Initiation à R

Brigitte SCHAEFFER[1] et Sophie SCHBATH[2]
[1] INRA-Jouy, Unité Mathématique et Informatique Appliquées
[2] INRA-Jouy, Unité Mathématique, Informatique & Génome

Mai 2008

Table des matières

Chapitre 1

Introduction

1.1 Historique

R est un logiciel libre basé sur le langage de programmation S, langage inventé chez AT & T Bell Laboratories par John Chambers et ses collègues (Becker et al., 1988). L'approche S est vite devenue très populaire dans les milieux académiques du fait de son caractère interactif et flexible. En effet, les utilisateurs peuvent implémenter leurs propres techniques et les diffuser sous forme de bibliothèques. Ainsi, il existe un très grand nombre de bibliothèques, disponibles sur Statlib, un système de distribution de logiciels statistiques (`http://lib.stat.cmv.edu/S/`).

A l'origine, S n'existait pas sous MacIntosh. Or Robert Gentleman et Ross Ihaka à l'Université d'Auckland en Nouvelle Zélande, étaient équipés de Mac. Ils décidèrent alors, au début des années 1990, d'initier un projet pour fournir un "environnement statistique" à leur laboratoire. Comme ils connaissaient S, ils choisirent d'implémenter une syntaxe à la S, et déposèrent leur logiciel sur Statlib. Puis en 1995, R est devenu un logiciel libre, imposant que toute modification reste dans le domaine du libre. R s'est depuis, largement développé, sans doute du fait d'Internet qui a permis à des personnes géographiquement dispersées de collaborer et d'y apporter des améliorations.

1.2 Environnement

R est distribué librement sous les termes de la *GNU, General Public Licence* ; son développement et sa distribution sont assurés par plusieurs statisticiens rassemblés dans le *R Development Core Team* (`http://www.r-project.org/`).
R est écrit principalement en C et parfois en Fortran. Pour les machines Windows, Linux et Macintosh, R est disponible sous forme d'exécutables précompilés. Pour télécharger une version du logiciel, il faut passer par CRAN (*Comprehensive R Archive Network*), un réseau mondial de sites miroirs qui stockent de façon identique les différentes versions, les mises à jour, les bibliothèques et la documentation. Afin d'optimiser le chargement, on choisit généralement le site miroir le plus près de chez soi. En France, il existe 2 sites, l'un à Toulouse, `http://cran.fr.r-project.org/` et l'autre à Paris `http://cran.miroir-francais.fr/`.

Une fois R installé, il suffit de lancer le programme correspondant (typiquement en tapant `R` sous Unix/Linux). Deux fichiers `.Rhistory` et `.RData` sont alors automatiquement créés dans le répertoire courant, et le symbole `>` apparaît indiquant que R est prêt.

Comme on va le voir, R travaille avec des données et des fonctions qui permettent de manipuler ces données. Plusieurs bibliothèques de fonctions sont installées par défaut (`base`, `graphics`, `stats`, etc...). Pour utiliser des bibliothèques spécifiques, à condition que celles-ci aient été chargées, il est nécessaire de les appeler en utilisant la commande `library`. Par exemple, les fonctions de classification développées par Rousseeuw et ses collaborateurs sont contenues dans la bibliothèque `cluster`. Pour les utiliser, il faut donc appeler cette bibliothèque :

```
> library(cluster)
```

Si la bibliothèque appelée n'est pas chargée sur votre machine, R retourne un message d'erreur. Par exemple, si on appelle la la bibliothèque "actuar" qui n'est pas chargée, on obtient :

```
Erreur dans library(actuar) : aucun package nommé 'actuar' n'est trouvé
```

La liste de toutes les bibliothèques existantes est accessible sur le site du CRAN sous la rubrique `packages`.

1.3 Quelques fonctions utiles

- `getwd()` permet de connaître le répertoire dans lequel on travaille.
- `dir()` permet de connaître le contenu du répertoire de travail.
- `setwd("repertoire")` permet de changer de répertoire de travail.
- `source("script.r")` exécute un fichier script.
- `save(monobjet,file="monobjet.rdata")` permet de sauver l'objet `monobjet` sous le format R. Il est aussi possible de sauver plusieurs objets dans le même fichier. Par exemple : `save(objet1,objet2,file="mesobjets.rdata")`.
- `load("mesobjets.rdata")` permet de recharger des objets sauvegardés au cours d'une session précédente.
- `write.table(montableau,file="montableau.txt")` permet de sauvegarder le tableau de données `montableau` sous un format .txt.

Noter que l'appel à une fonction se fait toujours sous la forme *fonction(arguments, options)* et que les paranthèses sont indispensables même sans arguments ni options.

Pour quitter R, il faut utiliser la commande `q()`. R pose alors la question :

```
Save workspace image? [y/n/c]:
```

Si vous répondez `y`, R sauvegarde le fichier `.RData`, qui contient tous les objets créés au cours de la session. Si vous répondez `n`, ces objets sont perdus. Pour continuer la session, il faut répondre par la lettre `c`.

L'historique des commandes est conservé automatiquement dans le fichier `.Rhistory`.

1.4 Le système d'aide

Il existe une aide directement disponible dans R : on y accède en utilisant le `?` suivi du nom de la fonction, ou bien `help(nom.fonction)`. Si la fonction appartient à une bibliothèque particulière, il faut que celle-ci soit chargée.

En réponse à cette commande d'aide, le logiciel affiche une page comportant :

- le nom de la fonction et le nom de la bibliothèque à laquelle elle appartient,
- le titre de la fonction,
- une brève description de la fonction et de son utilisation,
- une description détaillée des arguments de la fonction, et du type d'objet qu'elle retourne,
- des références, des liens vers des fonctions similaires et des exemples.

Nous pouvons, par exemple, obtenir la documentation de la fonction "mean" avec la commande :

```
> help(mean)
```

Cette commande ouvre la page suivante :

```
mean                    package:base                    R Documentation

Arithmetic Mean

Description:

     Generic function for the (trimmed) arithmetic mean.

Usage:

     mean(x, ...)

     ## Default S3 method:
     mean(x, trim = 0, na.rm = FALSE, ...)

Arguments:

       x: An R object.  Currently there are methods for numeric data
          frames, numeric vectors and dates.  A complex vector is
          allowed for 'trim = 0', only.

    trim: the fraction (0 to 0.5) of observations to be trimmed from
          each end of 'x' before the mean is computed.

   na.rm: a logical value indicating whether 'NA' values should be
          stripped before the computation proceeds.

     ...: further arguments passed to or from other methods.

Value:

     For a data frame, a named vector with the appropriate method being
     applied column by column.
```

```
     If 'trim' is zero (the default), the arithmetic mean of the values
     in 'x' is computed, as a numeric or complex vector of length one.
     If any argument is not logical (coerced to numeric), integer,
     numeric or complex, 'NA' is returned, with a warning.

     If 'trim' is non-zero, a symmetrically trimmed mean is computed
     with a fraction of 'trim' observations deleted from each end
     before the mean is computed.

References:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_. Wadsworth & Brooks/Cole.

See Also:

     'weighted.mean', 'mean.POSIXct'

Examples:

     x <- c(0:10, 50)
     xm <- mean(x)
     c(xm, mean(x, trim = 0.10))

     mean(USArrests, trim = 0.2)
```

Il existe également une aide au format html permettant l'utilisation d'un moteur de recherche ; on y accède en utilisant `help.start()` qui ouvre la page d'accueil de l'aide de R, dans la fenêtre d'un navigateur interne.

Il existe aussi des commandes permettant une investigation de la documentation à partir de mots-clés :

- la fonction `apropos("keyword")` retourne la liste de toutes les fonctions qui contiennent dans leur nom, le mot-clé recherché.
- la fonction `help.search("keyword")` retourne la liste de toutes les fonctions qui contiennent dans leur nom ou dans leur titre, le mot-clé recherché.

La fonction `args` donne en général la liste des arguments d'une fonction. Par exemple :

```
> args(sort)
```

```
function (x, decreasing = FALSE, ...)
NULL
```

Un conseil : ne pas hésiter à consulter l'aide d'une fonction la première fois qu'on l'utilise car on peut y découvrir des options intéressantes.

Chapitre 2

Langage de programmation

R est un langage interprété et non compilé, autrement dit les commandes tapées au clavier sont directement exécutées lorsque l'on appuie sur la touche "Entrée". On pourra soit taper une commande par ligne, soit taper plusieurs commandes séparées par le symbole ' ; ' sur une même ligne. Une commande peut s'écrire sur plusieurs lignes, auquel cas R matérialise le début de la 2ème ligne d'instructions par le symbole '+'.

R est un langage simple et intuitif dans lequel chaque chose est un objet. Utiliser R revient donc à créer et manipuler des objets. On distingue deux types d'objets, les données et les fonctions.

2.1 Les données

Les données se caractérisent par un type (scalaire, vecteur, tableau, ...), un mode (numérique, caractère, logique), une taille, une valeur et un nom.

2.1.1 Caractéristiques d'un objet

Type Les données manipulées par R peuvent être de différents types :
- vecteur : suite d'éléments (une dimension),
- tableau : tableau à k dimensions,
- matrice : tableau à 2 dimensions,
- facteur : suite d'éléments catégoriels avec les différents niveaux possibles,
- tableau de données : ensemble de vecteurs de même longueur,
- séries temporelles : jeu de données de type séries temporelles (avec fréquence, dates),
- liste : liste d'objets.

Mode Les éléments stockés dans un objet peuvent être de 4 modes différents :
- numérique,
- caractère (entre guillemets),
- complexe,
- logique (`TRUE` ou `FALSE`),

Pour connaître le mode d'un objet `x`, on utilise la fonction `mode` :

```
> x <- 0
> mode(x)
```

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

```
> titre <- "coucou"
> mode(titre)
```

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

Seuls les tableaux de données et les listes peuvent contenir des éléments de modes différents.

Taille La taille d'un objet est le nombre d'éléments qu'il contient. Elle s'obtient à l'aide de la fonction `length` :

```
> length(x)
```

```
[1] 1
```

Nom Le nom d'un objet doit obligatoirement commencer par une lettre et ne peut comporter que des lettres, des chiffres, des espaces soulignés et des points.

Valeur Quelque soit le mode, une valeur manquante est représentée par `NA`.

Voici quelques remarques sur les valeurs numériques.

- R représente correctement les valeurs infinies avec `Inf` et `-Inf` :

  ```
  > x <- 0
  > 1/x
  ```

  ```
  [1] Inf
  ```

- ainsi que les valeurs qui ne sont pas des nombres avec `NaN` :

  ```
  > y <- 1/0
  > y - y
  ```

  ```
  [1] NaN
  ```

- La notation exponentielle est utilisée pour les grandes valeurs :

  ```
  > x <- 100 * 5.3e+12
  > x
  ```

  ```
  [1] 5.3e+14
  ```

  ```
  > 1/x
  ```

  ```
  [1] 1.886792e-15
  ```

2.1.2 Créer des objets

L'opérateur d'assignation `<-` est une façon implicite de créer un objet ; le mode et le type de l'objet sont en effet automatiquement déterminés.

```
> x <- 2
> y <- x
> y

[1] 2
```

Création d'un vecteur L'opérateur d'assemblage `c()` est aussi une façon implicite pour créer un vecteur :

```
> x <- c(1, 3, 5, 7, 9)
> x

[1] 1 3 5 7 9

> length(x)

[1] 5

> y <- c(x, 11, 13)
> y

[1]  1  3  5  7  9 11 13
```

La création d'un vecteur pourra aussi être faite via la génération de suites (cf. page 16).

On peut aussi créer un vecteur de façon explicite en spécifiant simplement son mode et sa longueur ; la valeur des éléments du vecteur sera alors initialisée selon le mode : 0 pour numérique, `FALSE` pour logique, `""` pour caractère.

```
> x <- vector(mode = "numeric", length = 5)
> x

[1] 0 0 0 0 0

> y <- vector(mode = "logical", length = 4)
> y

[1] FALSE FALSE FALSE FALSE

> z <- vector(mode = "character", length = 3)
```

Les éléments du vecteur seront ensuite modifiés en utilisation l'indexation (cf. page 18).

Création d'une matrice Une matrice est créée à l'aide de la fonction `matrix`. On spécifie le nombre de lignes et le nombre de colonnes via les arguments `nrow` et `ncol`, et les valeurs des éléments de la matrice via l'argument (vectoriel) `data`. Par exemple, pour initialiser une matrice (2×3) à 0, on utilisera :

```
> x <- matrix(data = 0, nrow = 2, ncol = 3)
> x
```

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

Si l'on veut remplir la matrice avec des valeurs différentes, les valeurs fournies par le vecteur `data` rempliront alors par défaut les colonnes successives :

```
> x <- matrix(data = c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
> x
```

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

Pour remplir la matrice plutôt par ligne, il suffit de spécifier l'argument `byrow=TRUE` :

```
> x <- matrix(data = c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3, byrow = TRUE)
> x
```

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

On peut aussi rajouter des lignes (fonction `rbind`) ou des colonnes (fonction `cbind`) à une matrice existante (voire concaténer des matrices). Voici un exemple :

```
> cbind(x, c(7, 7))
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    7
[2,]    4    5    6    7
```

```
> rbind(x, c(8, 8, 8))
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    8    8    8
```

```
> cbind(x, x)
```

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

On a la possibilité d'attribuer des noms aux lignes et aux colonnes d'une matrice via l'option `dimnames` (ces noms doivent être de type *character*).

Pour connaître les dimensions d'une matrice, on utilise la fonction `dim` (la fonction `length` retourne le nombre d'éléments de la matrice :

```
> dim(x)
```

```
[1] 2 3
```

```
> length(x)
```

```
[1] 6
```

Création d'un tableau de données Un tableau de données est une collection de vecteurs de même longueur. On peut créer un tableau de données en utilisant la fonction `data.frame`. Par exemple :

```
> seq1 <- c(0.2, 0.3, 0.4, 0.1)
> seq2 <- c(0.25, 0.35, 0.15, 0.25)
> df <- data.frame(seq1, seq2)
> df
```

```
  seq1 seq2
1  0.2 0.25
2  0.3 0.35
3  0.4 0.15
4  0.1 0.25
```

Il est possible de donner des noms aux lignes avec l'option `row.names` qui doit fournir un vecteur de mode caractère et de longueur égale au nombre de lignes du tableau de données. Par exemple, dans l'exemple ci-dessus le tableau de données correspond aux compositions en nucléotides de deux séquences, on pourrait donc écrire :

```
> seq1 <- c(0.2, 0.3, 0.4, 0.1)
> seq2 <- c(0.25, 0.35, 0.15, 0.25)
> df <- data.frame(seq1, seq2, row.names = c("A", "C", "G", "T"))
> df
```

```
  seq1 seq2
A  0.2 0.25
C  0.3 0.35
G  0.4 0.15
T  0.1 0.25
```

On verra au paragraphe 2.3.1 page 26 que la fonction `read.table` qui permet de lire des données dans un fichier crée implicitement un objet de type tableau de données.

