UNIVERSITÉ PARIS OUEST NANTERRE LA DÉFENSE
U.F.R. SEGMI — Année universitaire 2013 – 2014
Master d'économie — Cours de M. Desgraupes

MATHS/STATS

Document 3 : Présentation de R

---

1 Installation et démarrage **2**

2 Manipulation des données **3**
- 2.1 Opérations élémentaires . . . 3
- 2.2 Les vecteurs . . . 3
- 2.3 Les fonctions . . . 4
- 2.4 Les listes . . . 7
- 2.5 Les tableaux . . . 8
- 2.6 Les matrices . . . 8
- 2.7 Vecteurs logiques . . . 12
- 2.8 Valeurs particulières . . . 13

3 Graphiques avec R **13**
- 3.1 Histogrammes . . . 13
- 3.2 Boîtes à moustaches . . . 16
- 3.3 Courbes . . . 16
- 3.4 Densités . . . 18
- 3.5 Polygône de fréquences . . . 20
- 3.6 Lignes et quadrillages . . . 20
- 3.7 Paramètres graphiques . . . 22

4 Échantillons avec R **22**
- 4.1 Les data frames . . . 22
- 4.2 Échantillons d'une loi de probabilité . . . 25
- 4.3 Échantillons par tirage . . . 28

5 Tests statistiques avec R **28**
- 5.1 Tests disponibles . . . 28
- 5.2 Exemples de tests . . . 29
  - 5.2.1 Test d'ajustement du χ^2 . . . 29
  - 5.2.2 Test de proportions . . . 29
  - 5.2.3 Test de Student . . . 30

6 Extensions de R **31**
- 6.1 Créer de nouvelles fonctions . . . 31
- 6.2 Créer des scripts . . . 32
- 6.3 Les *packages* . . . 32

7 Configuration de R **33**
7.1 Les options de R 33
7.2 Les chemins d'accès 34

8 Aide et tutoriels **35**
8.1 Obtenir de l'aide avec R 35
8.2 Tutoriels sur R 36

---

1 Installation et démarrage

Le site officiel de R est :

`http://www.r-project.org`

Pour se procurer R, il suffit de télécharger une version compilée. Il en existe pour toutes les plateformes usuelles (Windows, Mac OS X, Linux). On les trouvera aux adresses suivantes respectivement :

Windows http ://mirror.ibcp.fr/pub/CRAN/bin/windows/

Mac OS X http ://mirror.ibcp.fr/pub/CRAN/bin/macosx/

Unix http ://mirror.ibcp.fr/pub/CRAN/bin/linux/

On démarre R comme n'importe quelle application. Le programme affiche une fenêtre dite *espace de travail* ou *console* comportant un préambule comme ceci :

```
R version 3.0.0 (2013-04-03) -- "Masked Marvel"
Copyright (C) 2013 The R Foundation for Statistical Computing

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.
```

Le préambule est suivi d'une *invite* marquée par le symbole > en début de ligne. C'est la *ligne de commande*. On saisit des instructions sur cette ligne et on les fait exécuter par R en pressant la touche RETOUR (retour-chariot) ou la touche ENTER du clavier.

2 Manipulation des données

2.1 Opérations élémentaires

Les opérations arithmétiques de base peuvent être exécutées directement sur la ligne de commande : somme, multiplication (avec le symbole $*$), division, puissance (avec le symbole ˆ), modulo (`%/%` pour le quotient et `%%` pour le reste). Par exemple :

```
> 58+7

[1] 65

> 58*7

[1] 406

> 58/7

[1] 8.285714

> 58^7

[1] 2207984167552

> 58 %/% 7

[1] 8

> 58 %% 7

[1] 2
```

Les nombres décimaux sont délimités par un point et non pas par une virgule.

L'assignation de valeurs à des noms de variables se fait avec le signe = ou bien le double symbole <− (un signe < suivi d'un tiret). On peut donc utiliser l'une ou l'autre des syntaxes suivantes :

```
> x = 58
> x <- 58
```

La seconde est plus fréquemment utilisée car elle indique mieux une affectation à gauche. R peut aussi faire, avec le double symbole −>, des affectations à droite qui sont des instructions de la forme :

```
> 58 -> x
```

2.2 Les vecteurs

L'objet de base de R est le vecteur. Plusieurs fonctions permettent de générer des vecteurs. Cette section indique les plus couramment utilisées.

La fonction **c** permet de créer des vecteurs de données :

```
> notes <- c(10.5, 12.5, 14.5, 10.0, 10.5, 8.0, 9.5, 11.5,
             15.0, 10.0)
> notes
```

```
[1]  10.5 12.5 14.5 10.0 10.5  8.0  9.5 11.5 15.0 10.0
```

Le symbole deux-points sert à construire des séquences de nombres entiers. La notation `m:n` (avec $m, n \in \mathbb{N}$) produit la séquence des entiers de m à n :

```
> 1:10

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

> 15:10

[1] 15 14 13 12 11 10
```

La fonction **seq** permet de créer des séries plus complexes. Elle prend trois arguments qui sont la valeur minimale, une valeur maximale et un pas. Par exemple :

```
> seq(1,5,0.3)

[1] 1.0 1.3 1.6 1.9 2.2 2.5 2.8 3.1 3.4 3.7 4.0 4.3 4.6 4.9
```

Cette instruction a produit une séquence obtenue en partant de 1 et en augmentant chaque fois de 0.3 sans dépasser finalement la valeur 5 (la plus grande valeur ainsi obtenue est 4.9).

Pour saisir les valeurs d'un vecteur interactivement sur la ligne de commande, on peut aussi se servir de la fonction **scan**, comme ceci :

```
> notes <- scan()

    1: 16
    2: 10
    3: 7
    4: 15
    5: 11
    6:
    Read 5 items
```

Après chaque valeur, on appuie sur la touche RETOUR. Pour terminer la saisie, on appuie une fois de plus sur RETOUR. Toutes les valeurs saisies sont retournées (ici dans la variable appelée *notes*) sous forme d'un vecteur. On aurait dans cet exemple :

```
> notes

[1] 16 10  7 15 11
```

2.3 Les fonctions

On peut appliquer une fonction directement à un vecteur : selon la fonction, on obtient une valeur numérique ou bien un autre vecteur. La table 1 contient une liste des fonctions statistiques les plus courantes et la table 2 une liste des principales fonctions mathématiques disponibles dans R. Voici quelques exemples :

```
> round(notes)

[1] 10 12 14 10 10  8 10 12 15 10
```

```
> ceiling(notes)

[1] 11 13 15 10 11  8 10 12 15 10

 > floor(notes)

 [1] 10 12 14 10 10  8  9 11 15 10

> max(notes)

[1] 15

> min(notes)

[1] 8

> mean(notes)

[1] 11.2

> median(notes)

[1] 10.5

> var(notes)

[1] 4.9

> sd(notes)

[1] 2.213594

> cumsum(notes)

[1]  10.5  23.0  37.5  47.5  58.0  66.0  75.5  87.0 102.0 112.0
```

On peut obtenir des résumés au moyen des commandes **summary**, **fivenum** et **quantile**. Par exemple :

```
> summary(notes)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   8.00   10.00   10.50   11.20   12.25   15.00

> fivenum(notes)

[1]  8.0 10.0 10.5 12.5 15.0

> quantile(notes)

    0%   25%   50%   75%  100%
  8.00 10.00 10.50 12.25 15.00
```

mean	moyenne
var	variance
sd	écart-type
cor	coefficient de corrélation $cor(x, y)$
sum	somme
range	intervalle [min, max]
min	minimum
max	maximum
median	médiane
cumsum	somme cumulée
cumprod	produit cumulé
cummax	maxima cumulés
cummin	minima cumulés
IQR [1]	intervalle inter-quartile
mad	déviation médiane absolue

TABLE 1 – Fonctions statistiques élémentaires

La fonction **summary** est un exemple de ce qu'on appelle une *fonction générique* : les informations qu'elle produit dépendent de l'objet auquel elle s'applique. Ici il s'agissait d'un vecteur numérique. On en verra d'autres exemples par la suite.

On obtient la longueur d'un vecteur avec la fonction **length** :

```
> length(notes)
```

```
[1] 10
```

On peut trier les données d'un vecteur numérique au moyen de la fonction **sort** :

```
> sort(notes)
```

```
[1]  8.0  9.5 10.0 10.0 10.5 10.5 11.5 12.5 14.5 15.0
```

Noter que cela ne change pas le vecteur *notes* lui-même : la liste que renvoie la fonction **sort** est une autre liste que la liste originale. On peut ainsi affecter le résultat à une autre variable :

```
> notes_tri <- sort(notes)
```

Pour que la liste triée remplace la liste *notes* originale, on doit écrire :

```
> notes <- sort(notes)
```

On extrait des éléments d'un vecteur au moyen d'une paire de crochets `[ ]`. Les indices commencent à 1. Par exemple :

```
> notes[2]
```

```
[1] 12.5
```

On peut passer, dans la paire de crochets, une séquence ou un vecteur d'indices afin d'obtenir plusieurs valeurs à la fois. La commande suivante renvoie les notes d'indice 5, 6 et 7 :

1. *IQR* est l'abréviation de *interquartile range* et *mad* l'abréviation de *median absolute deviation*.

abs	atan	cosh	log	sinh
acos	atanh	exp	log10	sqrt
acosh	ceiling	factorial	round	tan
asin	choose	floor	signif	tanh
asinh	cos	gamma	sin	trunc

TABLE 2 – Fonctions mathématiques

```
> notes[5:7]
```

```
[1] 10.5  8.0  9.5
```

La commande suivante renvoie les notes d'indice 1, 4, 7 et 10 :

```
> notes[seq(1,10,3)]
```

```
[1] 10.5 10.0  9.5 10.0
```

Des indices négatifs permettent de supprimer des éléments :

```
> notes[-4]
```

```
[1] 10.5 12.5 14.5 10.5  8.0  9.5 11.5 15.0 10.0
```

À la suite de cette commande, l'élément d'indice 4 a disparu du vecteur de notes.

