaque couleur :

```
barplot(test_data, col = colors, ylim = range(test_data) + c(-2,
  2), space = 0)
abline(h = 0, lwd = 2)
box()
```



## Dynamique de populations

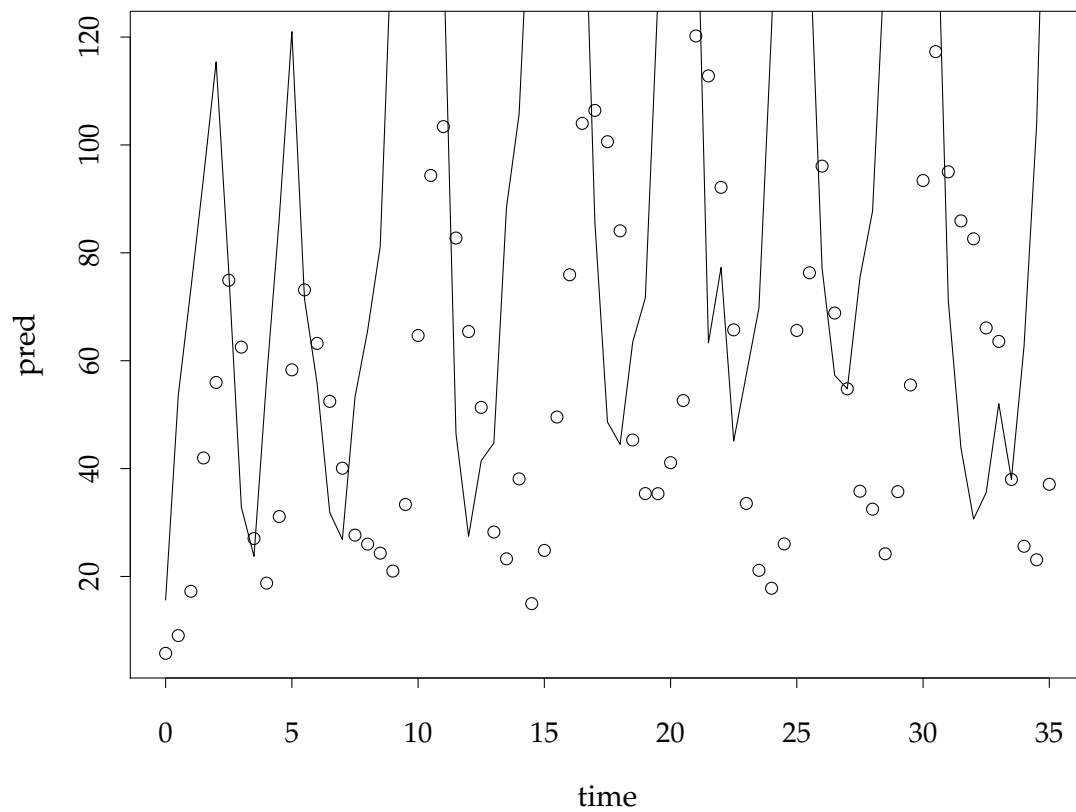
On commence par lire les données :

```
pp <- read.table("data/predprey.txt", h = TRUE, sep = " ", dec = ".")  
head(pp)
```

```
##   time  prey  pred  
## 1  0.0  15.65  5.76  
## 2  0.5  53.57  9.05  
## 3  1.0  73.34 17.26  
## 4  1.5  93.93 41.97  
## 5  2.0 115.40 55.97  
## 6  2.5  76.57 74.91
```