Les fonctions `cbind` et `rbind` que nous avons vu page 11 pour les matrices s'appliquent aussi pour des tableaux de données.

On notera aussi les fonctions `ncol` et `nrow` qui renvoient le nombre de colonnes et le nombre de lignes d'un tableau de données (ou d'une matrice).

Création d'une liste Une liste est une collection d'objets (non nécessairement de même types) ; elle se crée via la fonction `list` :

```
> seq1 <- c(0.2, 0.3, 0.4, 0.1)
> L1 <- 58402
> res <- list(L1, seq1)
> res


[[1]]
[1] 58402

[[2]]
[1] 0.2 0.3 0.4 0.1
```

Pour rallonger une liste d'éléments, on utilise l'opérateur d'assemblage `c()` :

```
> espece1 <- "phage"
> c(espece1, res)


[[1]]
[1] "phage"

[[2]]
[1] 58402

[[3]]
[1] 0.2 0.3 0.4 0.1
```

Contrairement à `data.frame`, la fonction `list` ne conserve pas les noms des objets ; autrement dit, la longueur de la séquence (`L1`) est le premier élément de la liste, et la composition en nucléotides est le 2ème élément. On peut attribuer des étiquettes **différentes** plus explicites aux différents éléments d'une liste, ce qui sera très pratique pour y accéder (sans avoir à se souvenir dans quel ordre ils sont rangés dans la liste !). Par exemple,

```
> seq1 <- c(0.2, 0.3, 0.4, 0.1)
> L1 <- 58402
> res <- list(longueur = L1, composition = seq1)
> res


$longueur
[1] 58402

$composition
[1] 0.2 0.3 0.4 0.1
```

On verra page 21 que pour accéder aux champs `longueur` (resp. `composition`) de la liste `res`, on utilisera l'opérateur `$` : `res$longueur` (resp. `res$composition`).

Création d'une chaîne de caractère Une fonction très utile pour créer une chaîne de caractère, outre l'opérateur d'assignation `<-`, est `paste` qui permet de concaténer des chaînes de caractères mais aussi des objets automatiquement convertis en chaîne de caractères. Ceci est particulièrement utile par exemple lorsque l'on veut créer un fichier de sortie dont le nom est "dynamique". Voici quelques exemples :

```
> nom <- paste("filename", ".", "txt", sep = "")
> nom


[1] "filename.txt"


> prefix <- "resultats."
> i <- 5
> nom <- paste(prefix, i, ".ps", sep = "")
> nom


[1] "resultats.5.ps"
```

Par défaut `sep=" "` et indique le séparateur de concaténation.

2.1.3 Convertir un objet

On peut vouloir convertir un objet d'un type à un autre. Voici quelques exemples :

– numérique → caractère

```
> i

[1] 5

> as.character(i)

[1] "5"
```

– numérique → logique

```
> x <- 0:3
> as.logical(x)
```

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

– logique → numérique

```
> ok <- c(TRUE, FALSE, FALSE, TRUE)
> as.numeric(ok)
```

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

2.1.4 Lister et supprimer des objets en mémoire

Quand R est utilisé, les variables, les objets, les fonctions, ... sont stockées dans la mémoire vive de l'ordinateur. On peut afficher la liste de tous les objets en mémoire en tapant la commande :

```
> ls()
```

```
 [1] "df"      "espece1" "i"       "L1"      "nom"     "ok"      "prefix"
 [8] "res"     "seq1"    "seq2"    "titre"   "x"       "y"       "z"
```

Si l'on souhaite se restreindre aux objets dont le nom contient une chaîne de caractère, il suffit de spécifier celle-ci via l'option `pattern`.
La commande `ls.str()` va quant à elle lister tous les objets en mémoire mais aussi en donner leur détail (type, mode, valeur, etc.).

```
> ls.str()
```

```
df : 'data.frame':        4 obs. of  2 variables:
 $ seq1: num  0.2 0.3 0.4 0.1
 $ seq2: num  0.25 0.35 0.15 0.25
espece1 :  chr "phage"
i :  num 5
L1 :  num 58402
nom :  chr "resultats.5.ps"
ok :  logi [1:4]  TRUE FALSE FALSE  TRUE
prefix :  chr "resultats."
res : List of 2
 $ longueur   : num 58402
 $ composition: num [1:4] 0.2 0.3 0.4 0.1
seq1 :  num [1:4] 0.2 0.3 0.4 0.1
seq2 :  num [1:4] 0.25 0.35 0.15 0.25
titre :  chr "coucou"
x :  int [1:4] 0 1 2 3
y :  logi [1:4] FALSE FALSE FALSE FALSE
z :  chr [1:3] "" "" ""
```

Pour effacer des objets de la mémoire, on utilise la fonction `rm` :

```
> rm(x)
```

Si l'on veut maintenant accéder à `x`, le message suivant apparaît "`Error : Object "x" not found`".
Pour effacer tous les objets de la mémoire, on tape `rm(list=ls())`.

2.1.5 Générer des données

Suites régulières Une suite d'entiers consécutifs peut s'obtenir avec l'opérateur ' `:`'; par exemple

```
> 5:12
```

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

Un vecteur est ainsi crée. Cet opérateur est prioritaire sur les opérations arithmétiques :

```
> n <- 10
> 1:(n - 2)
```

```
[1] 1 2 3 4 5 6 7 8
```

```
> 1:n - 2
```

```
 [1] -1  0  1  2  3  4  5  6  7  8
```

La fonction `rep` crée un vecteur dont les éléments sont identiques ou répétés :

```
> rep(1, times = 10)
```

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

```
> rep(1:5, times = 3)
```

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

```
> rep(1:5, each = 2)
```

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

La fonction `seq` peut générer des suites régulières de nombres de deux manières différentes :

- soit en lui spécifiant, le début, la fin, puis le pas (argument `by`)

  ```
  > seq(from = 2, to = 20, by = 2)
  ```

  ```
   [1]  2  4  6  8 10 12 14 16 18 20
  ```

- soit en lui spécifiant le début, la fin et le nombre d'éléments (argument `length`)

  ```
  > seq(1, 5, length = 10)
  ```

  ```
  [1] 1.000000 1.444444 1.888889 2.333333 2.777778 3.222222 3.666667 4.111111
  [9] 4.555556 5.000000
  ```

Variables aléatoires Il est très utile en statistique de pouvoir générer des variables aléatoires selon diverses lois de probabilité. R peut le faire pour un grand nombre de lois via les fonctions de la forme `r`*func*`(n,p1,p2,...)` où *func* indique la loi de probabilité, `n` est le nombre de variables à générer et `p1`, `p2`, ... sont les paramètres de la loi. En voici quelques exemples :

Loi	Fonction R
gaussienne	`rnorm(n, mean=0, sd=1)`
uniforme	`runif(n, min=0, max=1)`
Poisson	`rpois(n,lambda)`
exponentielle	`rexp(n, rate=1)`
binomiale	`rbinom(n,size,prob)`

```
> rnorm(10)
```

```
 [1] -2.33495809 -0.38774695  2.30444595  0.98042579 -0.01286859 -0.48970664
 [7] -1.33955227  0.34285615 -0.10031703 -0.63011227
```

```
> runif(10)
```

```
 [1] 0.1935150 0.8751255 0.1044929 0.5905437 0.7564251 0.2597194 0.1207282
 [8] 0.9478013 0.3995222 0.6379168
```

```
> rpois(10, lambda = 4)
```

```
 [1] 7 5 3 3 2 3 4 2 2 8
```

Remarque : les fonctions de la forme `r`*func* ont toutes des petites soeurs de la forme
- `p`*func*`(q,p1,p2,...)` : pour la probabilité cumulée jusqu'à `q`,
- `q`*func*`(p,p1,p2,...)` : pour le quantile associé à la probabilité cumulée `p`,
- `d`*func*`(x,p1,p2,...)` : pour la densité de probabilité en `x`.

Cela permet ainsi de tracer des densités et de connaître la probabilité d'être supérieur ou égal à q sous une loi donnée. Si on prend l'exemple de la loi normale de moyenne 0 et de variance 1 avec $q = 2.5$ (cf. Figure2.1), on a :

```
> 1 - pnorm(2.5)
```

```
[1] 0.006209665
```

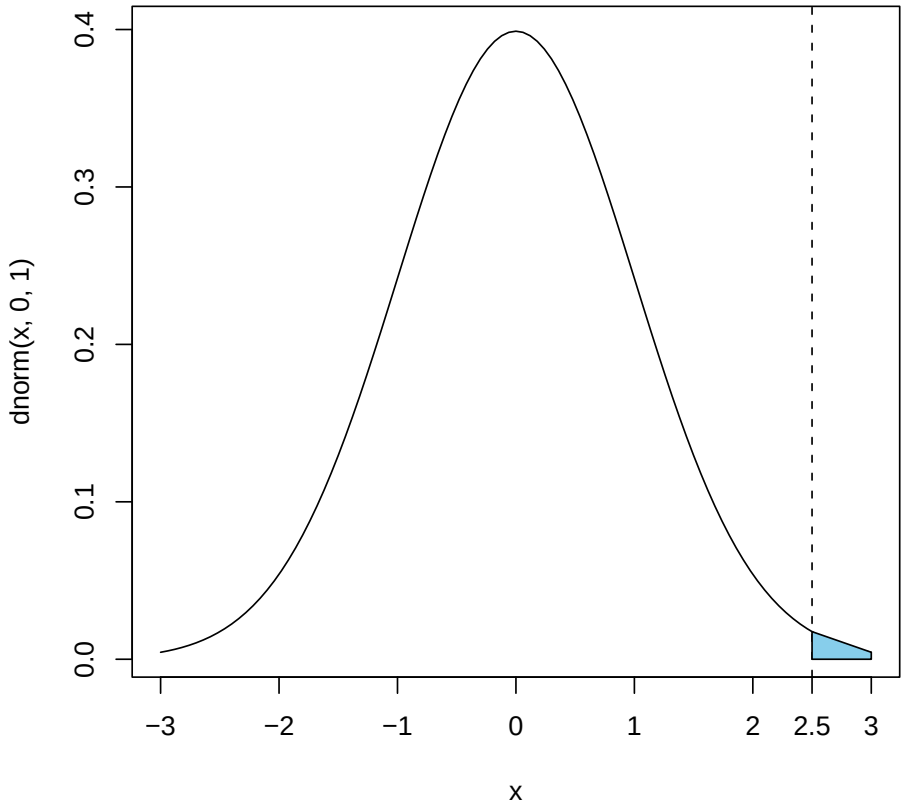

FIG. 2.1 – Densité de la loi normale centrée réduite et probabilité de dépasser 2.5

2.1.6 Accéder aux valeurs d'un objet

L'indexation (opérateur `[]`) permet d'accéder de façon sélective aux éléments d'un objet.

Vecteurs Pour accéder au 2ème élément d'un vecteur `x`, il suffit de taper `x[2]`. Cela permet aussi d'aller modifier spécifiquement certains éléments :

```
> x <- 1:10
> x[2]
```

```
[1] 2
```

```
> x[1] <- 10
> x
```

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

On peut aussi accéder à plusieurs éléments d'un vecteur et les modifier tous ensemble :

```
> x <- seq(2, 20, by = 2)
> x[1:5]

[1]  2  4  6  8 10


> x[1:5] <- rep(0, 5)
> x

 [1]  0  0  0  0  0 12 14 16 18 20
```

La commande `x[1 :5]` est équivalente à `x[c(1,2,3,4,5)]`.

On peut accéder aux éléments dont la valeur vérifie une expression de comparaison. Par exemple, aux valeurs positives d'un vecteur :

```
> x <- runif(10, min = -1, max = 1)
> x

 [1]  0.99738228 -0.38165478  0.18856005  0.50197410 -0.34154366  0.03917865
 [7]  0.93255402  0.82991814  0.45826231  0.62460486


> x >= 0

 [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE


> x[x >= 0]

[1] 0.99738228 0.18856005 0.50197410 0.03917865 0.93255402 0.82991814 0.45826231
[8] 0.62460486
```

On peut supprimer un ou plusieurs éléments d'un vecteur en utilisant des indices négatifs :

```
> x <- (1:5)^2
> x[-1]

[1]  4  9 16 25


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

[1]  1  9 25
```

Matrice On peut accéder à un élément, à une colonne, à une ligne, à une sous-matrice d'une matrice de la façon suivante :

```
> x <- matrix(data = c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
> x
```

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

```
> x[1, 3]
```

```
[1] 5
```

Noter que l'indice de ligne précède toujours l'indice de colonne.

```
> x[2, ]
```

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

```
> x[, 3]
```

```
[1] 5 6
```

```
> x[1:2, 1:2]
```

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

Noter que `x[,3]` est un vecteur car, par défaut, R retourne un objet de la plus petite taille. Pour conserver le format d'origine, ici une matrice, il suffit de spécifier l'option `drop=FALSE`. Par exemple

```
> x[, 3, drop = FALSE]
```

```
     [,1]
[1,]    5
[2,]    6
```

La modification des éléments d'une matrice se fait comme pour les vecteurs. On peut également supprimer une ou plusieurs lignes ou colonnes d'une matrice en utilisant des indices négatifs. Par exemple, pour supprimer la 2ème colonne d'une matrice :

```
> x <- matrix(data = c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
> x
```

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

```
> x[, -2]
```

```
     [,1] [,2]
[1,]    1    5
[2,]    2    6
```

Liste Pour accéder aux objets d'une liste on utilisera soit les crochets simples (dans ce cas une liste est retournée), soit les crochets doubles (dans ce cas l'objet est extrait) :

```
> seq1 <- c(0.2, 0.3, 0.4, 0.1)
> L1 <- 58402
> titre <- "phage"
> res <- list(titre, L1, seq1)
> res[1:2]
```

```
[[1]]
[1] "phage"

[[2]]
[1] 58402
```

```
> res[[3]]
```

```
[1] 0.2 0.3 0.4 0.1
```

Si l'on a donné des étiquettes aux différents objets de la liste, alors on peut extraire ces objets en utilisant l'opérateur `$` :

```
> seq1 <- c(0.2, 0.3, 0.4, 0.1)
> L1 <- 58402
> titre <- "phage"
> res <- list(espece = titre, longueur = L1, composition = seq1)
> res$espece
```

```
[1] "phage"
```

```
> res$longueur

[1] 58402

> res$composition

[1] 0.2 0.3 0.4 0.1
```

Tableau de données Pour récupérer une colonne d'un tableau de données, il y a plusieurs façon selon que l'on utilise le nom des colonnes ou non. Reprenons le tableau de données `df` de la page 13 pour extraire la première colonne :

```
> df

  seq1 seq2
A  0.2 0.25
C  0.3 0.35
G  0.4 0.15
T  0.1 0.25
```

Si on n'utilise pas le nom de la colonne :
- `df[1]` retourne un tableau de données (à une colonne),
- `df[[1]]` retourne un vecteur,
- `df[,1]` retourne un vecteur.