2.4 Les listes

Les listes sont des objets différents des vecteurs. Les éléments d'un vecteur doivent tous être du même type tandis que les listes peuvent être constituées d'éléments hétérogènes. Elles sont créées au moyen de la fonction **list**. Par exemple, la liste suivante, appelée *lst*, comporte deux chaînes de caractères, un nombre et un vecteur :

```
> lst <- list("pommes", "poires", 12345, c(1:10))
> lst
```

```
[[1]]
[1] "pommes"
[[2]]
[1] "poires"
[[3]]
[1] 12345
[[4]]
[1]  1  2  3  4  5  6  7  8  9 10
```

On accède aux éléments d'une liste au moyen d'une *double* paire de crochets `[[ ]]` (et non pas d'une simple paire de crochets comme avec les vecteurs) :

```
> lst[[1]]
```

```
[1] "pommes"
```

Inversement, on peut modifier n'importe quel élément de la liste avec le symbole d'affectation `<-`. Par exemple :

```
> lst[[2]] <- 3.1416
```

Attention : si on utilise une *simple* paire de crochets, le résultat est non pas la valeur de l'élément correspondant dans la liste, mais une sous-liste constituée de cet élément (bien comparer la manière dont sont affichés respectivement `lst[[1]]` et `lst[1]`) :

```
> lst[1]

[[1]]
[1] "pommes"
```

2.5 Les tableaux

Un *tableau* (en anglais *array*) est une collection de valeurs ayant des indices multiples. Une matrice, par exemple, est un tableau possédant deux indices (indice des lignes et indice des colonnes). R est capable de gérer des tableaux ayant plus de deux indices. Les indices sont toujours numérotés à partir de 1.

Les dimensions sont le nombre d'éléments autorisés pour chaque indice (par exemple, le nombre de lignes et de colonnes d'une matrice). On les spécifie au moyen de la fonction **dim**. Les dimensions sont exprimées comme un vecteur : une matrice de taille 3×5 aura ainsi pour dimension le vecteur $c(3, 5)$ dans R.

Voici un exemple qui illustre comment manipuler les dimensions. La commande suivante, qui sera expliquée plus loin, crée un vecteur z de 120 valeurs numériques uniformément réparties entre 0 et 1 :

```
> z <- runif(120)
```

La commande que voici

```
> dim(z) <- c(10,4,3)
```

transforme le *vecteur* z en un *tableau* de taille $10 \times 4 \times 3$, ce que l'on peut voir comme 3 tables ayant chacune 10 lignes et 4 colonnes.

Les fonctions **matrix** et **array**, présentées dans la prochaine section, permettent de créer directement des matrices ou des tableaux.

Pour désigner un élément particulier d'un tableau, on indique ses indices dans une paire de crochets en les séparant par une virgule. Par exemple, on accèderait à l'élément de coordonnées (7,2,1) comme ceci :

```
> z[7,2,1]
```

Au lieu d'une valeur unique pour un indice, on peut spécifier une séquence de valeurs, comme par exemple $a[2:3, 3:5]$ qui correspond, dans une matrice a, aux éléments situés sur les lignes 2 et 3 et les colonnes 3 à 5.

Par convention, si un indice est laissé vide, c'est la totalité des valeurs de cet indice qui est sous-entendue. Par exemple $a[, 2]$ désigne toutes les valeurs de la colonne 2, $a[3:4,]$ désigne toutes les valeurs des rangs 3 et 4.

2.6 Les matrices

Définissons un vecteur de 12 éléments :

```
> z <- c(1,3,5,7,8,5,6,4,2,9,0,5)
> z
```

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

Convertissons-le en une matrice de 2 lignes et 6 colonnes

```
> dim(z)<-c(2,6)
> z

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

> dim(z)

[1] 2 6
```

On notera que les éléments du vecteur initial sont rangés en colonnes dans la matrice : on retrouve le vecteur z en parcourant successivement les 6 colonnes.

Le même résultat peut être obtenu au moyen de la fonction **matrix**. Celle-ci prend comme arguments un vecteur de valeurs, un nombre de lignes et un nombre de colonnes :

```
> z <- c(1,3,5,7,8,5,6,4,2,9,0,5)
> matrix(z,3,4)

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

Voici un autre exemple utilisant cette fois la fonction **array** dont la syntaxe est légèrement différente : le deuxième argument est un *vecteur* de dimensions. Créons un tableau matriciel de taille 4×5 avec la séquence des nombres entiers de 1 à 20 :

```
> M <- array(1:20, dim=c(4,5))
> M

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

On peut, par exemple, annuler les éléments 2 et 3 de la quatrième colonne comme ceci :

```
> M[2:3,4]<-0
> M

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10    0   18
[3,]    3    7   11    0   19
[4,]    4    8   12   16   20
```

On peut créer une matrice 4×5 de nombres aléatoires distribués uniformément (voir § 4.2), comme ceci :

```
> matrix(runif(20),4,5)
```

```
          [,1]       [,2]      [,3]      [,4]      [,5]
[1,] 0.9649945 0.79221909 0.8990586 0.5770601 0.9013598
[2,] 0.6660538 0.40556309 0.9614618 0.4216994 0.5588716
[3,] 0.8895701 0.74987611 0.7434252 0.4967354 0.7443293
[4,] 0.2195017 0.02501352 0.7314724 0.9356399 0.3201397
```

On aurait pu écrire aussi :

```
> M[1:4,1:5] <- runif(20)
```

Calcul matriciel

On peut effectuer dans R les opérations usuelles sur les matrices. Le produit matriciel est désigné par le symbole `%*%`. On transpose une matrice avec la fonction **t** et on calcule le déterminant d'une matrice carrée avec la fonction **det**. Voici quelques exemples :

```
> M <- array(1:8, dim=c(2,4))
> N <- array(1:12, dim=c(4,3))
> P <- M %*% N
> P
```

```
     [,1] [,2] [,3]
[1,]   50  114  178
[2,]   60  140  220
```

```
> t(P)
```

```
     [,1] [,2]
[1,]   50   60
[2,]  114  140
[3,]  178  220
```

```
> Q <- matrix(1:4,2,2)
> Q
```

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

```
> det(Q)
```

```
[1] -2
```

On détermine les valeurs propres et les vecteurs propres d'une matrice avec la fonction **eigen**. Par exemple :

```
> vp <- eigen(Q)
> vp
```

```
$values
[1]  5.3722813 -0.3722813
$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

Les valeurs propres sont stockées dans la composante *$values* et les vecteurs propres dans la composante *$vectors* de la valeur de retour. On peut les récupérer avec la notation `vp$values` ou `vp$vectors` respectivement :

```
> vp$values
```

```
[1]  5.3722813 -0.3722813
```

```
> vp$vectors
```

```
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

Si on veut obtenir seulement les valeurs propres, on peut préciser l'option *only.values* comme ceci :

```
> eigen(Q,only.values=TRUE)
```

Le produit scalaire est obtenu avec la fonction **crossprod** :

```
> x <- c(1,2,3)
> y <- c(1,-2,1)
> crossprod(x,y)
```

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

`crossprod(x,y)` est plus rapide que d'effectuer directement `t(x) %*% y`.

Équations linéaires

Pour résoudre un système d'équations linéaires $Ax = b$, on utilise la fonction **solve**. Cherchons par exemple à résoudre le système suivant :

$$\begin{cases} x_1 + 3x_2 = 5 \\ 2x_1 + 4x_2 = -4 \end{cases}$$

On procède comme ceci :

```
> A <- matrix(1:4,2,2)
> A
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> b <- c(5,-4)
> solve(A,b)
[1] -16   7
```

La solution est $x_1 = -16$ et $x_2 = 7$.

On peut aussi utiliser cette fonction pour inverser une matrice carrée. La commande $solve(A)$ calcule la matrice A^{-1} :

```
> solve(A)
```

```
        [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

2.7 Vecteurs logiques

On peut tester une condition sur tous les éléments d'un vecteur. Par exemple, sont-ils supérieurs à 11 :

```
> notes >= 11
```

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

La réponse de R est un *vecteur logique* dont les éléments valent soit TRUE, soit FALSE.

On peut passer un vecteur logique comme série d'indices afin de ne garder que les éléments correspondant à la valeur TRUE :

```
> notes[notes>=12]
```

```
[1] 12.5 14.5 15.0
```

Cette instruction renvoie un vecteur avec uniquement les notes qui sont supérieures à 12. Le vecteur *notes* d'origine n'a pas été modifié.

Un résultat analogue peut être obtenu avec la fonction **which** qui renvoie les *indices* d'un vecteur correspondant à une certaine condition. Par exemple :

```
> which(notes>=12)
```

```
[1] 2 3 9
```

Cela signifie que les notes d'indices 2, 3 et 9 sont supérieures à 12.

On construit des expressions logiques plus complexes avec les opérateurs & (ET) et | (OU). Par exemple, quelles sont les notes inférieures à 10 ou supérieures à 14 :

```
> notes[notes < 10 | notes > 14]
```

```
[1] 14.5  8.0  9.5 15.0
```

Quel est l'indice de la meilleurs note ?