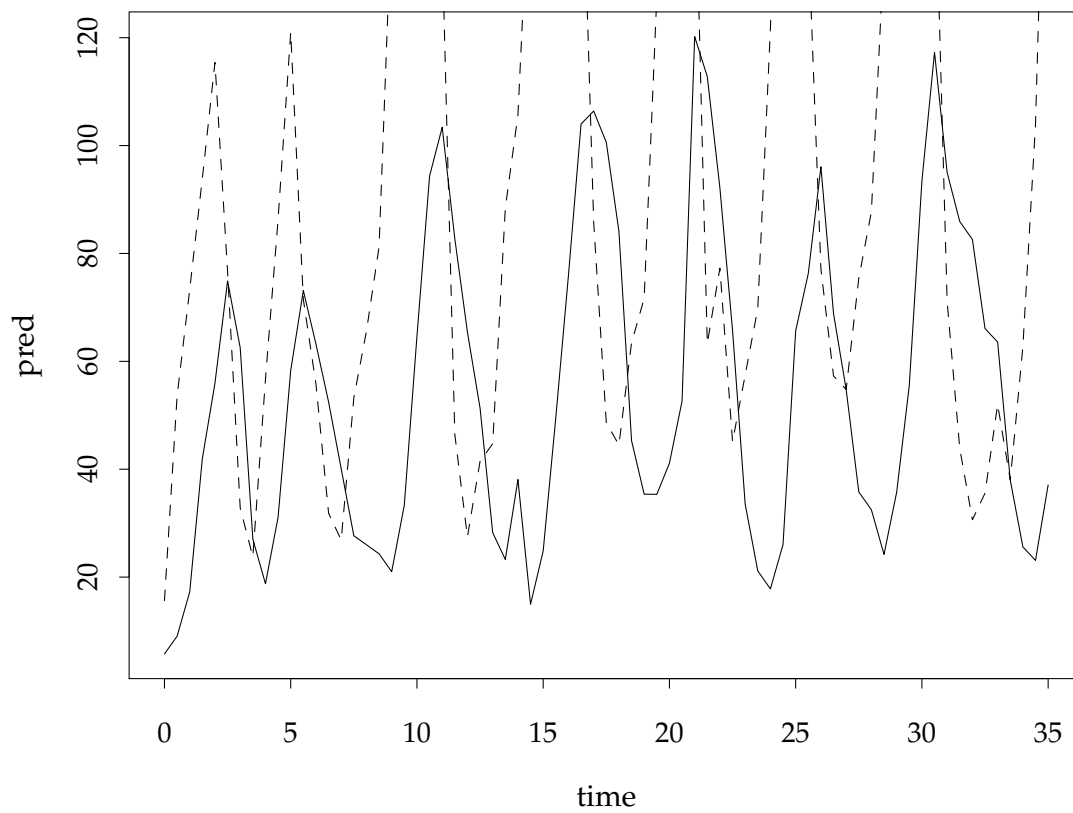
On commencer par faire une première visualisation :

```
with(pp, {
  plot(time, pred)
  lines(time, prey)
})
```



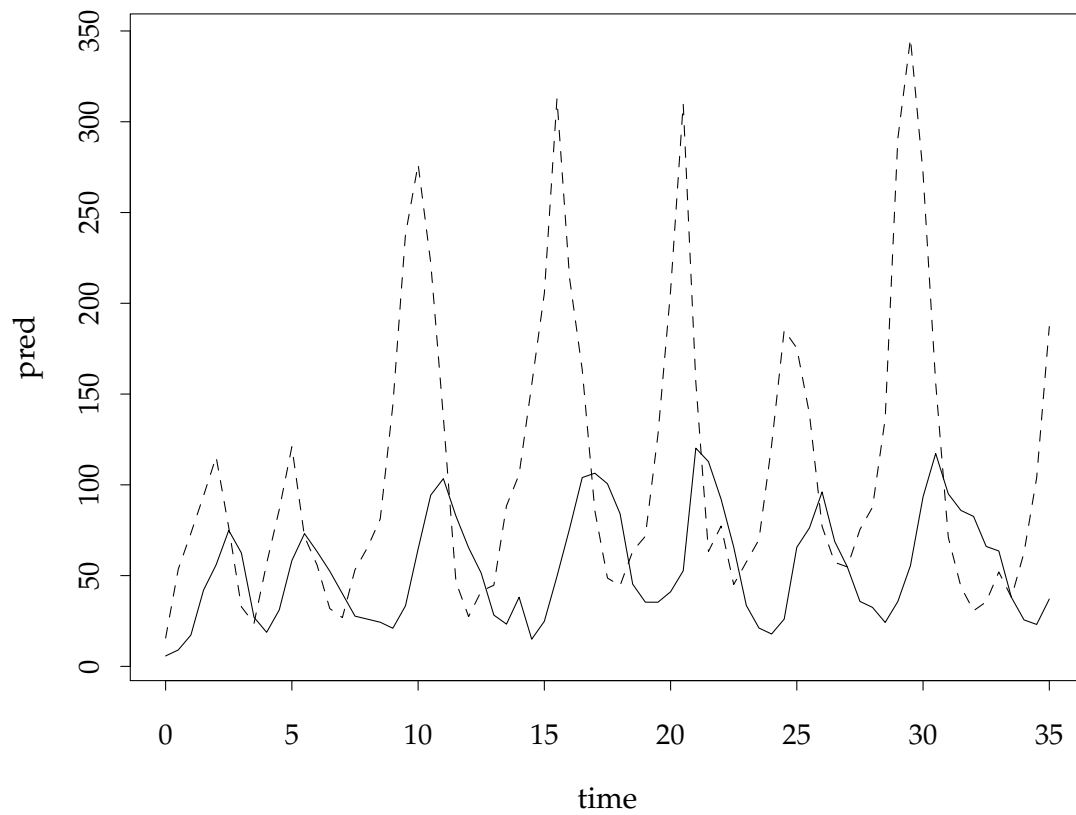
On va maintenant changer le type de lignes :

```
with(pp, {
  plot(time, pred, type = "l", lty = 1)
  lines(time, prey, lty = 2)
})
```