Si on utilise le nom `seq1` de la colonne :
- `df["seq1"]` retourne un tableau de données (à une colonne),
- `df[["seq1"]]` retourne un vecteur,
- `df$seq1` retourne un vecteur.

Pour récupérer une ligne (la 1ère par exemple), on utilisera `df[1,]` mais cela retourne un tableau de données.

2.2 Opérateurs et calcul arithmétique

2.2.1 Opérateurs arithmétiques

Les opérateurs arithmétiques agissent aussi bien sur des variables numériques que logiques (dans ce dernier cas, les valeurs logiques sont converties en valeurs numériques : 0 pour `FALSE` et 1 pour `TRUE`). Mis à part les opérateurs classiques, addition (`+`), soustraction (`-`), multiplication (`*`) et division (`/`), on notera les 3 autres opérateurs suivants :
- `^` : puissance
- `%%` : modulo
- `%/%` : division entière

Ces opérateurs peuvent aussi s'appliquer à des vecteurs (et à des matrices) ; dans ce cas, attention, l'opération est effectuée composante par composante :

```
> x <- 1:10
> x^2

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

> y <- 1:10
> x + y

 [1]  2  4  6  8 10 12 14 16 18 20

> x * y

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

Si les deux vecteurs n'ont pas la même taille le plus petit sera "recyclé" autant de fois que nécessaire (avec un *warning message*)

```
> x <- 1:10
> y <- 1:3
> x + y

 [1]  2  4  6  5  7  9  8 10 12 11
```

2.2.2 Opérateurs de comparaison

Les opérateurs de comparaison s'appliquent à deux objets de n'importe quel mode et retournent une valeur logique. En fait les opérateurs opèrent sur chaque élément des objets comparés et retourne donc un objet logique de même taille :

```
> x <- 1:10
> y <- (-3:6)^2
> x

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

> y

 [1]  9  4  1  0  1  4  9 16 25 36

> x <= y
```

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

L'opérateur `==` (resp `!=`) vérifie l'égalité (resp. la différence) de deux objets, élément par élément ; il retourne donc un vecteur logique :

```
> x <- seq(2, 10, by = 2)
> y <- 2 * (1:5)
> x == y
```

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

Pour effectuer une comparaison globale d'objets, la fonction équivalente est `identical(x,y)` :

```
> x <- seq(2, 10, by = 2)
> y <- 2 * (1:5)
> identical(x, y)
```

```
[1] TRUE
```

Cette fonction (ainsi que l'opérateur `==`) compare la représentation interne des données et retourne `TRUE` si les objets sont **strictement identiques**. On se méfiera ainsi des erreurs numériques dues aux calculs numériques sur ordinateur :

```
> (1.1 - 0.2) == 0.9
```

```
[1] FALSE
```

Pour y remédier, la fonction `all.equal` compare l'égalité approximative, c'est-à-dire avec un seuil de tolérance (qui par défaut dépend des paramètres de la machine), entre deux objets :

```
> all.equal(1.1 - 0.2, 0.9)
```

```
[1] TRUE
```

2.2.3 Opérateurs logiques

Les opérateurs logiques s'appliquent à un ou deux objets de mode logique et retournent une ou plusieurs valeurs logiques :

- `!x` : NON logique
- `x&y` : ET logique
- `x&&y` : idem mais sur les premiers éléments de `x` et `y`
- `x|y` : OU logique
- `x||y` : idem mais sur les premiers éléments de `x` et `y`
- `xor(x,y)` : OU exclusif

2.2.4 Fonctions mathématiques simples

Il existe un nombre très important de fonctions pour manipuler des données numériques (généralement sous la forme de vecteurs). Outre les fonctions mathématiques de base du type `log`, `exp`, `cos`, `abs`, `sqrt` etc. en voici quelques-unes assez courantes :

- `sum(x)`, `prod(x)` : somme, produit des éléments de `x`,
- `min(x)`, `max(x)` : minimum, maximum des éléments de `x`,
- `which.min(x)`, `which.max(x)` : indice du min, max des éléments de `x`,
- `length(x)` : nombre d'éléments de `x`,
- `rev(x)` : inverse l'ordre des éléments de `x`,
- `sort(x)` : trie les éléments de `x` dans l'ordre croissant,
- `choose(n,k)` : combinaisons de k éléments parmi n.

Par exemple

```
> x <- choose(6, 0:6)
> x

[1]  1  6 15 20 15  6  1

> sum(x)

[1] 64

> max(x)

[1] 20

> which.max(x)

[1] 4

> sort(x, decreasing = TRUE)

[1] 20 15 15  6  6  1  1
```

En voici d'autres à connotation statistique :

- `mean(x)` : moyenne des éléments de `x`,
- `median(x)` : médiane des éléments de `x`,
- `var(x)` : variance des éléments de `x`,
- `cov(x,y)` : covariance entre `x` et `y`,
- `cor(x,y)` : corrélation entre `x` et `y`,
- `sample(x,n)` : tirage aléatoire sans remise de `n` éléments parmi ceux de `x`.

2.2.5 Calcul matriciel

R offre des facilités pour le calcul matriciel. En voici quelques unes :
- `x%*%y` : produit de deux matrices,
- `t(x)` : transposée de `x`,
- `diag(x)` : extrait la diagonale d'une matrice `x` (dans ce cas renvoie un vecteur) ou crée une matrice diagonale de diagonale `x`,
- `solve(x)` : matrice inverse de `x`
- `solve(A,b)` : résout le système linéaire $Ax = b$,
- `eigen(x)` : calcule les valeurs et vecteurs propres de `x`.

2.3 Entrées et sorties

Pour les lectures et écritures de données dans des fichiers, R utilise le répertoire de travail. Il est donc nécessaire de préciser le chemin d'accès au fichier si celui-ci n'est pas dans le répertoire de travail.

2.3.1 Lire des données dans un fichier

Nous nous intéresserons ici à la lecture de fichiers ASCII. Les deux principales fonctions sont `read.table` et `scan`.

`read.table` Cette fonction crée un tableau de données directement à partir des données disposées dans un fichier sous la forme d'un tableau. Par exemple, prenons le fichier `composition3mer.tab` présent dans le répertoire `data`. Celui-ci est composé de 87 lignes et 66 colonnes : la première ligne contient la légende des colonnes et les colonnes correspondent successivement au nom d'une séquence d'ADN, à sa longueur puis au comptage des 64 trinucléotides de cette séquence. La commande suivante :

```
> mydata <- read.table("data/composition3mer.tab", header = TRUE)
```

crée donc un tableau de données nommé `mydata` dont les colonnes (variables) sont nommées selon la première ligne du fichier (`header=TRUE`). Ces noms peuvent s'obtenir en utilisant la fonction `names` :

```
> names(mydata)
```

```
 [1] "seq"  "long" "aaa"  "aac"  "aag"  "aat"  "aca"  "acc"  "acg"  "act"
[11] "aga"  "agc"  "agg"  "agt"  "ata"  "atc"  "atg"  "att"  "caa"  "cac"
[21] "cag"  "cat"  "cca"  "ccc"  "ccg"  "cct"  "cga"  "cgc"  "cgg"  "cgt"
[31] "cta"  "ctc"  "ctg"  "ctt"  "gaa"  "gac"  "gag"  "gat"  "gca"  "gcc"
[41] "gcg"  "gct"  "gga"  "ggc"  "ggg"  "ggt"  "gta"  "gtc"  "gtg"  "gtt"
[51] "taa"  "tac"  "tag"  "tat"  "tca"  "tcc"  "tcg"  "tct"  "tga"  "tgc"
[61] "tgg"  "tgt"  "tta"  "ttc"  "ttg"  "ttt"
```

Comme on l'a vu page 22, on accède à la colonne `long` du tableau de données `mydata` en tapant

```
> mydata$long

 [1]  23814  20869  31787  40449  39630  36895  38179  41172  24655  62730
[11]  42271 288050  75898  77670  14462  16067  28649  20531  26200  26476
[21]  18098  13321   7598   3444   7830  58498  21243  35422  24554  11040
[31]  12975   4967  28930 130989  23486  46867  36358  33303  33452  39521
[41]  50060 132678 155300 410554  41229 191346 365425   9689  12205   5350
[51]   6959   3373  28798  26111  29000  25960   5740   5705  28337  21566
[61]   7192  20417  13638   3619  13015  28097  30889  40047  32308  33641
[71]  35450  15465  14796  15135  15330  17385  15684  16473  17663  75294
[81]  23591  17195  27241  18823  20933  13669
```

Le résultat est un vecteur, tout comme `mydata[,2]`. Par contre `mydata["long"]` renvoie un tableau de données.

Par défaut, `header=FALSE`, ce qui implique que chaque colonne est nommée par défaut `V1`, `V2`, etc. On peut par la suite spécifier nos propres noms de colonnes en utilisant l'option `col.names`. De la même façon, par défaut les lignes sont numérotées 1, 2 etc. et on peut spécifier nos propres noms de ligne via l'option `row.names`.

Dans notre exemple, la première colonne (le nom des séquences) pourrait servir de noms pour les lignes. On aurait pu alors sauvegarder la première colonne de `mydata` (vecteur `noms.seq`), la supprimer du tableau de données (on obtient un nouveau tableau de données `mynewdata`) puis nommer les lignes du nouveau tableau avec le vecteur `noms.seq` à l'aide de la fonction `rownames` :

```
> noms.seq <- mydata[, 1]
> mynewdata <- mydata[-1]
> rownames(mynewdata) <- noms.seq
```

Si les lignes du fichier de données n'ont pas toutes le même nombre de valeurs, un message d'erreur apparaîtra aussitôt après la commande `read.table`.

Par défaut, le symbole séparant les décimales est le point, mais on peut le modifier via l'option `dec` ; par exemple, on pourra spécifier `dec=","` si besoin.

`scan` Une autre possibilité pour lire un fichier de données est d'utiliser la fonction `scan`. La principale différence est que `scan` crée un vecteur numérique par défaut (et non un tableau de données), ou une liste de vecteurs si on lui spécifie leurs modes via l'option `what`. Cela suppose que chaque colonne contienne des valeurs de même mode. Par exemple, supposons que le fichier `shortdata` (répertoire `data`) ne contienne que les colonnes 1, 2 et 3 du fichier `data/composition3mer.tab` (les noms de séquences, leur longueur et le nombre de aaa) pour les 10 premières séquences (on verra page 30 comment a été crée ce fichier). Alors en tapant

```
> myshortdata <- scan("data/shortdata", what = list("", 0, 0),
+     skip = 1)
> myshortdata
```

```
[[1]]
 [1] "virus01.sq" "virus02.sq" "virus03.sq" "virus04.sq" "virus05.sq"
 [6] "virus06.sq" "virus07.sq" "virus08.sq" "virus09.sq" "virus10.sq"

[[2]]
 [1] 23814 20869 31787 40449 39630 36895 38179 41172 24655 62730

[[3]]
 [1]  728  760  833 1215 1213 1033 1155 1488 1048 1875
```

on obtient une liste (`myshortdata`) formée de 3 vecteurs, le premier de mode caractère et les deux derniers numériques. L'option `skip=1` permet de spécifier le nombre de lignes du fichier à sauter avant d'enregistrer les données (en général des commentaires).

On peut spécifier les noms (ou étiquettes) de ces 3 objets :

```
> myshortdata <- scan("data/shortdata", what = list(seq = "", long = 0,
+     aaa = 0), skip = 1)
> myshortdata


$seq
 [1] "virus01.sq" "virus02.sq" "virus03.sq" "virus04.sq" "virus05.sq"
 [6] "virus06.sq" "virus07.sq" "virus08.sq" "virus09.sq" "virus10.sq"

$long
 [1] 23814 20869 31787 40449 39630 36895 38179 41172 24655 62730

$aaa
 [1]  728  760  833 1215 1213 1033 1155 1488 1048 1875
```

En pratique, si le fichier contient une unique colonne, on privilégiera `scan` à `read.table` en spécifiant `what=character()` ou `what=logical()` si besoin.

2.3.2 Afficher des résultats

Affichage à l'écran On a vu que la façon la plus simple d'afficher le contenu d'un objet à l'écran était de taper son nom suivi de la touche "Entrée". C'est équivalent à écrire explicitement `print(x)` :

```
> x <- choose(6, 0:6)
> print(x)


[1]  1  6 15 20 15  6  1
```

On utilisera `print(x)` à l'intérieur d'une boucle, d'une fonction ou d'un programme (§ 2.4).

Si l'on veut un "bel" affichage, on peut alors utiliser la fonction `cat` qui va convertir le contenu des objets en chaîne de caractères (donc sous la forme d'un vecteur), concaténer autant de chaînes de caractères que demandées et imprimer la chaîne résultante :

```
> x <- choose(6, 0:6)
> cat("Les combinaisons parmi 6 elements sont : ", x, "\n")

Les combinaisons parmi 6 elements sont :  1 6 15 20 15 6 1
```

La chaîne de caractères `"\n"` signifie un retour à la ligne. Par défaut, la concaténation des chaînes de caractères se fait avec le séparateur `sep=" "` .

La fonction `cat` est donc particulièrement adaptée à l'affichage de vecteurs (éventuellement de longueur 1). Pour afficher un tableau (matrice ou tableau de données) on utilisera `print`.

Affichage dans un fichier Si l'on veut imprimer des résultats dans un fichier, on peut procéder de deux manières :

- Soit rediriger la sortie standard (c'est-à-dire la console) dans un fichier via la fonction `sink(file="file.res", append=TRUE)` avant d'afficher ; si l'option `append` n'est pas spécifiée (`FALSE` par défaut), l'écriture écrase l'ancien contenu du fichier. Ne pas oublier de rebasculer la sortie standard après écriture en faisant `sink()`.

  ```
  > sink("toto")
  > cat("Combinaisons parmi 6 elements : \n")
  > print(x)
  > sink()
  ```

  Le fichier *toto* a été créé et contient :

  ```
  Combinaisons parmi 6 elements :
  [1]  1  6 15 20 15  6  1
  ```

- Soit en utilisant l'option `file` (et `append`) de la fonction `cat`.