```
> which(notes == max(notes))
```

```
[1] 9
```

```
> notes[9]
```

```
[1] 15
```

2.8 Valeurs particulières

R fait usage de symboles particuliers pour représenter les valeurs infinies ou indéterminées :

– les symboles `-Inf` et `Inf` représentent respectivement $-\infty$ et $+\infty$. Par exemple :

```
> c(-1,1)/0
[1] -Inf  Inf
```

– le symbole `NaN` signifie *Not a Number* et représente les valeurs indéterminées qui peuvent apparaître dans certaines opérations. Par exemple :

```
> 0/0
[1] NaN
> Inf - Inf
[1] NaN
```

– le symbole NA signifie *Not Available* et représente des valeurs qui ne sont pas disponibles (ou valeurs manquantes). Par exemple :

```
> V <- 1:3
> V[6] <- 100
> V
[1]   1   2   3  NA  NA 100
```

3 Graphiques avec R

On peut produire avec R des graphiques statistiques très variés. Les fonctions de dessin supportent en outre de très nombreuses options qui permettent d'affiner les paramètres. Cette section présente quelques exemples simples des possibilités graphiques de base de R. Plusieurs modules externes permettent d'étendre considérablement ces capacités mais ils ne sont pas abordés ici.

On considère, dans les exemples qui suivent, un vecteur de données comportant les notes de 30 étudiants :

```
> notes<-c(10.0, 10.0, 16.5, 10.0, 8.5, 14.5, 8.5, 7.0,
  10.0, 12.5, 12.5, 14.0, 10.5, 9.5, 11.5, 9.5, 8.0, 13.0,
  15.0, 12.5, 12.5, 8.5, 14.0, 11.5, 9.5, 9.5, 11.5, 10.5,
  13.0, 13.0)
> summary(notes)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   7.00    9.50   11.00   11.23   12.88   16.50
```

3.1 Histogrammes

On obtient un histogramme de l'échantillon de notes au moyen de la fonction **hist** comme ceci (voir la figure 1) :

```
> hist(notes)
```

On peut spécifier le nombre de barres dans l'histogramme au moyen d'un second argument. Pour avoir 10 barres, on écrit (voir la figure 2) :

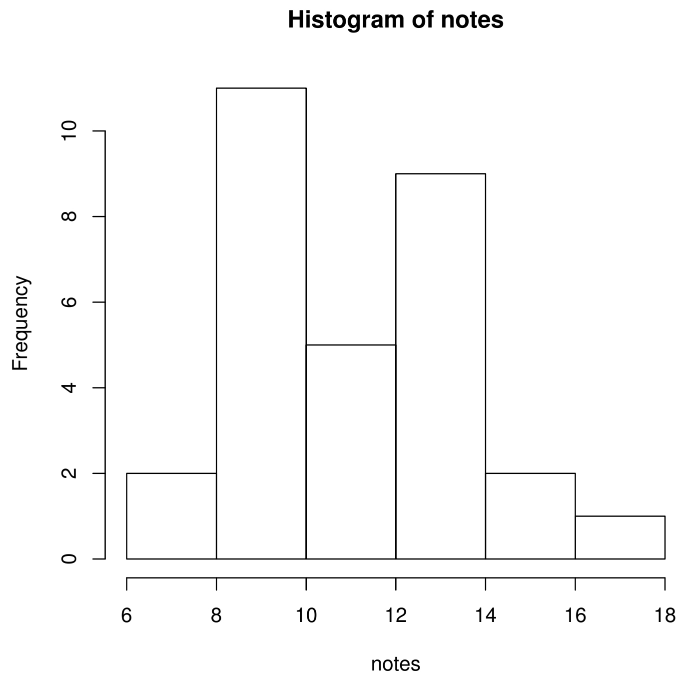

FIGURE 1 – hist(notes)

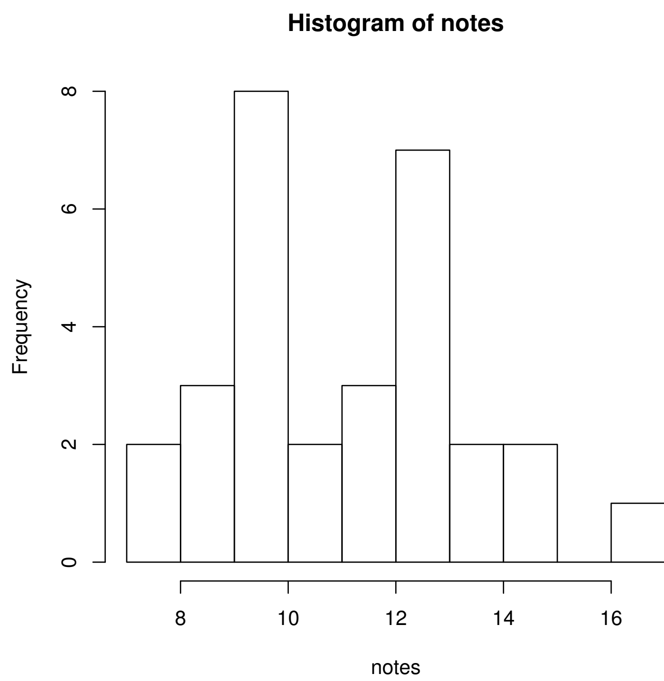

FIGURE 2 – hist(notes, 10)

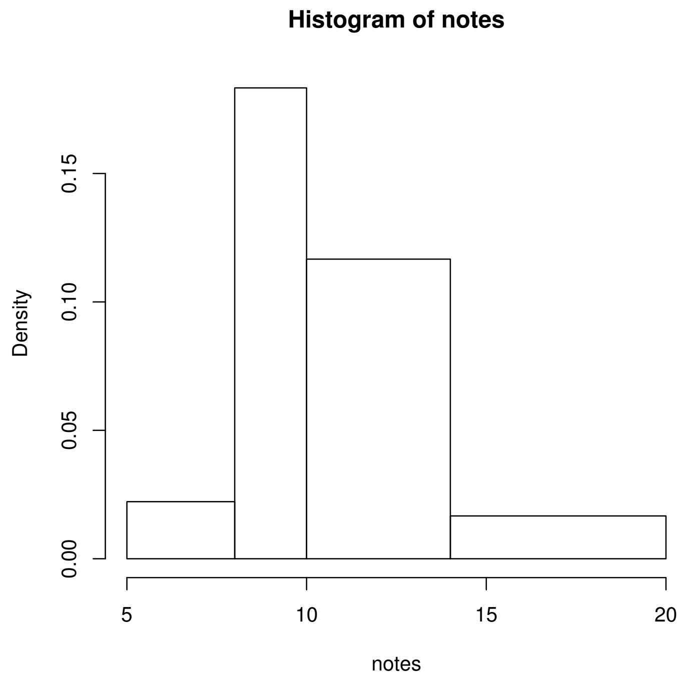

FIGURE 3 – hist(notes, breaks=c(5,8,10,14,20))

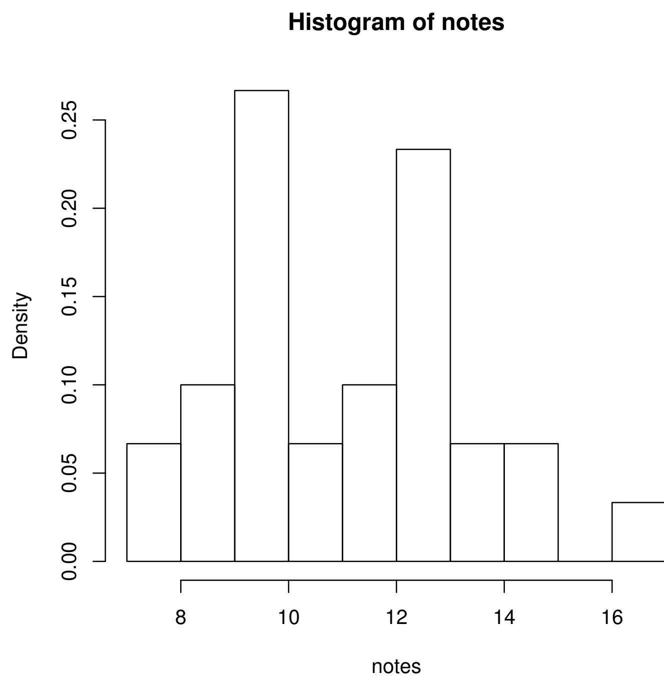

FIGURE 4 – hist(notes, 10, probability=TRUE)

```
> hist(notes, 10)
```

Ce second argument s'appelle ***breaks***. On aurait pu écrire la commande comme ceci :

```
> hist(notes, breaks=10)
```

Mais on peut aussi indiquer explicitement les points de subdivision en passant un vecteur à cette même option ***breaks***. Par exemple, on obtient des classes d'amplitude inégale avec la commande suivante (voir la figure 3) :

```
> hist(notes, breaks=c(5,8,10,14,20))
```

On peut aussi obtenir une représentation en proportions (plutôt qu'en fréquences) en spécifiant l'argument optionnel *probability* comme ceci :

```
> hist(notes, 10, probability=TRUE)
```

Cela modifie les valeurs représentées sur l'axe vertical (voir la figure 4).

3.2 Boîtes à moustaches

Une boîte à moustaches est obtenue au moyen de la fonction **boxplot**. La commande peut s'écrire :