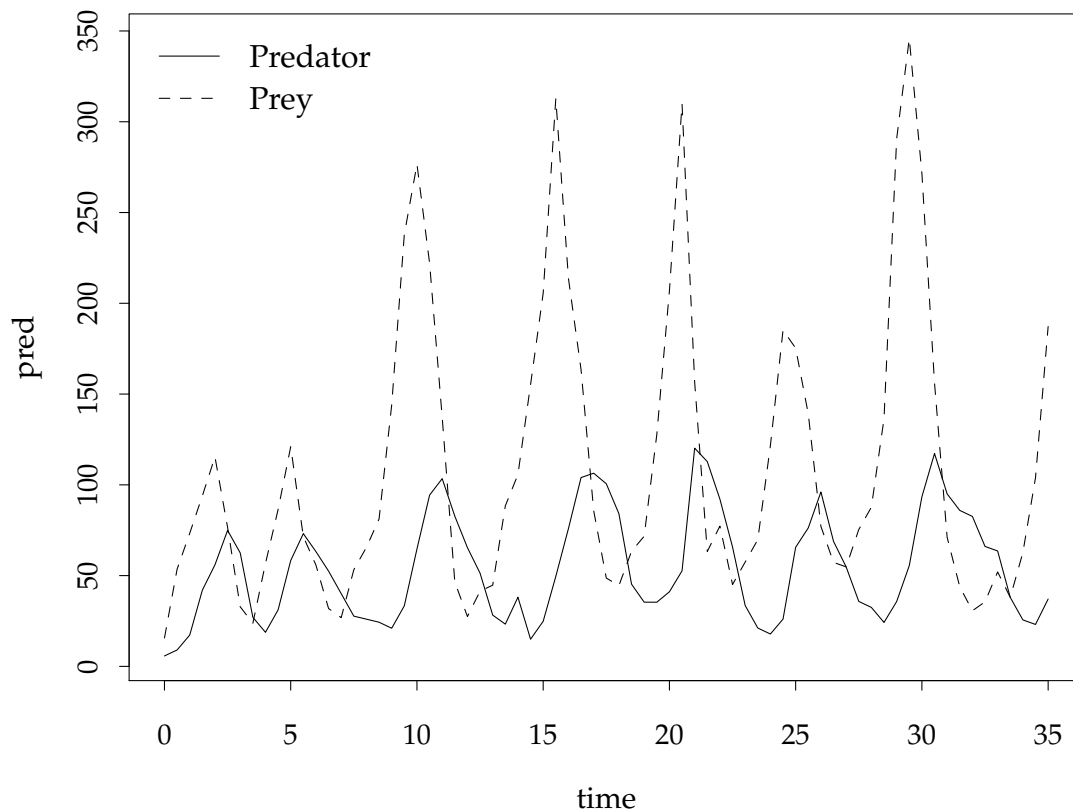
On peut obtenir le *range* directement :

```
with(pp, {  
  RangeY <- range(c(pred, prey))  
  plot(time, pred, type = "l", lty = 1, ylim = RangeY)  
  lines(time, prey, lty = 2)  
})
```