  ```
  > cat(file = "toto", "x=", x, append = TRUE)
  ```

  Le fichier *toto* contient maintenant :

  ```
  Combinaisons parmi 6 elements :
  [1]  1  6 15 20 15  6  1
  x= 1 6 15 20 15 6 1
  ```

2.3.3 Sauvegarder des données

La sauvegarde des données n'est pas automatique, toutefois, on a la possibilité de sauvegarder l'ensemble des objets en mémoire avant de quitter une session R. Pour cela, il suffit de répondre 'y' à la question posée suite à la commande `q()` : "`Save workspace image? [y/n/c]`". Les objets sont alors sauvegardés dans le répertoire `.RData` du répertoire de travail et seront disponibles lors de la prochaine session R.

On peut néanmoins sauvegarder explicitement des objets R dans un fichier en utilisant la fonction `save(mynewdata,file="mynewdata.Rdata")` (à noter : le fichier mynewdata.Rdata n'est pas lisible à l'oeil). Pour recharger l'objet enregistré lors d'une nouvelle session, il suffira de faire `load("mynewdata.Rdata")`.

Si l'on veut plutôt écrire le contenu d'un objet dans un fichier (pour former un nouveau fichier de données par exemple), on pourra utiliser la fonction `write.table` qui stocke les objets sous la forme de tableaux de données (le fichier créé pourra alors être relu avec `read.table`).

Par exemple, le fichier `shortdata` qui a été utilisé page 27 a été obtenu de la façon suivante à partir du tableau de données `mydata` :

```
> write.table(mydata[1:10, 1:3], file = "shortdata", row.names = FALSE,
+     quote = FALSE)
```

L'option `row.names` indique si le nom des lignes doit être écrit dans le fichier (`col.names` pour les colonnes), tandis que l'option `quote` indique si les chaînes de caractères sont écrites avec leurs guillemets.

2.4 Programmation

2.4.1 Boucles

Boucle `for` Voici un exemple tout simple pour illustrer la syntaxe d'une boucle `for` :

```
> x <- rep(0, 10)
> y <- x
> for (i in 1:10) {
+     x[i] <- i
+     y[i] <- i^2
+ }
> x

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

> y

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

Les accolades ne sont pas nécessaires s'il n'y a qu'une instruction.

On verra au § 2.4.3 que dans bon nombre de cas, les boucles `for` peuvent être évitées du fait que R travaille vectoriellement.

Boucle `while` Voici maintenant un exemple pour illustrer la syntaxe d'une boucle `while` :

```
> proba <- dpois(0:100, lambda = 20)
> quantile <- 0
> queue <- 1
> while (queue > 0.05) {
+     queue <- queue - proba[quantile + 1]
+     quantile <- quantile + 1
+ }
> quantile

[1] 29

> queue

[1] 0.03433352
```

2.4.2 Tests

La syntaxe d'une boucle *si alors sinon* est la suivante :

```
> x <- rpois(1, 20)
> if (x%%2 == 0) {
+     pair <- TRUE
+ } else {
+     pair <- FALSE
+ }
> x
```

```
[1] 15
```

```
> pair
```

```
[1] FALSE
```

L'instruction *sinon* n'est pas obligatoire mais si elle existe alors le mot-clé `else` doit être sur la même ligne que l'accolade fermante du *si*.
On peut imbriquer plusieurs boucles *si alors sinon si alors sinon* etc.

2.4.3 Vectorisation

Comme on l'a déjà vu, un gros avantage de R est sa capacité à travailler vectoriellement ce qui évite d'avoir à programmer des boucles `for` et/ou des tests. Ainsi, la boucle `for` du § 2.4.1 peut être remplacée simplement par

```
> x <- 1:10
> y <- x^2
```

tandis que la boucle suivante :

```
> n <- 10
> x <- rpois(n, 20)
> pair <- rep(FALSE, n)
> for (i in 1:n) {
+     if (x[i]%%2 == 0)
+         pair[i] <- TRUE
+ }
```

peut s'écrire simplement `pair[x%%2==0]<-TRUE`.

Plus généralement, toute fonction (de R ou créée par l'utilisateur) peut être appliquée à tous les éléments d'une matrice (ou seulement à certaines lignes ou colonnes), d'un vecteur ou d'une liste. Pour cela on utilise les fonctions `apply`, `sapply` ou `lapply` (on en verra un exemple au paragraphe suivant).

2.4.4 Écrire une fonction

L'essentiel du travail sous R se fait à l'aide de fonctions dont les arguments sont indiqués entre parenthèses. Outre les fonctions déjà implémentées dans R (on en a déjà vu un certain nombre), l'utilisateur peut écrire ses propres fonctions (c'est même d'ailleurs conseillé !) afin de pouvoir efficacement répéter certaines opérations. Voici par exemple l'écriture de la fonction `compte` qui prend deux arguments, `lettre` (un caractère) et `seq` (un vecteur de caractères), et retourne le nombre d'occurrences de `lettre` dans `seq` :

```
> compte <- function(lettre, seq) {
+     seqbinaire <- rep(0, length(seq))
+     seqbinaire[seq == lettre] <- 1
+     nb <- sum(seqbinaire)
+     nb
+ }
```

Si l'on veut que la fonction retourne le contenu d'un objet (ici la valeur de `nb`), ce dernier doit figurer seul sur la dernière ligne de la fonction.
Pour pouvoir exécuter cette fonction, il faut la charger en mémoire, soit en tapant chacune des lignes au clavier soit en l'écrivant dans un fichier et en "compilant" ce dernier avec la fonction `source`, comme pour un programme (paragraphe suivant). Dès que la fonction est chargée, on l'utilise naturellement :

```
> sequence <- c("a", "a", "t", "g", "a", "g", "c", "t", "a", "g",
+     "c", "t", "g")
> compte("a", sequence)
```

```
[1] 4
```

Dans cet appel à la fonction `compte`, le nom des arguments (`lettre` et `seq`) n'a pas été utilisé ce qui implique que les objets (`"a"` et `sequence`) doivent être passés dans l'ordre. Sinon on peut écrire `compte(seq=sequence,lettre="a")`.
Il est aussi possible d'attribuer des valeurs par défaut aux arguments d'une fonction ; par exemple `compte<-function(lettre="a",sequence){ ...}`.

Si l'on veut obtenir la composition de `sequence` en (a,c,g,t), on pourra faire :

```
> bases <- c("a", "c", "g", "t")
> composition <- rep(0, 4)
> for (i in 1:4) composition[i] <- compte(bases[i], sequence)
> composition
```

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

ou "appliquer" la fonction `compte` au vecteur `bases` :

```
> sapply(bases, FUN = compte, seq = sequence)
```

```
a c g t
4 2 4 3
```

Les variables définies dans les fonctions (`nb` et `seqbinaire` dans la fonction `compte` par exemple) sont considérées comme des variables locales. Si ces variables existaient déjà dans l'environnement "extérieur" à la fonction alors leur contenu n'est pas modifié. Si elles n'existaient pas, alors elles n'existeront pas plus après l'appel à la fonction. Par contre, si la fonction utilise une variable non définie dans la fonction, alors R va la chercher dans l'environnement extérieur ; si elle existe, R utilise sa valeur, sinon un message d'erreur est donné.

Le code des fonctions créées et chargées en mémoire s'obtient simplement en tapant le nom de la fonction suivi de "entrée".

2.4.5 Écrire un programme

On écrira typiquement un programme R dans un fichier texte ayant l'extension '`.r`', par exemple `myprog.r`.

Si le programme définit des fonctions et les utilise, il convient de les définir avant de les utiliser (par exemple en tête du fichier, éventuellement après l'assignation des variables globales).

Le caractère '`#`' sert à ajouter des commentaires : en effet, R passe à la ligne suivante.

Pour exécuter le programme contenu dans le fichier `myprog.r`, il suffit de taper

```
> source("myprog.r")
```

Chapitre 3

Graphiques

R offre de nombreuses possibilités pour effectuer des graphiques et les multiples options graphiques permettent une grande flexibilité. La commande `demo(graphics)` donne un aperçu de ces possibilités. Ici nous allons nous contenter de voir les représentations graphiques les plus classiques (graphe 2D, histogramme, boxplot) ainsi que les options graphiques les plus courantes. Pour chaque fonction, la liste complète des options s'obtient via l'aide-en-ligne.

Généralement, l'affichage d'un graphique se fait dans une fenêtre graphique que l'on ouvre avec la commande

```
> x11()
```

La fermeture de la fenêtre graphique se fait avec la commande :

```
> dev.off()
```

On verra plus loin que le graphique peut aussi être effectué dans un périphérique graphique de type fichier (page 42).

3.1 Graphe 2D

3.1.1 Nuage de points

Nous allons nous mettre ici dans le cas où nous avons un certain nombre de points à représenter en dimension 2, c'est-à-dire que chaque point a une abscisse et une ordonnée. En d'autres termes, nous disposons d'un vecteur d'abscisses (`x`) et un vecteur d'ordonnées associées (`y`). L'exemple sur lequel nous allons travailler va être le suivant : les points correspondent à des trinucléotides, leur abscisse (resp. ordonnée) est leur comptage dans la première (seconde) séquence du tableau de données `mydata` :

```
> comptage <- t(mydata[c(-1, -2)])
> x <- comptage[, 1]
> y <- comptage[, 2]
> length(x)
```

```
[1] 64
```

```
> length(y)
```

```
[1] 64
```

Le graphique de base (cf. Figure 3.1) s'obtient alors en faisant :

```
> plot(x, y)
```

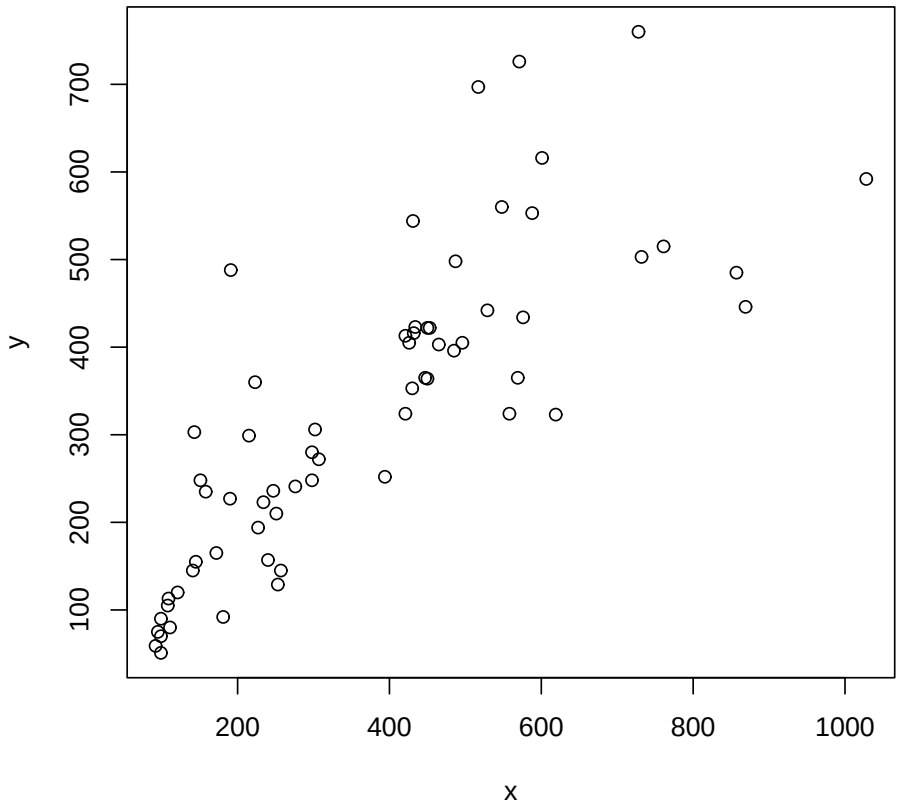

FIG. 3.1 – La fonction `plot(x,y)` sans options

.

Nous allons modifier ce graphique en jouant sur plusieurs options.

`xlim`, `ylim` fixent les limites inférieures et supérieures des axes (vecteurs à deux éléments),

`xlab`, `ylab` permettent de spécifier la légende des axes (mode caractère),

`main` permet de mettre un titre au dessus du graphique (mode caractère),

`pch` définit le symbole représentant les points : soit un entier de 1 à 25, soit n'importe quel caractère entre guillemets,

`col` spécifie la couleur des symboles (`"blue"`,`"red"` etc. la liste des couleurs est disponible avec `colors()`),

`bty` contrôle la forme du cadre : carré par défaut (`"o"`), en L (`"l"`), en U (`"u"`), en C (`"c"`), en 7 (`"7"`) ou en crochet (`"]"`).

Par exemple, la Figure 3.2 a été obtenue par :

```
> plot(x, y, xlab = "Sequence 1", ylab = "Sequence 2", main = "Comptage des trinucleotides",
+     xlim = c(0, 1200), ylim = c(0, 1200), pch = 18, col = "blue",
+     bty = "l")
```

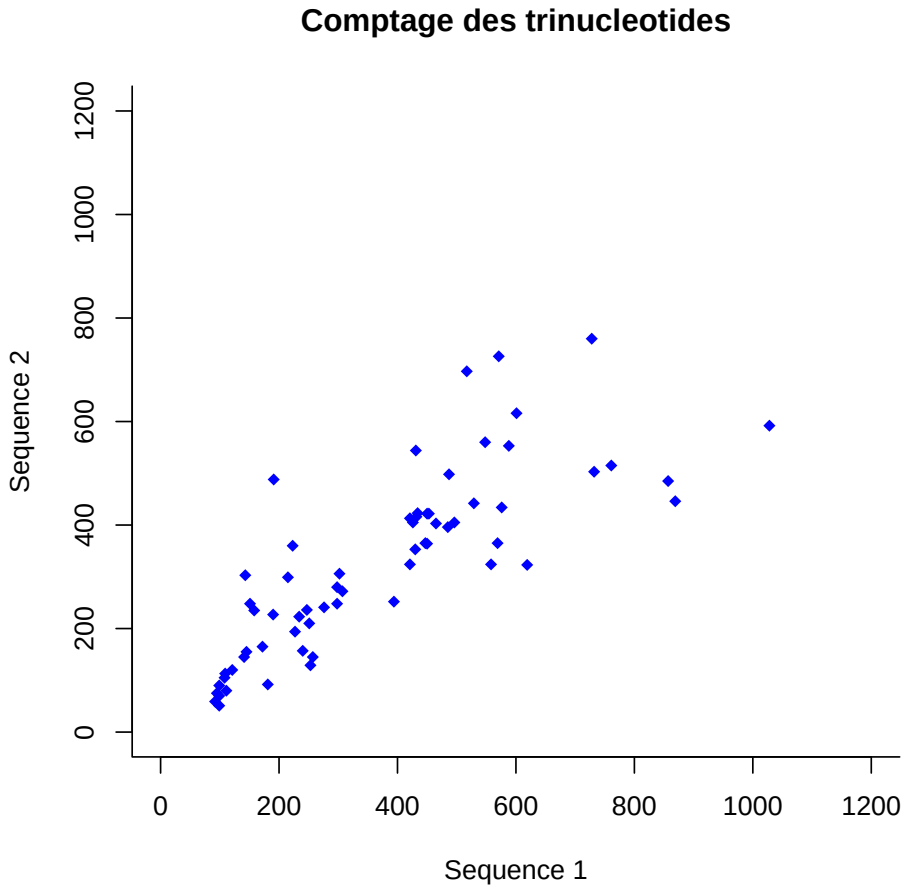

FIG. 3.2 – La fonction `plot` avec les options `xlab`, `ylab`, `main`, `xlim`, `ylim`, `pch`, `bty` et `col` .