```
> boxplot(notes,horizontal=TRUE,
          xlab="boxplot pour les notes")
```

On a ici rajouté une option *horizontal* pour spécifier l'orientation de la boîte et une option *xlab* (abréviation de "label pour l'axe des x") afin de placer une légende sur l'axe horizontal du graphique. Voir la figure 5.

3.3 Courbes

On fait des tracés de courbes au moyen des fonctions **plot**, **curve** ou **lines**. Les deux premières ont pour effet de démarrer un nouveau graphique (et donc d'effacer ce qui existe déjà) tandis que la dernière dessine en surimpression sur un graphique existant. Si on veut dessiner plusieurs courbes sur un même graphique, on dessinera la première avec les fonctions **plot** ou **curve** et les autres avec la fonction **lines**.

On passe un vecteur d'abscisses en premier argument, et le vecteur d'ordonnées correspondantes en deuxième argument. Par exemple, pour tracer le courbe d'équation $y = x^2 - x$ dans l'intervalle $[-4, 4]$, on crée tout d'abord un vecteur d'abscisses régulièrement espacées. On utilise ici la fonction **seq** pour créer des valeurs allant de -4 à 4 par pas de 0.1 :

```
> x <- seq(-4,4,1/10)
```

et on appelle la fonction **plot** comme ceci :

```
> plot(x, x^2-x)
```

Utilisée sous cette forme, la fonction **plot** dessine seulement les points correspondant aux abscisses de x. On obtiendra une courbe "continue" (en réalité, une suite de petits segments) en spécifiant la lettre l en troisième argument :

```
> plot(x, x^2-x, "l")
```

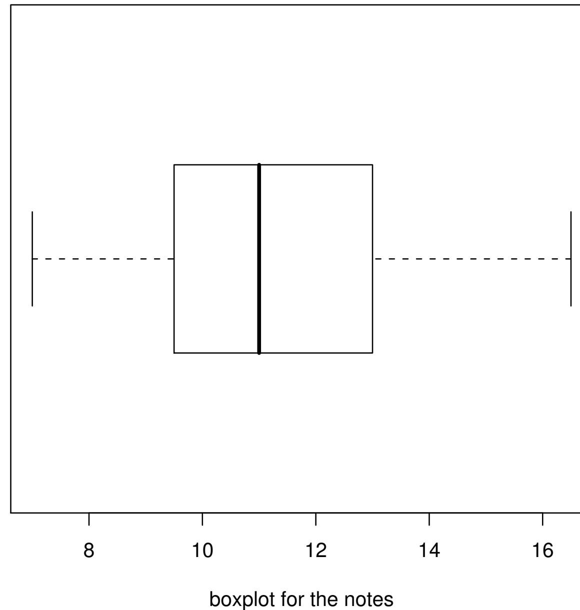

FIGURE 5 – boxplot(notes)

Autre exemple : voici comment tracer la fonction *sinus* entre 0 et 2π (voir la figure 6)

```
> x <- seq(0,2*pi,2*pi/100)
> plot(x, sin(x), "l")
```

Ces fonctions supportent des arguments supplémentaires. Voici comment spécifier le titre général du graphique ainsi que des étiquettes sur les deux axes. On utilise pour cela les arguments *main*, *xlab*, *ylab* respectivement :

```
> plot(x, sin(x), "l", main="Titre principal",
         xlab="axe des x",
         ylab="axe des y")
```

On notera que la fonction **plot** est un exemple typique de *fonction générique* qui possède des définitions différentes selon l'objet auquel on l'applique. Les exemples vus dans cette section concernent des vecteurs de valeurs numériques (les abscisses et les ordonnées). La fonction **plot** se comporte différemment si on l'applique à d'autres objets tels que des *séries temporelles* (*time series*) ou des régressions linéaires.

Une autre fonction utile pour tracer des courbes est la fonction **curve**. La différence avec la fonction **plot** est qu'on lui passe une fonction (ou le nom d'une fonction) directement au lieu des vecteurs d'abscisses et d'ordonnées. Par exemple, pour tracer la courbe de la fonction $y = \sin(x^2)$, on utilise la commande suivante :

```
> curve(sin(x^2), xlim=c(-pi,pi))
```

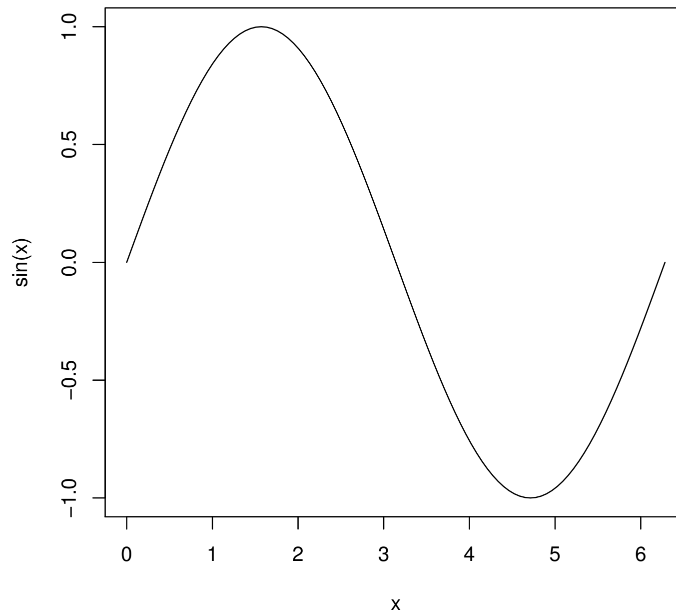

FIGURE 6 – plot(x,sin(x))

L'argument *xlim* sert ici à préciser l'intervalle sur lequel on veut tracer la fonction (en l'occurrence $[-\pi, \pi]$). Une syntaxe équivalente peut être obtenue avec les options *from* et *to* comme ceci :

```
> curve(sin(x^2), from = -pi, to = pi)
```

Voir la figure 7.

Enfin, il existe une fonction **points** qui permet de placer des points isolés sur un graphique : comme pour la fonction **plot**, on lui passe en argument les abscisses et les ordonnées des points concernés.

3.4 Densités

On peut utiliser la fonction **lines** pour ajouter une courbe à un graphique existant. L'exemple suivant montre comment ajouter une courbe de densités à l'histogramme dessiné précédemment (voir la figure 8).

La fonction **density** calcule une densité de probabilité et renvoie la liste des abscisses et des ordonnées. On lui applique donc directement la fonction **lines**.

```
> hist(notes,15,prob=TRUE)
> lines(density(notes))
```

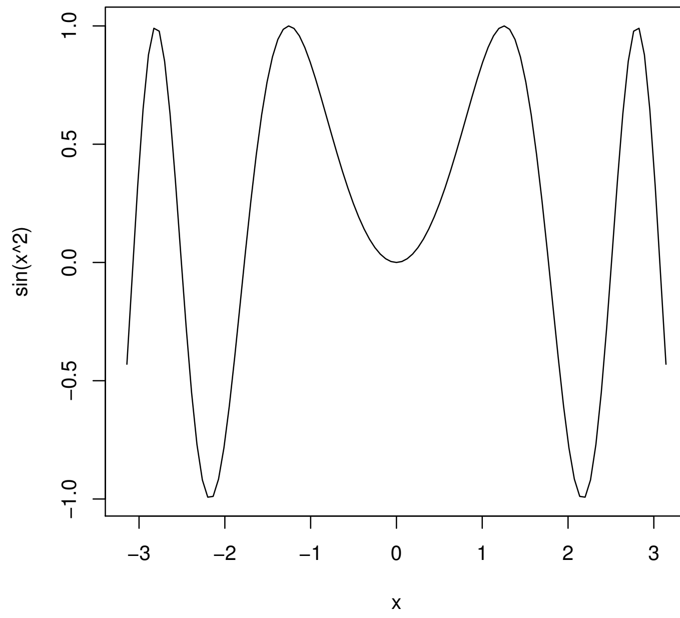

FIGURE 7 – Graphique $y = \sin(x^2)$

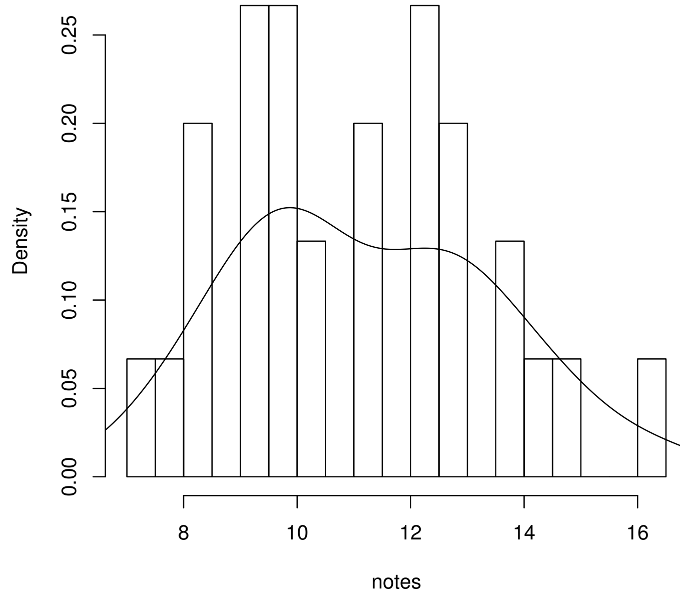

FIGURE 8 – lines(density(notes))

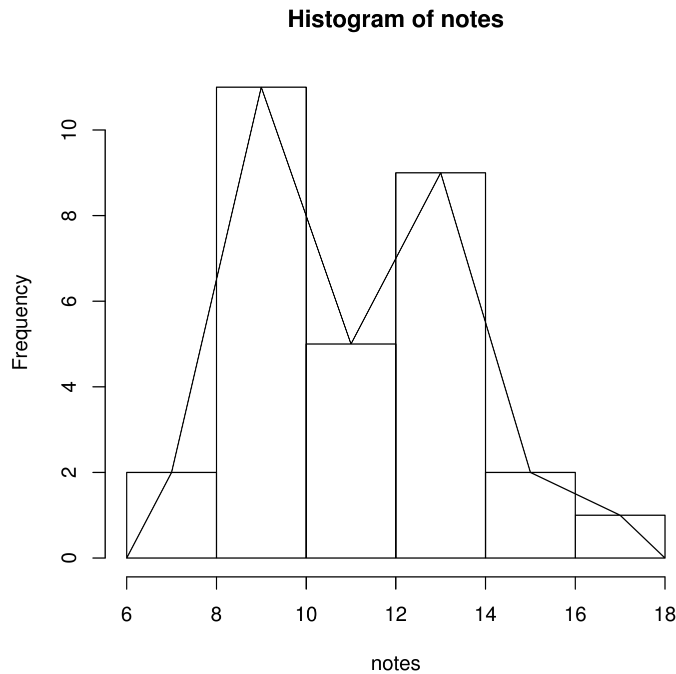

FIGURE 9 – Polygône de fréquences)

3.5 Polygône de fréquences

La fonction **lines** permet de manière générale de tracer des lignes brisées. Le premier argument de **lines** est un vecteur contenant les abscisses des points, et le second argument est un vecteur contenant leurs ordonnées.

Par exemple :