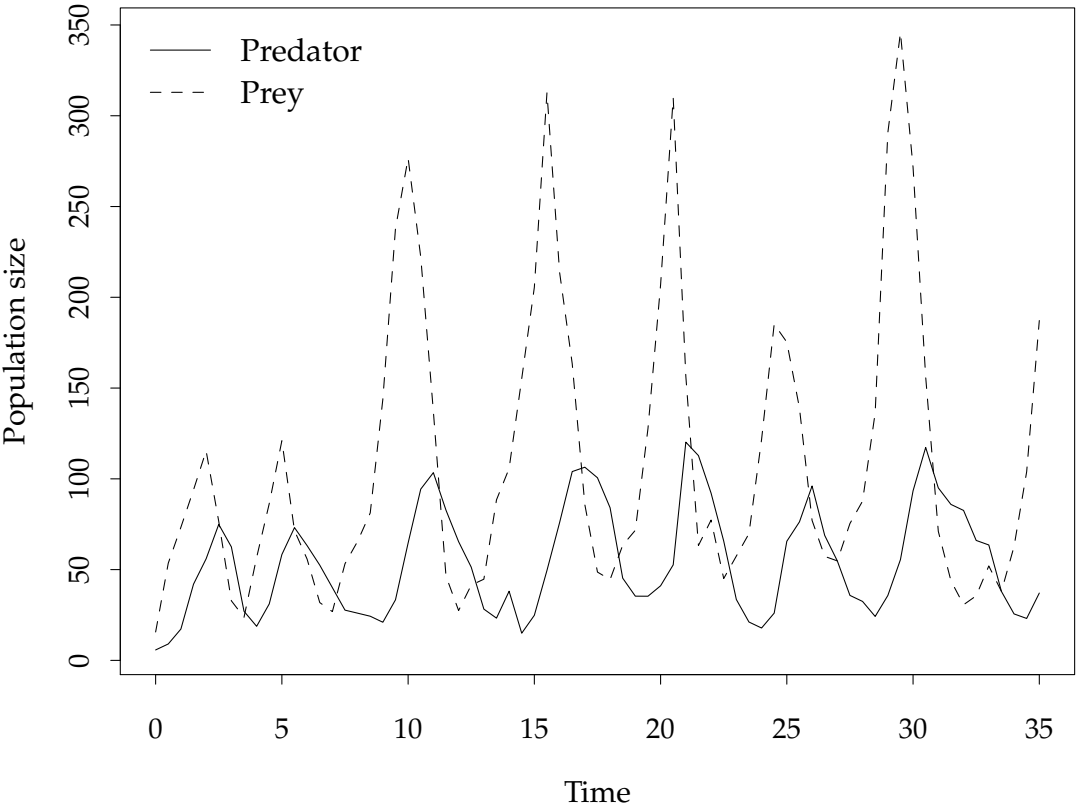
On ajoute maintenant la légende

```
with(pp, {
  RangeY <- range(c(pred, prey))
  plot(time, pred, type = "l", lty = 1, ylim = RangeY)
  lines(time, prey, lty = 2)
  legend("topleft", lty = c(1, 2), legend = c("Predator", "Prey"), bty = "n")
})
```



Enfin, on ajoute les titres des axes :

```
with(pp, {  
  RangeY <- range(c(pred, prey))  
  plot(time, pred, type = "l", lty = 1, ylim = RangeY, xlab = "Time", ylab =  
"Population size")  
  lines(time, prey, lty = 2)  
  legend("topleft", lty = c(1, 2), legend = c("Predator", "Prey"), bty = "n")  
})
```





# ***Bibliographie***

- [1] R DEVELOPMENT CORE TEAM. *R : A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2008.
- [2] T. KACI-CHAOUCH et al. "Host specificity is linked to intraspecific variability in the genus *Lamellodiscus* (Monogenea)". Dans : *Parasitology* 135.05 (2008), p. 607–616.
- [3] T. POISOT et Y. DESDEVISES. "Putative speciation events in *Lamellodiscus* (Monogenea : Diplectanidae) assessed by a morphometric approach". Dans : *Biological Journal of the Linnean Society* 99.3 (fév. 2010), p. 559–569.
- [4] D. F. ANDREWS et A. M. HERZBERG. "Data : a collection of problems from many fields for the student and research worker". Dans : Springer, 1985, p. 442.
- [5] P. MURRELL. *R Graphics*. Chapman et Hall, 2005.
- [6] I. H. IVERSEN. "Tactics of graphic design : A review of Tufte's *The Visual Display of Quantitative Information*". Dans : *Journal of the experimental analysis of behavior* 49.1 (1988), p. 171–189.
- [7] B. G. VEILLEUX. "An Analysis of the Predatory Interaction Between *Paramecium* and *Didinium*". Dans : *The Journal of Animal Ecology* 48.3 (oct. 1979), p. 787.