On verra au §3.4 comment rajouter des éléments graphiques sur cette figure : axes, droites, texte, légende etc.

On peut jouer sur la taille des symboles grâce à l'option `cex`. Par défaut `cex=1` ; on peut lui fournir un nombre positif qui représente un coefficient multiplicatif par rapport à la taille par défaut (une valeur entre 0 et 1 pour réduire la taille, ou plus grande que 1 pour au contraire

l'augmenter). De la même façon les options `cex.axis`, `cex.lab` et `cex.main` contrôlent la taille des graduations des axes, des labels des axes et du titre.

Pour changer le style du texte, on utilise l'option `font` qui se décline aussi sous les formes `font.axis`, `font.lab` et `font.main`, (`1` pour normal, 2 pour italique, 3 pour gras et 4 pour gras italique).

3.1.2 Courbe

Pour dessiner une "courbe", on utilise aussi la fonction `plot` à la "seule" différence que l'on souhaite relier les points entre eux. Pour cela l'option principale est `type` qui vaudra soit `"b"` si l'on veut des points connectés par une ligne soit `"l"` si l'on veut seulement une ligne (par défaut `type="p"`).

Comme exemple, nous allons tracer la densité de la loi normale de moyenne 0 et de variance 1 entre -3 et 3 (cf. Figure 3.3) :

```
> t <- seq(-3, 3, by = 0.2)
> plot(t, dnorm(t, mean = 0, sd = 1), type = "l", col = "red",
+     main = "densite N(0,1)", font.main = 4)
```

Pour cet exemple, on aurait pu utiliser la fonction `curve` qui trace la courbe de fonctions de type $f(x)$ pour lequel R a accès aux valeurs pour tout x.

On peut jouer sur le type de ligne à l'aide de l'option `lty`. Par défaut `lty=1` ce qui donne des lignes continues, mais on peut obtenir des tirets (`lty=2`), des pointillés (`lty=3`), une alternance de points et tirets (`lty=4`), des tirets longs (`lty=5`) ou une alternance de tirets courts et longs (`lty=6`).

On peut aussi jouer sur l'épaisseur des lignes à l'aide de l'option `lwd` qui vaut 1 par défaut.

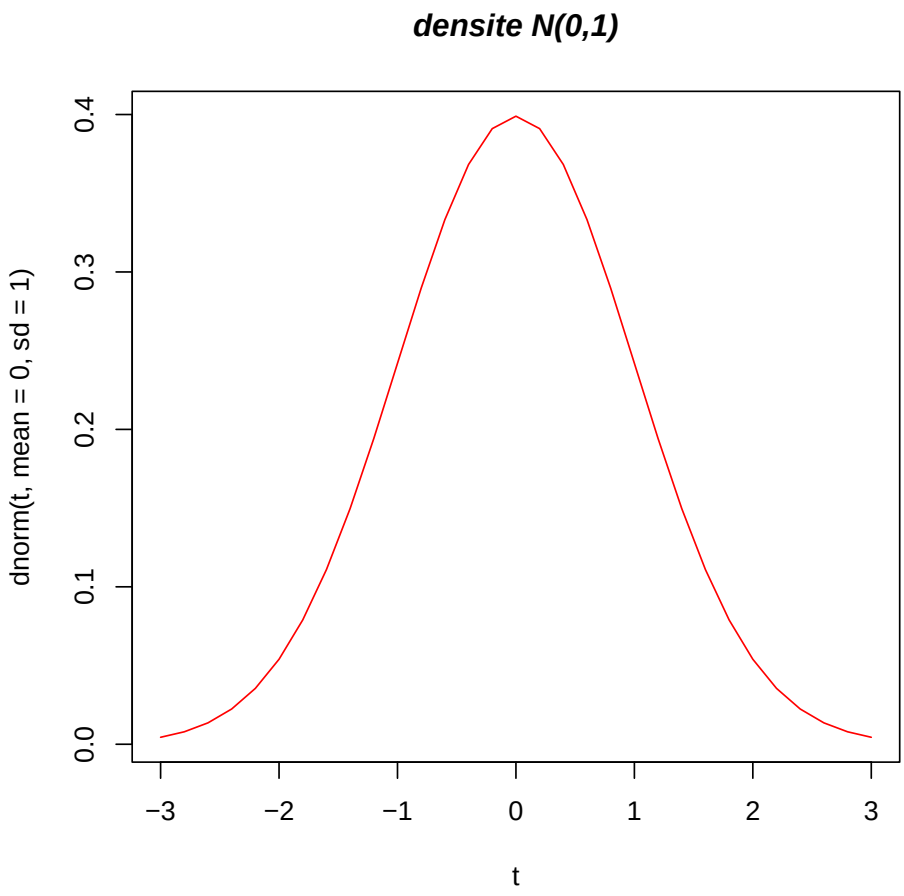

FIG. 3.3 – La fonction `plot` avec `type="l"`

3.2 Histogramme

Pour tracer un histogramme, la commande de base est la fonction `hist` (cf. Figure 3.4) :

```
> hist(x)
```

Voici certaines options spécifiques de la fonction `hist` :

- `breaks` : permet de spécifier les points de ruptures entre les barres de l'histogramme, soit sous la forme d'un vecteur, soit sous la forme d'un nombre de barres.
- `freq` : permet de choisir l'histogramme des effectifs, *frequency* en anglais (`freq=TRUE`, option par défaut), ou celui des proportions (`freq=FALSE`).
- `col` : indique la couleur pour remplir les barres.
- `plot` : si `plot=FALSE`, l'histogramme n'est pas tracé et la fonction renvoie la liste des points de ruptures et les effectifs.
- `right` : permet de choisir des intervalles de la forme $]a, b]$ si `right=TRUE` (par défaut ils sont de la forme $[a, b[$).

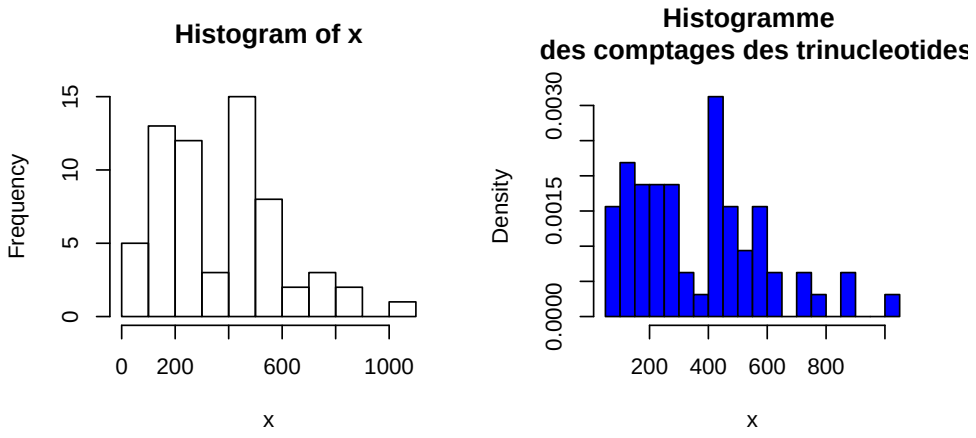

FIG. 3.4 – La fonction `hist` sans option (à gauche) et avec les options spécifiques `breaks`, `freq` et `col`(à droite)

L'histogramme de droite de la Figure 3.4 est ainsi obtenu :

```
> hist(x, breaks = 15, freq = FALSE, col = "blue", main = "Histogramme \n comptages des trinucl
```

3.3 Boxplot

La représentation en "*boîte à moustache*" d'un vecteur est à mi-chemin entre un histogramme et la fonction de répartition. Elle permet de visualiser la moyenne, la médiane, les quartiles à 25% et 75% de la distribution empirique. Elle s'obtient avec la fonction `boxplot`. Voici comment obtenir le boxplot de nos deux vecteurs `x` et `y` (cf. Figure 3.5) :

```
> boxplot(list(x, y))
```

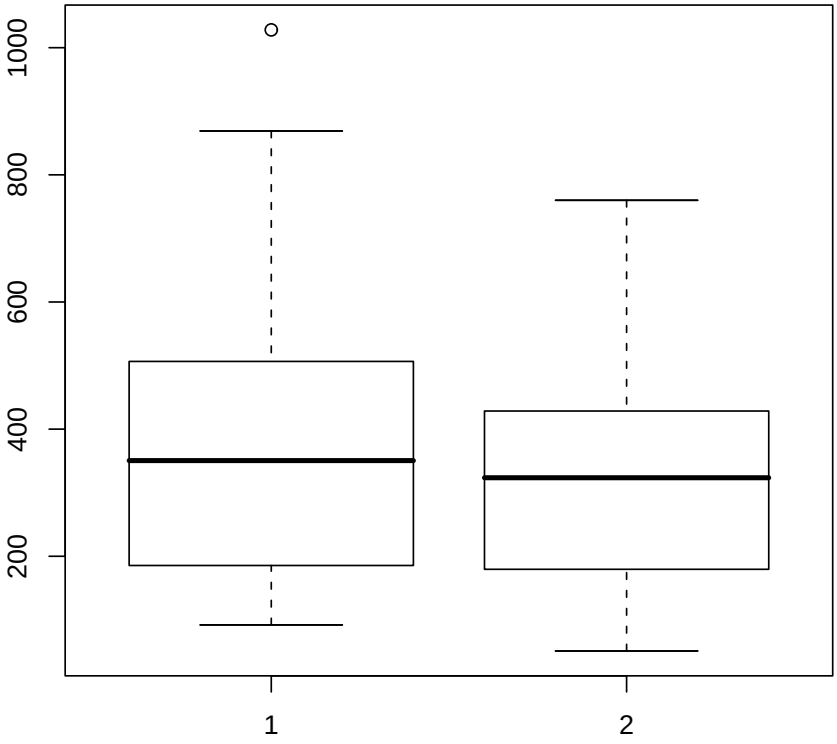

FIG. 3.5 – La fonction `boxplot` sans option

Par défaut, les moustaches ont une longueur maximale égale à 1.5 fois la taille de la boite. Ce coefficient peut être modifié avec l'option numérique `range`. On peut aussi modifier la largeur de boîte avec l'option `width` (c'est un vecteur aussi long qu'il y a de boxplots à afficher). L'option `names` de type caractère permet de spécifier les labels à afficher sous chaque boîte, par exemple ici on pourrait utiliser `names=c("x","y")`).

3.4 Fonctions graphiques secondaires

Voici d'autres fonctions graphiques dites *secondaires* car elles agissent sur un graphique existant.

- `abline(a,b)` : trace la droite $y = bx+a$. Pour tracer une droite horizontale (resp. verticale), on pourra utiliser `abline(h=)` (resp. `abline(v=)`).
- `points(x,y,...)` : ajoute des points,
- `lines(x,y,...)` : relie de nouveaux points,
- `text(x,y,labels,...)` : ajoute le texte défini par `labels` au point de coordonnées (`x`,`y`).
- `mtext(text,side=3,line=0,...)` : ajoute le texte défini par `text` dans la marge (du bas si `side=1`, de gauche si `side=2`, du haut si `side=3`, de droite si `side=4`) et à un nombre de lignes du cadre spécifié par `line`.
- `legend(x,y,legend,...)` : ajoute une légende au point de coordonnées (`x`,`y`).

Cette liste n'est pas exhaustive.

Voici un premier exemple (cf. Figure 3.6) qui reprend la Figure 3.2

```
> trinucleotides <- names(mydata)[c(-1, -2)]
> plot(x, y, xlab = "Sequence 1", ylab = "Sequence 2", main = "Comptage des trinucleotides",
+     xlim = c(0, 1200), ylim = c(0, 1200), type = "n")
> text(x, y, labels = trinucleotides, col = "green")
> abline(h = 0)
> abline(v = 0)
> abline(a = 0, b = 1)
```

Noter l'option `type="n"` qui affiche les points de façon invisible.

Voici un deuxième exemple (cf. Figure 3.7) qui reprend la Figure 3.3

```
> t <- seq(-3, 3, by = 0.2)
> plot(t, dnorm(t, mean = 0, sd = 1), ylab = "densite", type = "l",
+     col = "red")
> points(t, dnorm(t, mean = 0, sd = sqrt(2)), type = "l", lty = 2,
+     col = "blue")
> legend(1.5, 0.35, c("N(0,1)", "N(0,2)"), lty = c(1, 2), col = c("red",
+     "blue"))
> mtext(side = 3, line = 1, "Densites gaussiennes")
```

3.5 Paramètres graphiques

Certaines options des fonctions graphiques vues précédemment (`cex`, `col`, `font`, `lty`, `lwd`, `pch`, `bty` etc.) sont en fait des paramètres graphiques qui peuvent être modifiés indépendamment de ces fonctions. Néanmoins, il existe d'autres paramètres graphiques qui ne peuvent pas être utilisés comme options de fonctions graphiques. Le nombre de paramètres graphiques est élevé (leur liste s'obtient en tapant `?par`). Nous en citerons 3 nouveaux :

- `bg` : spécifie la couleur de l'arrière-plan,
- `mfcol` : un vecteur de la forme `c(nrow,ncol)` qui partitionne la fenêtre graphique en `nrow` lignes et `ncol` colonnes ; les graphiques s'affichent alors successivement par colonne,

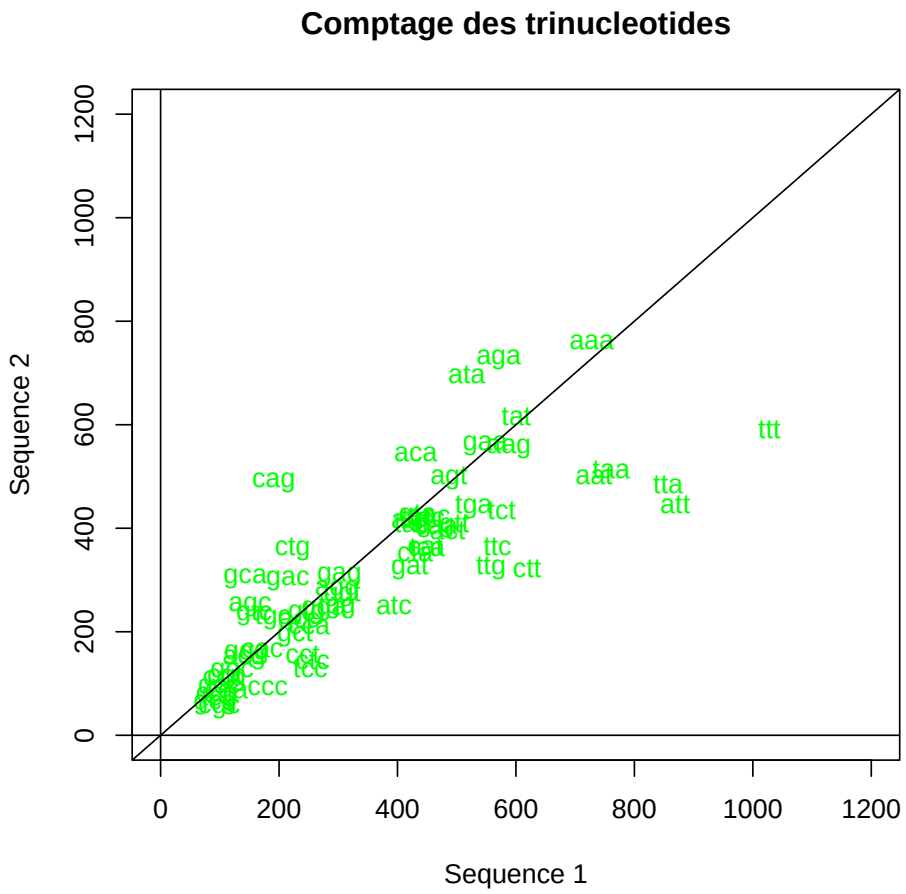

FIG. 3.6 – Utilisation des fonctions `text` et `abline` avec la fonction `plot`

– `mfrow` : analogue de `mfcol` avec un affichage par ligne.