```
> abscisses <- c(1:6)
> ordonnees <- c(1.2,3.5,2,5.1,-1,3)
```

Voici une utilisation plus avancée où la fonction **lines** trace une ligne brisée, dite *polygône de fréquences*, passant par les milieux des sommets de chaque barre d'un histogramme (voir la figure 9).

```
> tmp = hist(notes)
> tmp
> lines(c(min(tmp$breaks),tmp$mids,max(tmp$breaks)),
        c(0,tmp$counts,0),type="l")
```

Les variables `$breaks`, `$mids` et `$counts` sont des composantes internes de l'*objet tmp* (qui est de classe *histogram*).

3.6 Lignes et quadrillages

La fonction **abline** permet de placer des droites sur un graphique déjà existant.

Il y a deux manières de l'utiliser :

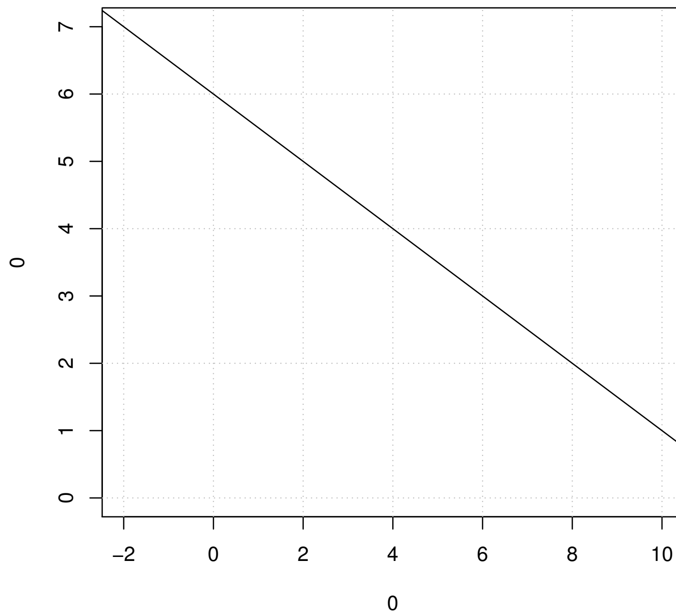

FIGURE 10 – Droite et quadrillage

– en spécifiant l'ordonnée à l'origine et la pente de la droite ;
– en utilisant les arguments *h* et *v* pour tracer des droites horizontales ou verticales.

Le code suivant commence par créer un graphique vide avec la fonction **plot** : l'option `type='n'` signifie qu'il ne faut rien tracer et les options *xlim* et *ylim* spécifient les intervalles en abscisse et en ordonnée :

```
> plot(0, 0, type='n', xlim=c(-2,10), ylim=c(0,7))
```

Puis on ajoute un quadrillage grisé (option *col*) et pointillé (option *lty*) de 2 en 2 et enfin la droite d'équation $y = -\frac{1}{2}x + 6$ comme ceci :

```
> abline(h=seq(0,7,2), v=seq(-2,10,2), col="gray",lty=3)
> abline(6,-1/2)
```

Voir la figure 10.

Noter qu'une méthode plus simple pour faire un quadrillage consiste à utiliser la fonction **grid**. On aurait pu écrire :

```
> grid(5,3)
```

Les arguments de la fonction **grid** sont le nombre de cellules que l'on veut dans la direction des abscisses et dans celle des ordonnées.

3.7 Paramètres graphiques

La fonction **par** permet d'effectuer divers réglages concernant les éléments des graphiques tels que les axes, les marques de graduation, les couleurs, les polices de caractères et leur taille, le type des lignes (continues, pointillées), etc.

Cette fonction dispose d'un très grand nombre d'options qui correspondent à tous ces aspects. Pour avoir la liste complète des options disponibles, il faut se reporter à la documentation de cette fonction au moyen de la commande :

```
> ?par
```

L'exemple suivant spécifie que l'axe des x utilise une échelle logarithmique et que la courbe doit être tracée en rouge :

```
> par(xlog=TRUE, col="red")
```

La fonction legend()

La fonction **legend** permet d'ajouter un bloc rectangulaire comportant des légendes. C'est utile pour distinguer plusieurs courbes sur un même graphique. Le code suivant trace les fonctions $\cos(2x)$ et $\sin(2x)$ sur le même graphique respectivement en traits pleins et en pointillés, et ajoute une légende (voir la figure 11) :

```
> x <- seq(-pi, pi, len = 100)
> plot(x, cos(2*x), type= 'l')
> lines(x, sin(2*x), lty = 2)
> legend(-1.9, 1, c("cos(2x)" ,"sin(2x)"), lty=1:2)
> abline(h=-1:1, v=pi*c(-1:1), col="gray")
```

La commande **abline** ajoute ici des lignes de quadrillage en gris.

La fonction text()

La fonction **text** permet de rajouter du texte sur un graphique. On doit indiquer un vecteur d'abscisses, un vecteur d'ordonnées et un vecteur de chaînes de caractères à placer aux endroits indiqués.

Par exemple, on peut ajouter l'indication *(0,0)* sous le point origine du graphique précédent comme ceci :

```
> text(0.25, -0.05, "(0,0)")
```

Voir encore la figure 11.

4 Échantillons avec R

4.1 Les data frames

Lorsqu'on étudie de grandes quantités de données, on les stocke généralement dans un fichier et on fait charger ce fichier par R dans une variable à laquelle on peut ensuite appliquer des fonctions ou des traitements statistiques.

Un des formats de fichier de données les plus utilisés est le *dataframe*. Dans un *dataframe*, les données sont organisées en lignes : chaque ligne représente une observation et les champs de données sont séparés par des tabulations. Par exemple, le fichier *notesEtudiants.txt* pourrait avoir le format suivant :

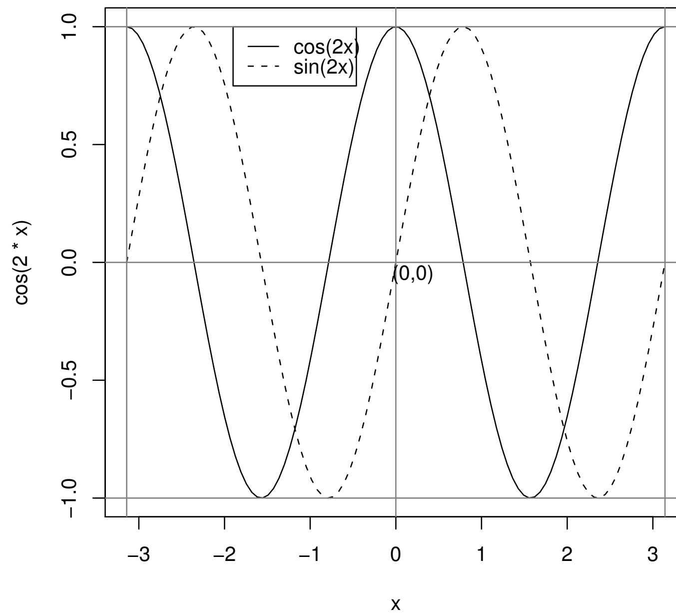

FIGURE 11 – Graphique avec légende

```
NomEtudiant    NoteCC    NoteExam
etudiant_01    11.5      8.0
etudiant_02    11.5      13.0
etudiant_03    18.0      9.0
etudiant_04    11.5      12.5
```

La première ligne s'appelle l'en-tête (*header* en anglais) et permet de donner un nom aux champs. Cet en-tête n'est pas obligatoire. Les données de chaque colonne peuvent être numériques ou logiques ou même indiquer une catégorie. Par exemple :

```
NomEtudiant    Poids     Tabac
etudiant_01    51.5      fumeur
etudiant_02    66.7      non-fumeur
etudiant_03    75.0      non-fumeur
etudiant_04    72.5      fumeur
```

On peut charger un fichier *dataframe* dans R au moyen de la fonction **read.table**, en indiquant en argument le chemin d'accès au fichier, comme ceci par exemple :

```
> donnees <- read.table("notesEtudiants.txt", header=TRUE)
```

Le *dataframe* est donc maintenant stocké dans une variable, appelée *donnees* dans cet exemple. L'objet *donnees* obtenu est un objet de classe *data.frame* (noter le point entre *data* et *frame*) comme on peut le vérifier avec la fonction **class** :

```
> class(donnees)
```

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

L'argument *header* de la fonction **read.table** a servi à indiquer à R que la première ligne du *dataframe* doit être interprétée comme un en-tête comportant le nom des variables (autrement dit le nom des colonnes) et non comme une ligne d'observations.

Pour connaître les noms des différents champs (colonnes) qui constituent un *dataframe*, on utilise la fonction **names** :

```
> names(donnees)
```

```
[1] "NomEtudiant"  "NoteCC"  "NoteExam"
```

Ces noms ont été lus par R dans la ligne d'en-tête.

Chaque série de données (correspondant à chaque colonne du *dataframe*) peut maintenant être appelée par le nom correspondant. Par exemple, *donnees$NoteCC* sera un vecteur contenant toutes les notes de la deuxième colonne. On obtiendra donc, par exemple, leur moyenne comme ceci :

```
> mean(donnees$NoteCC)
```

```
[1] 12.73333
```

Cette notation peut être encore simplifiée si on « attache » le *dataframe* :

```
> attach(donnees)
```

Dans ce cas, on peut désigner une colonne simplement par son nom, comme, par exemple, *NoteCC* plutôt que *donnees$NoteCC*. La fonction **detach** permet par la suite de détacher un *dataframe*.

Un *dataframe* peut être vu comme une matrice dont les colonnes peuvent avoir des modes différents (caractères, valeurs numériques, valeurs logiques, etc.). Les rangs et colonnes peuvent ainsi être extraits au moyen des conventions d'indice usuelles.

On peut aussi, inversement, créer un *dataframe* à partir de plusieurs vecteurs de données de même longueur au moyen de la fonction **data.frame**. Par exemple :

```
> noms=c("e1","e2","e3","e4")
> res=c(12,11,13.5,16)
> opt=c("oui","non","non","oui")
> df=data.frame(Noms=noms,Notes=res,Option=opt)
> df
```

```
  Noms Notes Option
1   e1  12.0    oui
2   e2  11.0    non
3   e3  13.5    non
4   e4  16.0    oui
```

On a ici spécifié des noms pour les colonnes dans la fonction **data.frame** : dans l'exemple précédent, les trois colonnes pourront donc être désignées respectivement par `df$Noms`, `df$Notes`, `df$Option`. Ce sont des vecteurs auxquels on peut appliquer les fonctions usuelles :

```
> mean(df$Notes)
```

```
[1] 13.125
```

Loi	Nom dans R	Arguments
beta	beta	shape1, shape2, ncp
binômiale	binom	size, prob
binômiale négative	nbinom	size, prob
Cauchy	cauchy	location, scale
exponentielle	exp	rate
F	f	df1, df1, ncp
gamma	gamma	shape, scale
géométrique	geom	prob
hypergéométrique	hyper	m, n, k
khi-2	chisq	df, ncp
log-normale	lnorm	meanlog, sdlog
logistique	logis	location, scale
normale de Gauss	norm	mean, sd
Poisson	pois	lambda
Student	t	df, ncp
uniforme	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

TABLE 3 – Lois de probabilité dans R

4.2 Échantillons d'une loi de probabilité

Les lois de probabilité courantes sont toutes disponibles dans R. Le tableau 3 indique les plus usuelles. La deuxième colonne de ce tableau indique le nom utilisé dans R pour désigner chacune de ces lois : par exemple norm désigne la loi normale de Gauss (loi normale centrée réduite), geom désigne la loi géométrique, exp la loi exponentielle, etc.

On utilise les préfixes *d*, *p*, *q*, *r*, que l'on place devant le nom d'une loi, pour obtenir les fonctions suivantes :

- avec le préfixe *d*, on obtient la densité de probabilité ;
- avec le préfixe *p*, on obtient la fonction de répartition. Elle est définie, de manière générale pour une probabilité P par
$$F(x) = P(X \leq x);$$
- avec le préfixe *q*, on obtient la fonction quantile (la réciproque de la fonction de répartition) ;
- avec le préfixe *r*, on obtient un échantillon aléatoire obéissant à cette loi.

Loi normale

Voici quelques exemples avec la loi normale norm : si on préfixe avec les lettres *d*, *p*, *q*, *r* on obtient les fonctions **dnorm**, **pnorm**, **qnorm**, **rnorm**. On calculera donc la densité en $x = 1.6$ comme ceci [2] :

2. *Rappel :* la densité de la loi normale $\mathcal{N}(0, 1)$ est la fonction $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$.

```
> dnorm(1.6)
[1] 0.1109208
```

On calcule la fonction de répartition au point 1.6 comme ceci :

```
> pnorm(1.6)
[1] 0.9452007
```

C'est la probabilité $P(X \leq 1.6)$ qu'une variable aléatoire X suivant la loi de Gauss soit inférieure ou égale à 1.6.

On calcule le quantile de 0.75 comme ceci :

```
> qnorm(0.75)
[1] 0.6744898
```

Le résultat (0.6744898) est le nombre a tel que la probabilité pour que X soit inférieure à a soit de 75% (donc 0.75). La fonction **qnorm** est la réciproque de **pnorm** comme on peut le vérifier avec l'instruction suivante qui redonne la valeur 0.75 :

```
> pnorm(0.6744898)
[1] 0.75
```

Enfin pour obtenir un échantillon de 20 valeurs aléatoires réparties selon une loi normale de Gauss, on écrit :

```
> rnorm(20)
[1]  0.7436211 -1.2551834 -0.1455878 -1.1775339  1.0448606
[6]  1.2029419  0.4428474 -0.1611212 -0.1836381  0.7840379
[11]  0.5378645 -0.5494457 -1.1387270  0.8846411 -1.0621926
[16] -0.5582748 -1.6658083 -1.1250059 -0.6409872 -0.8251526
```

Noter que ces nombres seront chaque fois différents : c'est pour cela qu'ils sont aléatoires !

Plus généralement, si on veut un échantillon de 6 valeurs centrées en 10 avec un écart type de 2, on écrit :

```
> rnorm(6,10,2)
[1] 8.279681 9.409680 8.315161 7.724435 5.664878 10.031787
```

Loi exponentielle

Le principe est le même pour toutes les autres lois. Par exemple, pour la loi exponentielle, on utilisera respectivement les fonctions **dexp**, **pexp**, **qexp**, **rexp**.

Voici le quantile correspondant à une probabilité de 95% avec la loi exponentielle de paramètre $\lambda = 1/10$:

```
> qexp(0.95, 0.1)
[1] 29.95732
```

Voici un exemple de densité de la loi exponentielle de paramètre $\lambda = 1/2500$ obtenue avec la fonction **dexp**. Il est représenté sur la figure 12 :

```
> x=rexp(100,1/2500)
> hist(x,prob=TRUE)
> curve(dexp(x,1/2500),add=TRUE)
```

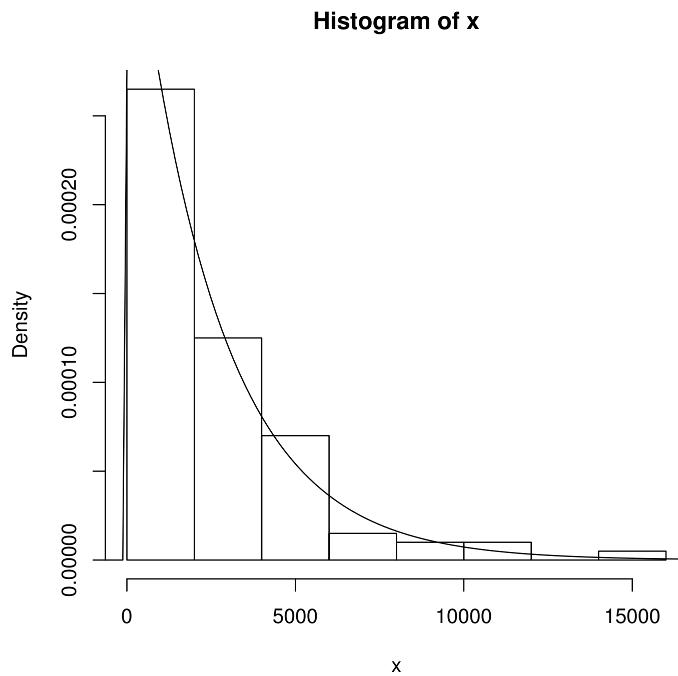

FIGURE 12 – Échantillon de la loi exponentielle

Loi uniforme

Pour la loi uniforme, on utilisera **dunif**, **punif**, **qunif**, **runif**. Voici, par exemple, 20 nombres uniformément répartis entre 0 et 1 :

```
> runif(20)
```

```
[1] 0.03779713 0.03588347 0.14798461 0.99118949 0.16882158
[6] 0.18167451 0.34934482 0.85962929 0.80629370 0.63290539
[11] 0.10946911 0.85036727 0.32149538 0.91549894 0.11318108
[16] 0.52903834 0.04714854 0.25949998 0.27320634 0.07063805
```

Pour obtenir 6 nombres uniformément répartis entre 15 et 25, on écrit :

```
> runif(6,15,25)
```

```
[1] 24.21007 16.41219 19.17167 15.08003 23.44818 22.03628
```

Loi binômiale

La loi binômiale $B(n, p)$ concerne le nombre de succès parmi n expériences indépendantes ayant une probabilité p de réussite. Dans R, elle est désignée par binom : on utilisera donc les fonctions **dbinom**, **pbinom**, **qbinom**, **rbinom** respectivement. Il faut passer les nombres n et p en argument. Par exemple, construisons un échantillon aléatoire de 15 résultats possibles lorsqu'on répète 10 fois une expérience ayant une probabilité de 0.25 de réussite :

```
> n = 10; p = .25
> rbinom(15,n,p)

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

4.3 Échantillons par tirage

On peut créer artificiellement des échantillons par tirage aléatoire dans un ensemble de données grâce à la fonction **sample**. C'est la technique dite du *bootstrapping*.