Ces deux dernières options sont particulièrement utiles pour afficher plusieurs graphiques sur la même page.

De manière générale, les paramètres graphiques se modifient via la fonction `par`. Par exemple, la commande

```
> par(bg = "lightsalmon")
```

a pour effet d'attribuer un fond saumon aux prochains graphiques.

Avant de modifier les paramètres graphiques, il est important de sauvegarder les valeurs initiales de façon à pouvoir les rétablir par la suite.

3.6 Périphériques graphiques

Les graphiques peuvent s'exécuter dans d'autres périphériques que la fenêtre graphique, en particulier dans des fichiers. Ces périphériques sont ouverts avec une fonction qui dépend du

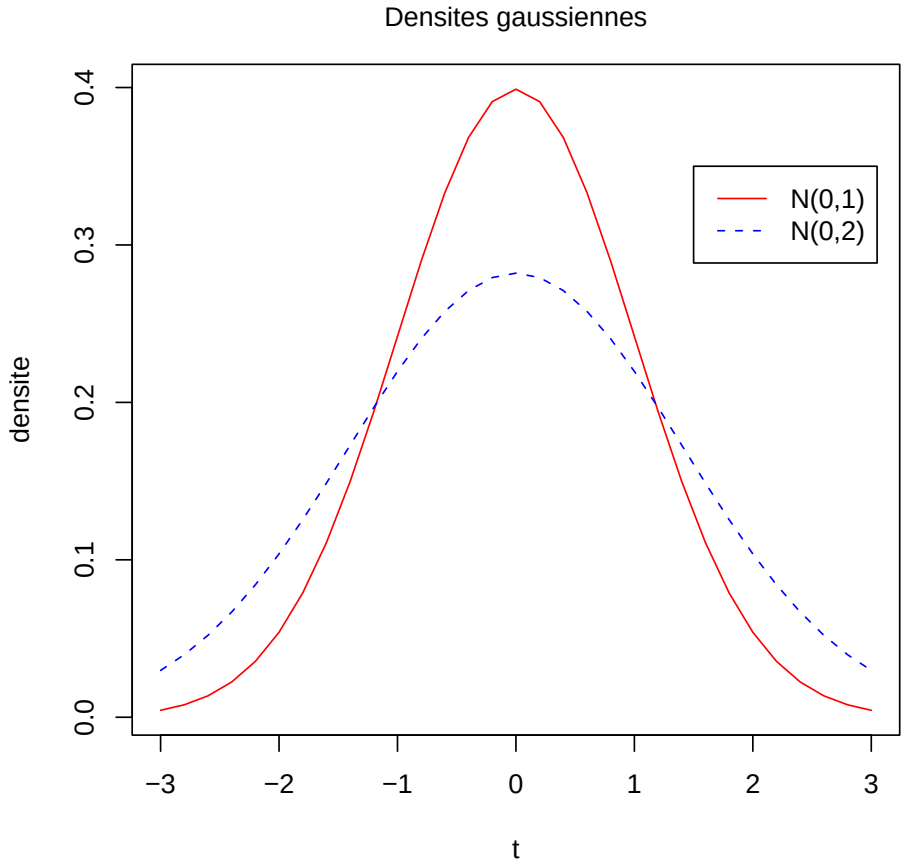

FIG. 3.7 – Utilisation des fonctions `points`, `legend` et `mtext` avec la fonction `plot`

format du fichier, par exemple :

```
postscript(file="graphe.ps")
pdf(file="graphe.pdf")
jpeg(file="graphe.jpg")
```

La liste des périphériques disponibles pour votre installation s'obtient en tapant `?device`.

Ces périphériques se ferment de la même façon que pour la fenêtre graphique à savoir avec `dev.off()`.

Le dernier périphérique ouvert devient celui sur lequel les graphiques s'affichent. Si plusieurs périphériques ont été ouverts (et non fermés), leur liste s'obtient par la commande `dev.list()` et on peut les fermer en spécifiant à `dev.off` leur numéro.

Chapitre 4

Quelques analyses statistiques

Nous illustrerons ce chapitre en utilisant deux jeux de données.

Le premier jeu de données provient du tableau `mydata` déjà utilisé au chapitre 2 ; il concerne des comptages de mots dans des séquences d'ADN de différents virus.

```
> mydata <- read.table("data/composition3mer.tab", header = T)
> dim(mydata)
```

```
[1] 86 66
```

```
> colnames(mydata)
```

```
 [1] "seq"  "long" "aaa"  "aac"  "aag"  "aat"  "aca"  "acc"  "acg"  "act"
[11] "aga"  "agc"  "agg"  "agt"  "ata"  "atc"  "atg"  "att"  "caa"  "cac"
[21] "cag"  "cat"  "cca"  "ccc"  "ccg"  "cct"  "cga"  "cgc"  "cgg"  "cgt"
[31] "cta"  "ctc"  "ctg"  "ctt"  "gaa"  "gac"  "gag"  "gat"  "gca"  "gcc"
[41] "gcg"  "gct"  "gga"  "ggc"  "ggg"  "ggt"  "gta"  "gtc"  "gtg"  "gtt"
[51] "taa"  "tac"  "tag"  "tat"  "tca"  "tcc"  "tcg"  "tct"  "tga"  "tgc"
[61] "tgg"  "tgt"  "tta"  "ttc"  "ttg"  "ttt"
```

Comme nous avons besoin de la fréquence des mots dans la séquence (et non du nombre de mots), nous allons construire, à partir de `mydata`, un nouveau jeu de données `Y` en divisant le nombre de mots par la longueur de la séquence correspondante.

```
> Y <- cbind(mydata[, c(1, 2)], mydata[, -c(1, 2)]/mydata[, 2])
> dim(Y)
```

```
[1] 86 66
```

Le deuxième jeu de données est extrait d'une expérimentation en nutrition animale ; il concerne l'augmentation de poids de porcelets en fonction du régime alimentaire de leur mère.

```
> porcelets <- read.table("data/poids.txt", header = T)
> dim(porcelets)

[1] 24  2

> colnames(porcelets)

[1] "aliment" "pds"
```

4.1 Statistiques descriptives

4.1.1 Analyse en Composantes Principales (ACP)

L'analyse en composantes principales ou ACP, donne une représentation synthétique, numérique ou graphique, des données. Cette méthode est particulièrement intéressante lorsqu'on dispose d'un nuage de points dans un espace de dimension élevée, c'est-à-dire qu'à chaque observation correspond un grand nombre de variables. L'ACP va conduire à un sous-espace de plus petite dimension, tel que la projection sur ce sous-espace retienne la majeure partie de l'information.

L'ACP s'utilise avec des données quantitatives. Les données sont structurées en lignes, associées aux individus (individu $\equiv$ unité observable), et en colonnes, associées aux variables qui décrivent les individus. Les lignes sont généralement indicées par i, $i \in \{1, ..., n\}$ et les colonnes par j, $j \in \{1, ..., p\}$. Si l'on appelle X ce tableau de données, l'élément (i, j) de X, noté $x_{i,j}$, est égal à la valeur prise par la variable j pour l'individu i.

Le principe de l'ACP est d'établir une nouvelle base orthonormée dans laquelle les axes (appelés composantes) sont construits séquentiellement de façon à absorber une quantité maximale de variance. Les nouveaux axes sont ainsi ordonnés, le premier axe portant la plus grande part de la variabilité et le dernier axe portant la plus faible part.

L'ACP nécessite une phase de préparation des données qui doivent être centrées. Les variables seront aussi réduites si l'on veut qu'elles aient toutes la même influence dans l'analyse.

La fonction `scale()` permet de centrer et/ou réduire les données en colonnes (à noter : par défaut `center` et `scale` valent `TRUE`) :

```
scale(x, center = TRUE, scale = TRUE)
```

Dans les fonctionnalités chargées par défaut avec R (package **stats**), 2 fonctions sont disponibles pour réaliser une ACP : `princomp()` et `prcomp()`.
`princomp()` calcule les valeurs propres sur la matrice de covariance ou de corrélation alors que `prcomp()` utilise la décomposition en valeurs singulières ; `prcomp` est une généralisation de la méthode `princomp`.

Pour illustrer l'ACP, nous allons selectionner un sous-ensemble du jeu de données Y :

```
> indd1 <- c(3, 4, 6, 7, 9, 12, 16, 21, 22, 30, 31, 37, 39, 44,
+     46, 47, 53, 54, 60, 66, 70, 81, 82)
> mots1 <- c("aaa", "agg", "cgt", "tgt", "ttt")
> Y1 <- Y[indd1, mots1]
> dim(Y1)
```

```
[1] 23  5
```

Sur ce sous-ensemble, noté Y1, nous allons appliquer une ACP en utilisant la fonction `prcomp` qui possède les options `center` et `scale` fixées respectivement à `TRUE` et `FALSE` par défaut. Ici, nous avons choisi de normaliser les données, aussi nous modifions l'option `scale` :

```
> Y1.prcomp <- prcomp(Y1, scale = T)
```

Nous utilisons ensuite la fonction `summary()` qui résume les résultats d'une analyse. Cette fonction reconnaît le type d'objet qu'on lui attribue et applique sur l'objet la fonction `summary` spécifique à cet objet.

```
> summary(Y1.prcomp)
```

```
Importance of components:
                          PC1   PC2    PC3    PC4     PC5
Standard deviation     1.623 1.390 0.5624 0.3044 0.15789
Proportion of Variance 0.527 0.386 0.0633 0.0185 0.00499
Cumulative Proportion  0.527 0.913 0.9765 0.9950 1.00000
```

La variabilité totale est représentée par la surface de l'histogramme (Figure 4.1). Le but de l'analyse étant de conserver le maximum de variabilité avec le minimum de composantes, on observe la décroissance de la variance sur les différentes composantes et on cherche une rupture de la pente. Ainsi, on remarque sur la Figure 4.1 une rupture de la pente entre la deuxième et la troisième composante. En se rapportant aux valeurs de proportions de variance cumulées, on voit que les deux premières composantes portent 91,3% de la variance. On peut donc choisir de réduire l'espace à ces 2 premières composantes.

La représentation sous forme de biplot (Figure 4.2) permet de visualiser la proximité entre les individus. Elle permet également d'apprécier les corrélations entre les variables. Par défaut, le nuage est projeté sur les deux premiers axes principaux, mais on peut spécifier explicitement les axes en utilisant l'option `choices=c(1,2)` (valeur par défaut).

4.1.2 Classification

L'objectif de la classification est de regrouper les objets similaires parmi un ensemble d'objets dont les caractéristiques sont connues.
La similarité des objets est appréciée en calculant leurs distances. Plusieurs distances sont disponibles dans R. Il existe aussi plusieurs façons de regrouper les objets : soit par un processus hiérarchique, ascendant ou descendant, soit sur la base du "partitionnement" à partir d'un nombre prédéfini de classes.

Dans R, il existe la bibliothèque `cluster` qui propose plusieurs méthodes de classification adaptées à différents types de données.

Par exemple, la fonction `agnes()` permet de réaliser une classification hiérarchique ascendante : à chaque étape du processus les deux objets ou classes les plus proches sont fusionnés pour former une nouvelle classe, et les distances de cette nouvelle classe à tous les autres, objets ou classes, sont recalculées.

```
> plot(Y1.prcomp)
```

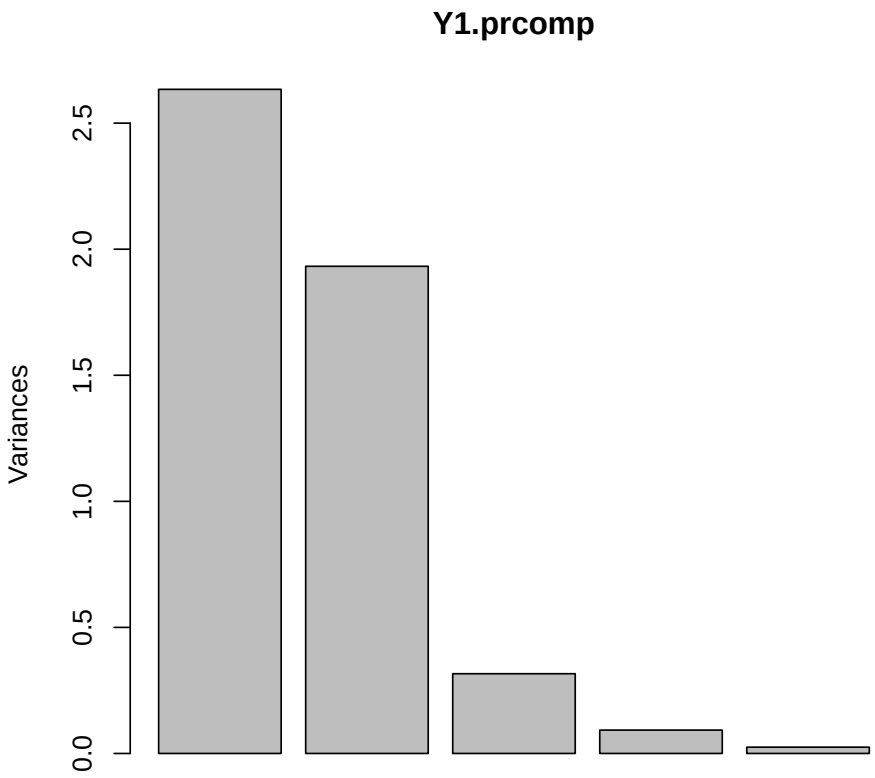

FIG. 4.1 – Variance portée par chacune des composantes

```
> biplot(Y1.prcomp)
```