On passe en arguments le vecteur de données et la taille de l'échantillon que l'on veut obtenir. Cette fonction a aussi un argument optionnel, appelé *replace*, qui permet de préciser si le tirage se fait avec ou sans remise (par défaut, il est sans remise). S'il n'y a pas remise, il faut bien sûr que la taille de l'échantillon demandé soit inférieure à la taille des données.

Par exemple :

```
> x <- c(3, 4, 3, 3, 1, 4, 7, 8, 7, 4, 7, 6, 8, 1, 6)
> sample(x, 50, replace=TRUE)

 [1] 1 8 7 3 7 8 4 8 6 8 6 3 6 1 4 3 1 4 1 8 3 7 3 4 8 7 1
[30] 8 6 1 7 1 7 7 4 8 7 7 4 7 7 7 4 4 7 1 3 8 4 6
```

5 Tests statistiques avec R

5.1 Tests disponibles

Par convention, la plupart des fonctions qui effectuent des tests statistiques sur des ensembles de données ont un nom qui se termine en *.test*. On peut en obtenir la liste avec la fonction **apropos** comme ceci :

```
> apropos(".test")
 [1] "Box.test"          "PP.test"               "ansari.test"
 [4] "bartlett.test"     "binom.test"            "chisq.test"
 [7] "cor.test"          "fisher.test"           "fligner.test"
[10] "friedman.test"     "kruskal.test"          "ks.test"
[13] "mantelhaen.test"   "mauchley.test"         "mcnemar.test"
[16] "mood.test"         "oneway.test"           "pairwise.prop.test"
[19] "pairwise.t.test"   "pairwise.wilcox.test"  "power.anova.test"
[22] "power.prop.test"   "power.t.test"          "prop.test"
[25] "prop.trend.test"   "quade.test"            "shapiro.test"
[28] "t.test"            "var.test"              "wilcox.test"
```

On reconnaît en particulier :
- **t.test** pour le test de Student ;
- **chisq.test** pour le test du χ^2 ;
- **fisher.test** pour le test exact de Fisher sur des tables de contingnece ;
- **ks.test** pour le test de Kolmogorov-Smirnov ;
- **cor.test** pour les test de *corrélation* entre échantillons appariés (Pearson, Kendall ou Spearman) ;
- **binom.test** pour le test binômial.

La liste ci-dessus n'est pas exhaustive : on peut se procurer de nombreuses autres implémentations de tests sur l'Internet (sites CRAN) en chargeant des modules complémentaires (voir § 6.3).

5.2 Exemples de tests

5.2.1 Test d'ajustement du χ^2

On lance un dé 60 fois et on obtient les résultats suivants :

Faces	1	2	3	4	5	6
Effectifs	14	7	5	11	7	16

Peut-on rejeter l'hypothèse H_0 que le dé n'est pas truqué ? Il s'agit d'un test d'ajustement pour lequel R possède la fonction **chisq.test** (*chisq* est l'abréviation de *chi square* qui désigne le χ^2).

On exécutera le test comme ceci :

```
> valeurs<-c(14, 7, 5, 11, 7, 16)
> chisq.test(valeurs)


Chi-squared test for given probabilities
data:  obs
X-squared = 11.6, df = 5, p-value = 0.0407
```

La p-valeur calculée par le test est 0.0407. On rejette l'hypothèse H_0 (avec un risque 0.05 de le faire à tort) car $0.0407 < 0.05$.

5.2.2 Test de proportions

On interroge 100 personnes au hasard pour tester une opinion (réponse possible *oui* ou *non*) et on constate que 42 personnes ont répondu *oui*. Cela est-il en accord avec l'hypothèse que la vraie proportion est 50%.

L'hypothèse nulle est :

- $H_0 : p = 0.5$

L'hypothèse alternative serait $p \neq 0.5$ donc il s'agit d'un test bilatéral. On exécute le test au moyen de la fonction **prop.test** comme ceci :

```
> prop.test(42,100,p=.5)


1-sample proportions test with continuity correction
data:  42 out of 100, null probability 0.5
X-squared = 2.25, df = 1, p-value = 0.1336
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.3233236 0.5228954
sample estimates:
   p
0.42
```

Au seuil $\alpha = 0.05$, on acceptera l'hypothèse H_0 car la p-valeur 0.1336 est supérieure à 0.05. La p-valeur représente ici la probabilité que 42% des personnes répondent *oui* lorsque la vraie valeur est 0.05. Cette proportion est de 13.36%, ce qui n'est pas faible.

On recommence maintenant le test avec 1000 personnes parmi lesquelles 420 ont répondu *oui*.

```
> prop.test(420,1000,p=.5)


        1-sample proportions test with continuity correction
data:  420 out of 1000, null probability 0.5
X-squared = 25.281, df = 1, p-value = 4.956e-07
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.3892796 0.4513427
sample estimates:
   p
0.42
```

Cette fois la p-valeur est très petite (0.0000004956) et l'hypothèse nulle doit cette fois être rejetée. Cela illustre le fait que la p-valeur ne dépend pas seulement de la proportion mais aussi de la taille n de l'échantillon. En particulier, l'écart-type diminue quand n augmente.

5.2.3 Test de Student

Ce test utilise la variable de décision :

$$t = \frac{\bar{X} - \mu}{s/\sqrt{n}}$$

On l'exécute dans R au moyen de la fonction **t.test**.

Par exemple, supposons qu'une personne se pèse régulièrement et obtienne les résultats suivants :

72.5	73	71.5	72.5	72	71.5	71.5	73	71.5	74.5

Le poids idéal compte-tenu de la taille est de 72. L'échantillon est-il conforme avec ce poids idéal ?

On exécutera le test comme ceci :

```
> poids <- c(72.5, 73, 71.5, 72.5, 72, 71.5, 71.5, 73, 71.5, 74.5)
> t.test(poids, mu=72)

        One Sample t-test
data:  poids
t = 1.1372, df = 9, p-value = 0.2848
alternative hypothesis: true mean is not equal to 72
95 percent confidence interval:
 71.65378 73.04622
sample estimates:
mean of x
        72.35
```

La p-valeur calculée par le test (0.2848) est supérieure à 0.05. On accepte donc l'hypothèse nulle H_0.

6 Extensions de R

R est aussi un langage de programmation. On peut stocker les instructions dans des fichiers que l'on appelle des scripts et demander à R d'exécuter ces fichiers : R lit les instructions les unes à la suite des autres et les exécute comme si elles avaient été entrées sur la ligne de commande.

Le langage permet aussi de définir de nouvelles fonctions et donc d'étendre sa syntaxe. Les nouvelles fonctions peuvent être appelées ensuite aussi bien sur la ligne de commande que dans les scripts.

6.1 Créer de nouvelles fonctions

La syntaxe de définition d'une fonction obéit au modèle suivant :

nom_fonction <- **function**(*arguments*) *définition*

Une fonction peut avoir 0 ou plusieurs arguments. La définition est un bloc d'instructions en langage R placées entre une paire d'accolades.

Exemple 1

Si on a souvent besoin de centrer les données d'un échantillon x sur sa moyenne, on peut définir une fonction *centrer* comme ceci [3] :

```
> centrer <- function(x) {x-mean(x)}
```

On appliquera ensuite cette fonction à n'importe quel vecteur de la manière suivante :

```
> x = c(5, 1, 3, 2, 2, 1, 2, 1, 1, 5)
> centrer(x)
```

```
 [1]  2.7 -1.3  0.7 -0.3 -0.3 -1.3 -0.3 -1.3 -1.3  2.7
```

Exemple 2

L'instruction suivante simule, grâce à la fonction **sample**, un échantillon de 10 jets d'un dé bien équilibré :

```
> sample(1:6,10,replace=TRUE)
```

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

On peut la généraliser en une fonction appelée *Jet* qui prendra un argument n désignant le nombre de jets, comme ceci :

```
> Jet <- function(n) sample(1:6,n,replace=TRUE)
```

On peut maintenant appliquer la fonction avec n'importe quel argument entier. Par exemple :

3. Ce code a seulement valeur d'exemple. Il existe bien évidemment une fonction de R qui permet de centrer ou de réduire des vecteurs de données : c'est la fonction **scale**.

```
> Jet(5)

[1] 3 6 1 2 2
```

Exemple 3

On définit ici une fonction appelée *compareMoyenne* qui prend en argument deux échantillons y_1 et y_2 de tailles respectives n_1 et n_2 et calcule la variable statistique suivante (utilisée dans les tests de comparaison de moyenne entre deux petits échantillons) :

$$X = \frac{\bar{y}_1 - \bar{y}_2}{\sqrt{\Big(\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}\Big)\Big(\frac{1}{n_1} + \frac{1}{n_2}\Big)}}$$

La définition de la fonction sera :

```
> compareMoyenne <- function(y1, y2) {
            n1  <- length(y1)
            n2  <- length(y2)
            m1 <- mean(y1)
            m2 <- mean(y2)
            s1  <- var(y1)
            s2  <- var(y2)
            V   <- ((n1-1)*s1^2 + (n2-1)*s2^2)/(n1+n2-2)
            tst <- (m1 -  2)/sqrt(V*(1/n1 + 1/n2))
            tst
  }
```

6.2 Créer des scripts

Un script R est un fichier texte contenant des instructions en langage R. Plutôt que d'exécuter les instructions une à une depuis la ligne de commande de R, on fait lire et exécuter le script. C'est la fonction **source** qui permet de lire un fichier d'instructions. On lui passe en argument le chemin d'accès du fichier.