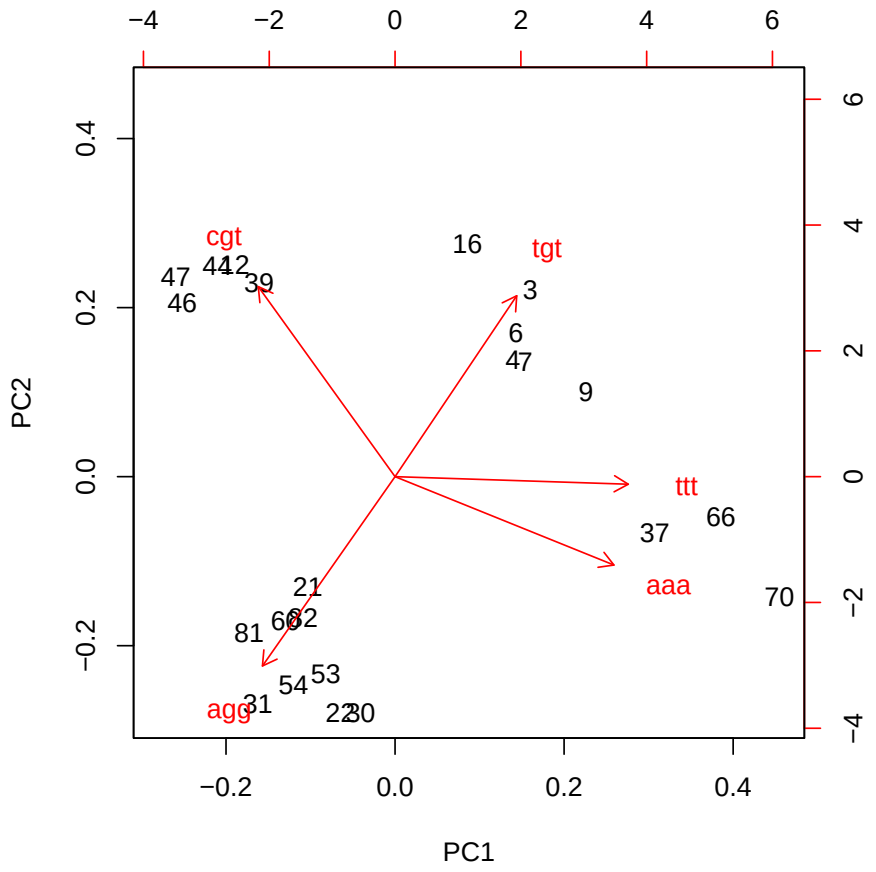

FIG. 4.2 – Biplot issu des 2 premières composantes de l'ACP

```
> library(cluster)
> agnes(Y1)

Call:         agnes(x = Y1)
Agglomerative coefficient:  0.8894554
Order of objects:
 [1] 3  16 4  7  6  9  37 12 44 39 46 47 21 60 82 81 22 31 53 54 30 66 70
Height (summary):
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
0.001691 0.003807 0.006542 0.012210 0.014180 0.056260

Available components:
[1] "order"     "height"    "ac"        "merge"     "diss"      "call"
[7] "method"    "order.lab" "data"
```

La fonction `agnes` retourne une liste composée de plusieurs champs, en particulier, le coefficient d'agglomération (`ac`) qui est une mesure de force de la "structure en classes". Cette mesure est calculée de la façon suivante : soit $d(i)$ la hauteur de la première fusion de chaque objet et soit h la hauteur de la dernière fusion, le coefficient d'agglomération s'écrit alors comme la moyenne des $(1 - d(i)/h)$. Une valeur proche de 1 du coefficient d'agglomération signifie que les objets sont fortement structurés en classes. Une valeur proche de 0 signifie que les objets appartiennent tous à une même classe.

Le champ "`merge`" liste les étapes de fusion. Les chiffres affectés du signe "-" correspondent aux objets de départ. Les chiffres sans signes sont les numéros des classes attribués au fur et à mesure qu'elles sont construites.

Le champ "`height`" est le vecteur des hauteurs de fusion.

Tous les champs peuvent être extraits à l'aide du `$`. Par exemple :

```
> agnes(Y1)$height

 [1] 0.007256906 0.011765932 0.001708030 0.003026567 0.022953801 0.014979353
 [7] 0.034979462 0.001872430 0.003747635 0.005237884 0.001691141 0.029214443
[13] 0.005472515 0.003984952 0.005827433 0.011098483 0.008916999 0.005466319
[19] 0.003279229 0.018711126 0.056256535 0.011160517
```

Deux types de graphiques sont proposés pour afficher les résultats de la fonction `agnes` :

- la bannière, qui s'obtient avec la fonction `bannerplot()` et qui reproduit les hauteurs des différentes fusions dans l'ordre de sortie des objets ;
- l'arbre de classification qui s'obtient avec la fonction `pltree()` et qui permet une vue synthétique des regroupements et le repérage d'éventuelles classes.

Nous avons représenté sous forme de bannière (Figure 4.3) puis d'arbre (Figure 4.4) les résultats de la classification des séquences de Y1.

La classification peut être réalisée à partir des corrélations. Par exemple, les Figures 4.5 et 4.6 montrent l'arbre de classification établi pour les séquences de Y1 et celui établi pour les variables de Y, tous deux réalisés à partir des corrélations.

```
> bannerplot(agnes(Y1), main = "", sub = "")
```

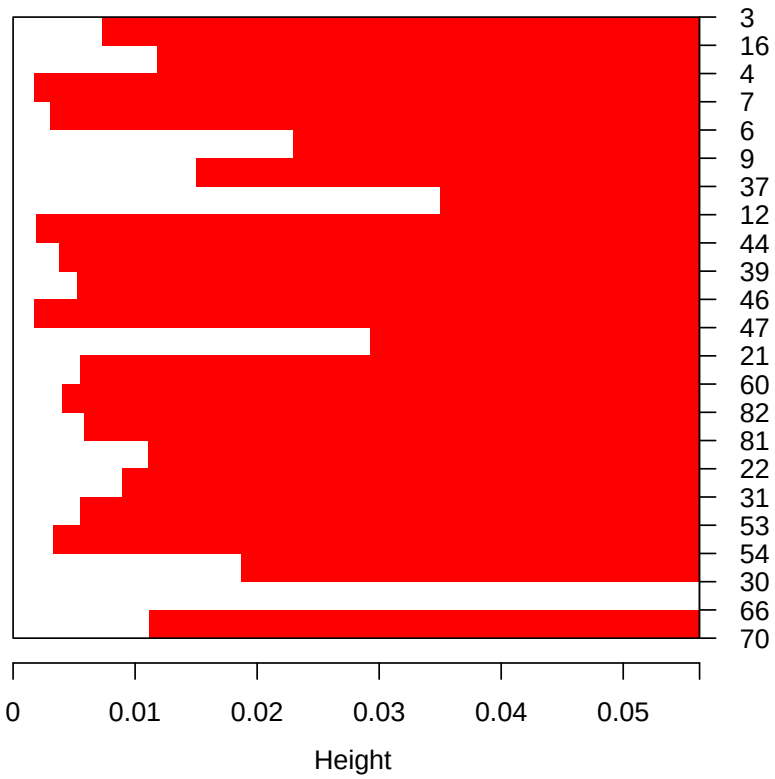

FIG. 4.3 – Bannière de Y1

```
> pltree(agnes(Y1))
```

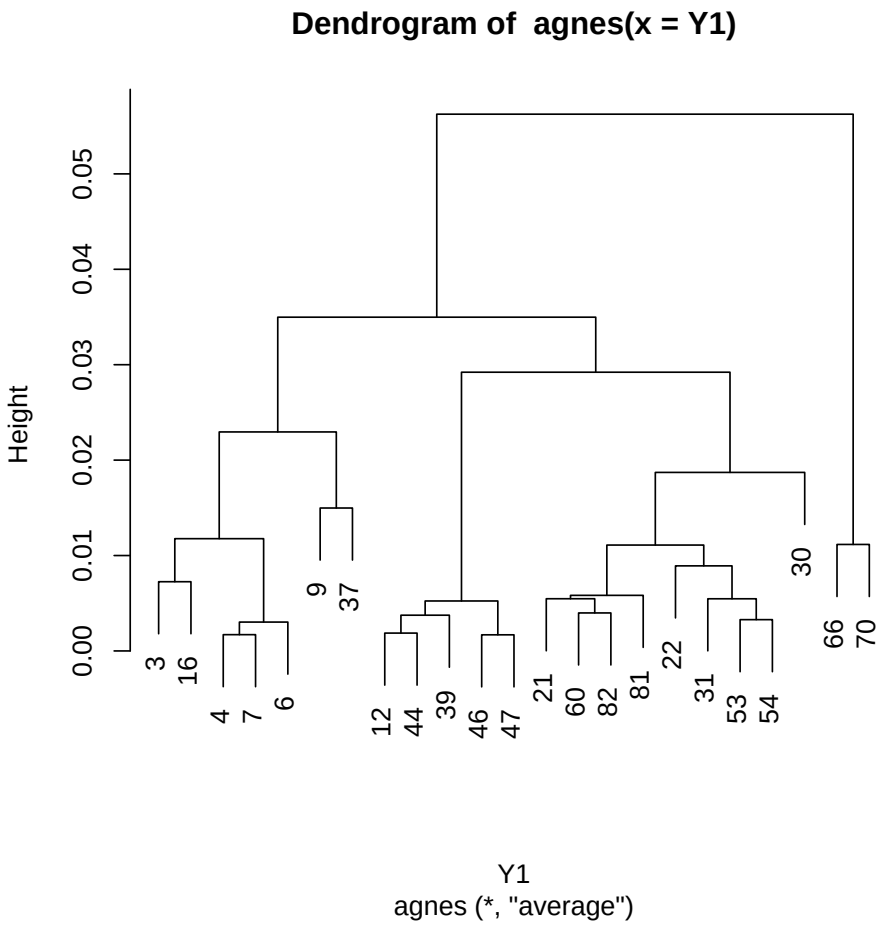

FIG. 4.4 – Arbre de classification des séquences de Y1

```
> pltree(agnes(cor(t(Y1))))
```

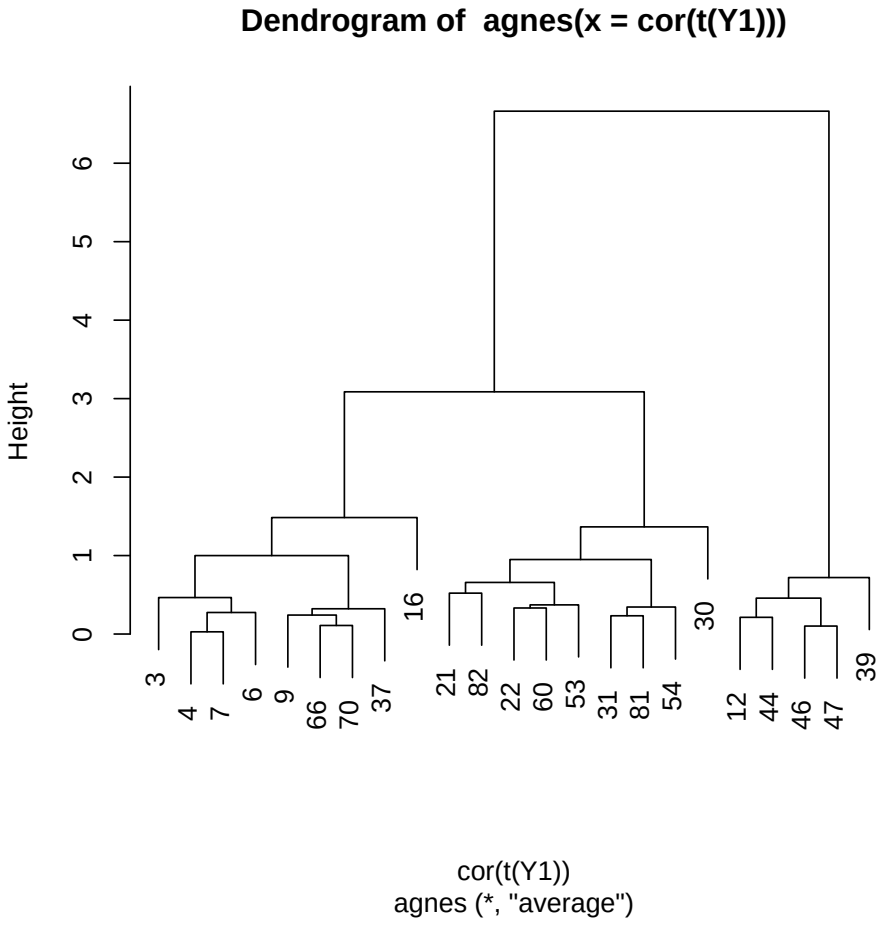

FIG. 4.5 – CHA des séquences de Y1 à partir des corrélations

```
> pltree(agnes(cor(Y[, -c(1, 2)])))
```

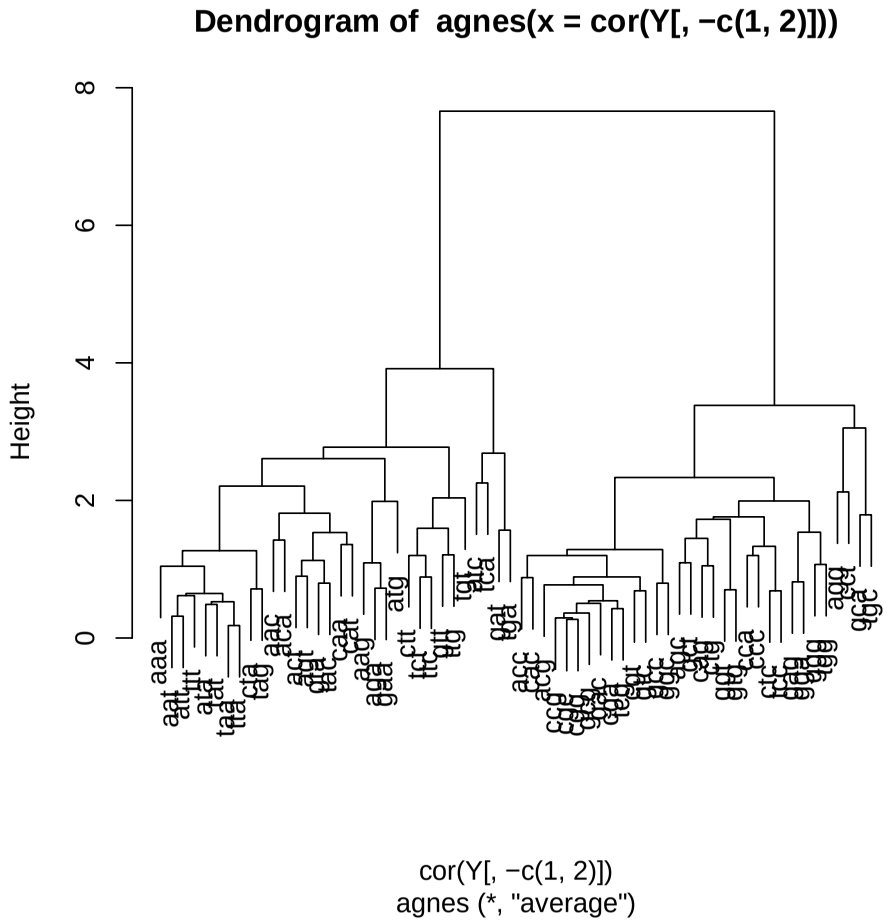

FIG. 4.6 – CHA des variables de Y à partir des corrélations

4.2 Statistiques inférentielles

4.2.1 La régression linéaire

La régression linéaire est une technique statistique utilisée pour modéliser des relations entre un ensemble de variables prédictives (les prédicteurs) et une ou plusieurs variables "réponse". Toutes les variables impliquées sont quantitatives.
Pour illustrer cette technique, nous considérerons la forme la plus simple avec une seule variable réponse, le mot "aaa", et un seul prédicteur, le mot "gtg". Si nous notons y la réponse et x_1, le prédicteur, le modèle statistique s'écrit de la façon suivante :

$$y = \mu + \alpha x_1 + \varepsilon$$

où μ représente l'intersection de la droite de régression avec l'axe des ordonnées, α la pente de la droite de régression et ε l'erreur résiduelle.

La fonction R permettant de réaliser une régression linéaire est la fonction `lm()`. Les arguments principaux attendus par cette fonction sont le modèle, qui en langage R s'écrit : *reponse* $\sim$ *predicteur*, et le nom du jeu de données.

```
> Y.lm <- lm(aaa ~ gtg, data = Y)
> summary(Y.lm)


Call:
lm(formula = aaa ~ gtg, data = Y)

Residuals:
      Min        1Q    Median        3Q       Max
-0.022422 -0.007019 -0.002900  0.006263  0.034720

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.062529   0.003566  17.536  < 2e-16 ***
gtg         -2.759367   0.280193  -9.848 1.17e-15 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.01074 on 84 degrees of freedom
Multiple R-Squared: 0.5359,        Adjusted R-squared: 0.5303
F-statistic: 96.98 on 1 and 84 DF,  p-value: 1.171e-15
```

Il est habituel de regarder la distribution des résidus afin de vérifier l'hypothèse de normalité sous-jacente au modèle. Cette distribution doit être centrée autour de zéro, et donc la médiane doit être proche de zéro. On peut le vérifier avec le champ `Residuals` du résumé de l'analyse.
Une autre façon de vérifier la normalité est de tracer un Q-Q plot des résidus. Le vecteur des résidus peut être extrait en utilisant soit `$resid`, soit la fonction `resid()`. On voit sur la Figure 4.7 que la linéarité n'est pas parfaite. L'hypothèse de normalité pourrait donc être, dans ce cas, remise en question. Le champ `Coefficients` fournit l'estimation des valeurs de la pente

```
> qqnorm(resid(Y.lm))
> qqline(resid(Y.lm), col = "blue")
```

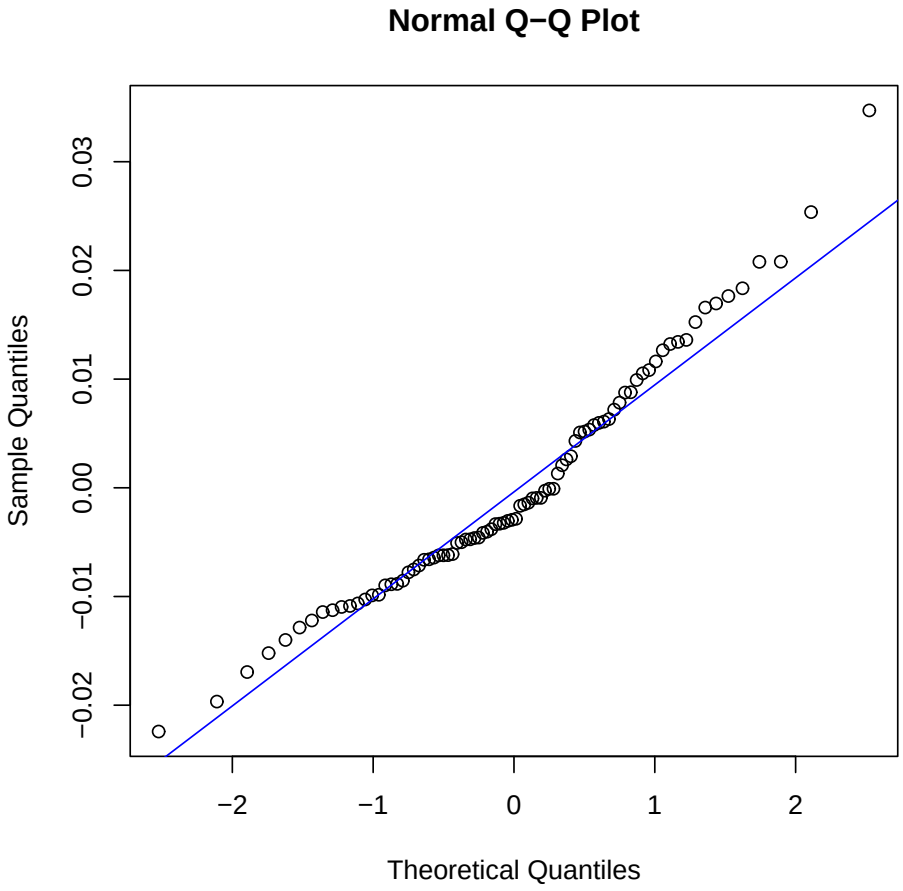

FIG. 4.7 – Q-Qplot des résidus de la régression

α et de l'intercept μ ainsi que les statistiques qui leur sont associées. Le vecteur des coefficients peut être extrait en utilisant `$coeff`. A partir de ces valeurs, on peut tracer la droite estimée sur le graphique représentant aaa en fonction de gtg (Figure 4.8).

4.2.2 L'analyse de variance

L'analyse de variance est une méthode statistique permettant de comparer des groupes. Elle peut être utilisée, par exemple, pour rechercher l'action d'un facteur sur une variable d'intérêt. Nous illustrerons cette méthode avec le jeu de données "porcelets" relatif à l'augmentation du poids de i porcelets, $i \in \{1, \ldots, n\}$, en fonction du type d'alimentation de leur mère.

Dans cette expérience, trois types d'aliments sont utilisés. Ils correspondent aux niveaux du facteur aliment :

```
> levels(porcelets$aliment)
```

```
[1] "al1" "al2" "T"
```

On peut représenter les données sous forme de boxplots pour chacun des groupes j, $j \in \{al1, al2, T\}$ (Figure 4.9). Dans la suite, nous noterons x_{ij} l'observation du porcelet i dont la mère reçoit l'aliment j.

Pour savoir s'il existe une influence de l'alimentation de la mère sur la croissance des porcelets, il faut comparer la part de variabilité due à l'alimentation de la mère et la part de variabilité résiduelle (variabilité entre les porcelets d'un même groupe). Pour cela, on calcule la somme des carrés des écarts à la moyenne intra groupes $(x_{ij} - x_{.j})^2$ que l'on notera "SSW", et la somme des carrés des écarts à la moyenne inter groupes $(x_{.j} - x_{..})^2$ que l'on notera "SSB". La somme des carrés totale $(x_{ij} - x_{..})^2$, notée "SST", peut ainsi se décomposer :

$$SST = SSW + SSB$$

La fonction `aov()` calcule les sommes de carrés intra et inter groupes.

```
> pds.aov <- aov(pds ~ aliment, data = porcelets)
> pds.aov
```

```
Call:
   aov(formula = pds ~ aliment, data = porcelets)

Terms:
                 aliment Residuals
Sum of Squares  35.41750  54.10875
Deg. of Freedom        2        21

Residual standard error: 1.605181
Estimated effects may be unbalanced
```

```
> intercept <- Y.lm$coeff[1]
> pente <- Y.lm$coeff[2]
> plot(Y$gtg, Y$aaa)
> abline(a = intercept, b = pente, col = "blue")
```

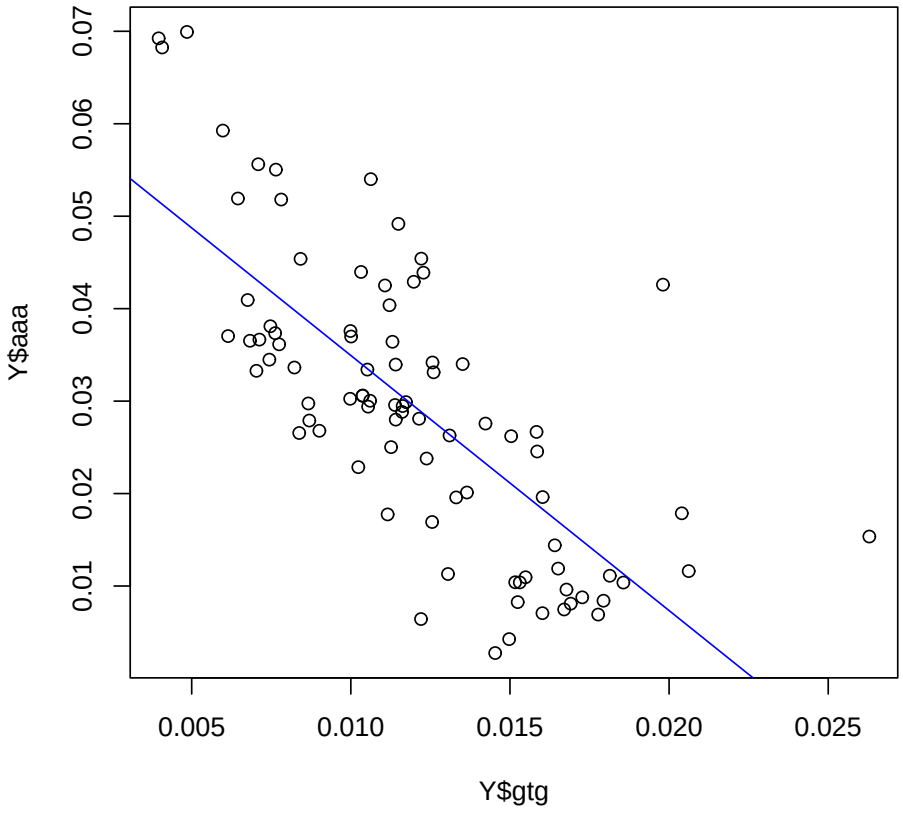

FIG. 4.8 – Graphe de `aaa` en fonction de `gtg` et droite estimée

```
> plot(porcelets)
```

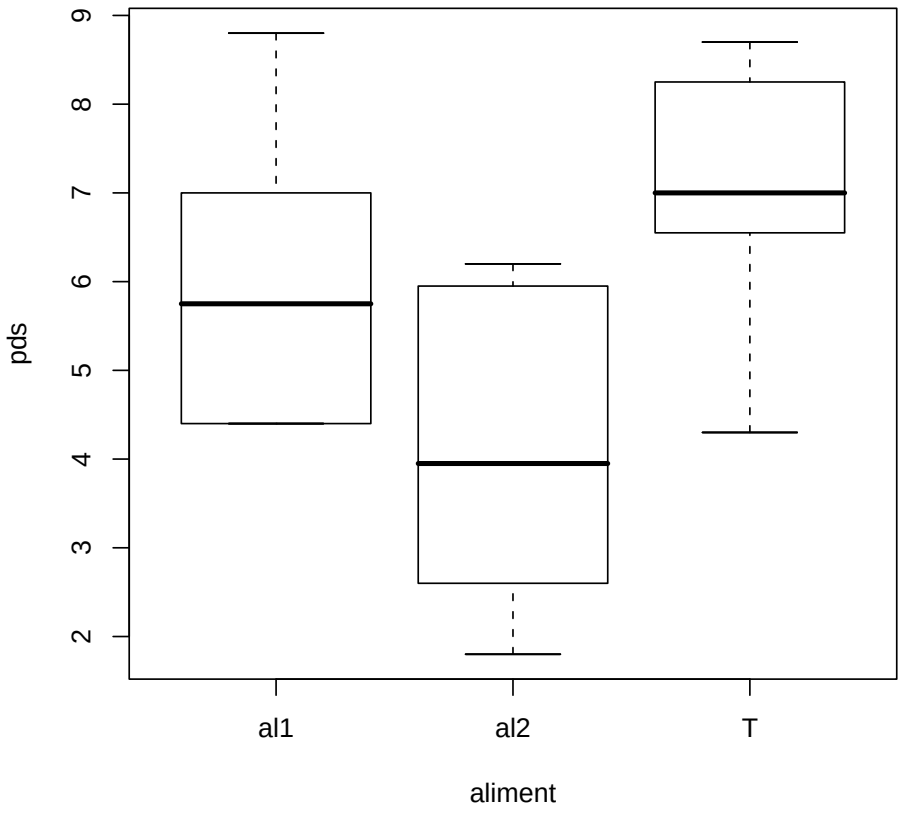

FIG. 4.9 – Augmentation de poids de porcelets en fonction du type d'alimentation de leur mère.

Chaque somme de carrés est associée à un degré de liberté (df) qui est pour l'aliment, son nombre de niveaux - 1, et pour la résiduelle, le nombre total d'observations - le df de l'aliment -1.

La fonction `anova()` réalise les tests statistiques à partir des résultats de l'`aov` et produit un tableau d'analyse de variance.

```
> anova(pds.aov)

Analysis of Variance Table

Response: pds
          Df Sum Sq Mean Sq F value   Pr(>F)
aliment    2 35.418  17.709  6.8729 0.005056 **
Residuals 21 54.109   2.577
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Ce tableau d'analyse de variance montre que la probabilité associée à l'effet de l'alimentation de la mère sur la croissance des porcelets est significative (si on considère un seuil de significativité à 0.05). Cela veut dire qu'au moins une des différences entre les 3 groupes est significative.

On peut en conclure que, globalement, l'alimentation de la mère influe sur la croissance des porcelets.

On peut aussi vouloir comparer les groupes 2 à 2. Pour cela, on utilise la fonction `pairwise.t.test()` qui réalise des comparaisons par paires basées sur des tests de Student. Une option de cette fonction permet de spécifier la méthode d'ajustement, en cas de comparaisons multiples. Par défaut, cette option est "holm".

```
> pairwise.t.test(porcelets$pds, porcelets$aliment, p.adjust.method = "bonferroni")

        Pairwise comparisons using t tests with pooled SD

data:  porcelets$pds and porcelets$aliment

    al1    al2
al2 0.1041 -
T   0.5132 0.0042

P value adjustment method: bonferroni

> pairwise.t.test(porcelets$pds, porcelets$aliment, p.adjust.method = "fdr")

        Pairwise comparisons using t tests with pooled SD

data:  porcelets$pds and porcelets$aliment
```

```
    al1    al2
al2 0.0520 -
T   0.1711 0.0042

P value adjustment method: fdr
```

Les résultats affichés montrent dans les 2 cas une différence significative de la croissance des porcelets entre les groupes T et al2. La différence entre les groupes al1 et al2 n'apparaît pas significative si la méthode d'ajustement choisie est celle de Bonferonni.

Index