Par convention, les scripts R ont un nom qui se termine par une extension ".R" (ou parfois ".r"). Par exemple :

```
> source("/Documents/Scripts/monScript.R")
```

6.3 Les *packages*

La fonction **library** permet de charger en mémoire des bibliothèques supplémentaires, appelées extensions ou modules (en anglais *packages*) : ces bibliothèques étendent les capacités de R en définissant de nouvelles fonctions et en introduisant de nouvelles tables ou jeux de données.

Pour savoir quelles bibliothèques sont disponibles sur une installation, on utilise la fonction **library** sans arguments :

```
> library()
```

```
MASS      Main Package of Venables and Ripley's MASS
base      The R Base Package
boot      Bootstrap R (S-Plus) Functions (Canty)
class     Functions for Classification
cluster   Cluster Analysis Extended Rousseeuw et al.
datasets  The R Datasets Package
etc.
```

On trouve sur l'Internet des centaines de packages qui peuvent être ajoutés à la distribution standard de R. Chaque package traite de problèmes statistiques particuliers, implémente des algorithmes de résolution, des tests, des méthodes d'analyse, des fonctions graphiques, etc.

Pour avoir idée de ce qui existe, on peut aller sur le site CRAN (*Comprehensive R Archive Network*) à l'adresse suivante :

`http://mirror.ibcp.fr/pub/CRAN/`

Pour charger une bibliothèque particulière, on passe son nom en argument, comme par exemple :

```
> library(splines)
```

Pour avoir des informations sur le contenu d'une bibliothèque, on utilise l'argument *help*, comme ceci :

```
> library(help = splines)
```

Pour retirer une bibliothèque du chemin d'accès, on utilise la fonction **detach** et on écrit par exemple :

```
> detach("package:splines")
```

7 Configuration de R

7.1 Les options de R

La fonction **options** donne les valeurs des options de base de R ou permet de modifier leur valeur. Pour connaître toutes les valeurs, on utilise la fonction sans argument. Voici un (court) extrait du résultat :

```
> options()
```

```
$prompt
[1] "> "

$continue
[1] "+ "

$editor
[1] "vi"

$expressions
[1] 5000
```

```
$width
[1] 94

$digits
[1] 7

$echo
[1] TRUE

$verbose
[1] FALSE

...
```

Pour avoir la valeur d'une option particulière, on passe son nom en argument (entre guillemets). Par exemple :

```
> options("digits")

$digits
[1] 7
```

Pour modifier une valeur, on utilise la syntaxe *nom=valeur* comme argument de la fonction (cette fois, il n'y a pas de guillemets). Par exemple :

```
> options(digits=15)
```

Après ce changement, les calculs se feront avec 15 décimales au lieu de 7 :

```
> 58/7

[1] 8.28571428571429
```

7.2 Les chemins d'accès

La fonction **search** donne la liste des extensions qui ont été chargées (*packages*) et des *dataframes* qui ont été attachés. Cette liste commence toujours par `.GlobalEnv` et finit toujours par `package:base`. Lorsqu'un *dataframe* est attaché par la commande **attach**, il est placé en seconde position dans la liste. Par exemple :

```
> search()

[1] ".GlobalEnv"        "package:methods"   "package:stats"
[4] "package:graphics"  "package:grDevices" "package:utils"
[7] "package:datasets"  "Autoloads"         "package:base"

> attach(df)
> search()

 [1] ".GlobalEnv"        "df"                "package:methods"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "Autoloads"
[10] "package:base"
```

La fonction **searchpaths** donne la liste des chemins d'accès sur le système. Ce sont les répertoires dans lesquels R recherche les packages lorsqu'on tente de les charger avec une commande **library**. Par exemple :

```
> searchpaths()

[1] ".GlobalEnv"
[2] "/Library/Frameworks/R.framework/Resources/library/methods"
[3] "/Library/Frameworks/R.framework/Resources/library/stats"
[4] "/Library/Frameworks/R.framework/Resources/library/graphics"
etc.
```

Les fonctions **ls** et **objects** renvoient la liste de tous les objets connus de R. Un objet peut être supprimé de la mémoire au moyen de la commande **rm**.

8 Aide et tutoriels

8.1 Obtenir de l'aide avec R

Pour effectuer une recherche dans l'ensemble du système d'aide de R, on utilise la fonction **help.search**. Par exemple, si l'on ne sait pas s'il existe une implémentation du test de Kolmogorov-Smirnov, on exécute l'instruction suivante :

```
> help.search("kolmogorov")
```

La réponse mentionne la fonction **ks.test**.

Maintenant, pour obtenir des renseignements sur cette fonction, il suffit de taper un point d'intérrogation suivi du nom de la fonction, comme ceci :

```
> ?ks.test

Kolmogorov-Smirnov Tests
Description
Performs one or two sample Kolmogorov-Smirnov tests.
Usage
ks.test(x, y, ...,
        alternative = c("two.sided", "less", "greater"),
                exact = NULL)
etc.
```

Au lieu du point d'interrogation, on peut utiliser le fonction **help** qui est synonyme :

```
> help(ks.test)
```

Les fonctions **apropos** et **find** sont également utiles pour trouver des objets définis dans R. Par exemple, l'instruction suivante va chercher tous les objets dont le nom contient « *ks* » :

```
> apropos("ks")

[1] "cooks.distance" "ks.test"     "ksmooth"        "axTicks"
[5] "SweaveHooks"    "warpbreaks"  ".userHooksEnv"  "backsolve"
```

Pour connaître les tests disponibles, on peut rechercher les fonctions dont le nom contient *.test* (voir § 5.1) :

```
> apropos(".test")
```

La fonction **find** permet de trouver dans quel module (*package*) une fonction est définie. Par exemple :

```
> find("ks.test")
```

```
[1] "package:stats"
```

8.2 Tutoriels sur R

Voici une liste de quelques tutoriels en français ou en anglais pour en savoir plus sur R :

- R pour les débutants (Emmanuel Paradis)
  `http://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf`
  (81 pages, format pdf, 564 ko)

- Simple R : Using R for Introductory Statistics (John Verzani)
  `http://www.math.csi.cuny.edu/Statistics/R/simpleR/index.html`

- Statistics with R (Vincent Zoonekynd)
  `http://zoonek2.free.fr/UNIX/48_R/all.html`
  Il existe une version ancienne de cette page en français :
  `http://zoonek2.free.fr/UNIX/48_R_2004/all.html`

- Using R for Data Analysis and Graphics (J. H. Maindonald)
  `http://cran.r-project.org/doc/contrib/usingR.pdf`
  (104 pages, format pdf, 1.4 Mo)

- The R Guide (W. J. Owen)
  `http://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf`
  (59 pages, format pdf, 456 ko)

Index

khi2, 28

abline (pkg. *graphics*), 20, 22
abs (maths), 7
acos (maths), 7
acosh (maths), 7
Array, 8
asin (maths), 7
asinh (maths), 7
atan (maths), 7
atanh (maths), 7

beta (distribution), 25
binom (distribution), 25, 27
bootstrapping, 28
breaks (arg. de *hist*), 16

cauchy (distribution), 25
ceiling (maths), 7
chisq (distribution), 25
choose (maths), 7
col (arg. de *abline*), 21
Corrélation, 28
cos (maths), 7
cosh (maths), 7
curve (pkg. *graphics*), 17

data.frame, 23
data.frame (classe), 23

exp (distribution), 25
exp (maths), 7
Extensions, 32

f (distribution), 25
factorial (maths), 7
Fisher, 28
floor (maths), 7
fonction générique, 6, 17
from (arg. de *curve*), 18

gamma (distribution), 25
gamma (maths), 7
geom (distribution), 25
grid (pkg. *graphics*), 21

h (arg. de *abline*), 21
header, 24
histogram (classe), 20
horizontal, 16
hyper (distribution), 25

Inf, 13

Kendall, 28
Kolmogorov-Smirnov, 28

legend (pkg. *graphics*), 22
lnorm (distribution), 25
log (maths), 7
log10 (maths), 7
logis (distribution), 25
Lois de probabilité
    beta, 25
    binom, 25, 27
    cauchy, 25
    chisq, 25
    exp, 25
    f, 25
    gamma, 25
    geom, 25
    hyper, 25
    lnorm, 25
    logis, 25
    nbinom, 25
    norm, 25
    pois, 25
    t, 25
    unif, 25
    weibull, 25
    wilcox, 25
lty (arg. de *abline*), 21

main (arg. de *plot*), 17
Module, 36
Modules, 32

NA, 13
NaN, 13
nbinom (distribution), 25
norm (distribution), 25

only.values (option), 11

Package, 36
Packages, 32

Pearson, 28
plot (pkg. *graphics*), 18
points (pkg. *graphics*), 18
pois (distribution), 25
polygône de fréquences, 20
probability, 16

round (maths), 7

séries temporelles, 17
signif (maths), 7
sin (maths), 7
sinh (maths), 7
Spearman, 28
sqrt (maths), 7
Student, 28

t (distribution), 25
Tableau, 8
tan (maths), 7
tanh (maths), 7
text (pkg. *graphics*), 22
to (arg. de *curve*), 18
trunc (maths), 7

unif (distribution), 25

v (arg. de *abline*), 21
vecteur logique, 12

weibull (distribution), 25
wilcox (distribution), 25

xlab, 16
xlab (arg. de *plot*), 17
xlim (arg. de *plot*), 18, 21

ylab (arg. de *plot*), 17
ylim (arg. de *plot*), 21