Initiation au logiciel R

Références :

- Initiation à la statistique avec R, F. Bertrand et M. Maumy-Bertrand, ed. Dunod.
- Statistiques avec R, P-A. Cornillon *et al.*, ed. PUR.
- http://www-irma.u-strasbg.fr/ fbertran/enseignement/
- http://www.fabricemonna.com/wp-content/uploads/2011/09/Rpourlesstatophobes.pdf

1 Qu'est-ce-que R?

- R est un logiciel libre de traitement de données et d'analyses statistiques.
- R est également un langage de programmation.
- Homepage du projet : http://www.r-projet.org/
- R est gratuit, vous pouvez le télécharger et l'installer sur votre ordinateur.

2 Commencer avec R

2.1 L'interface

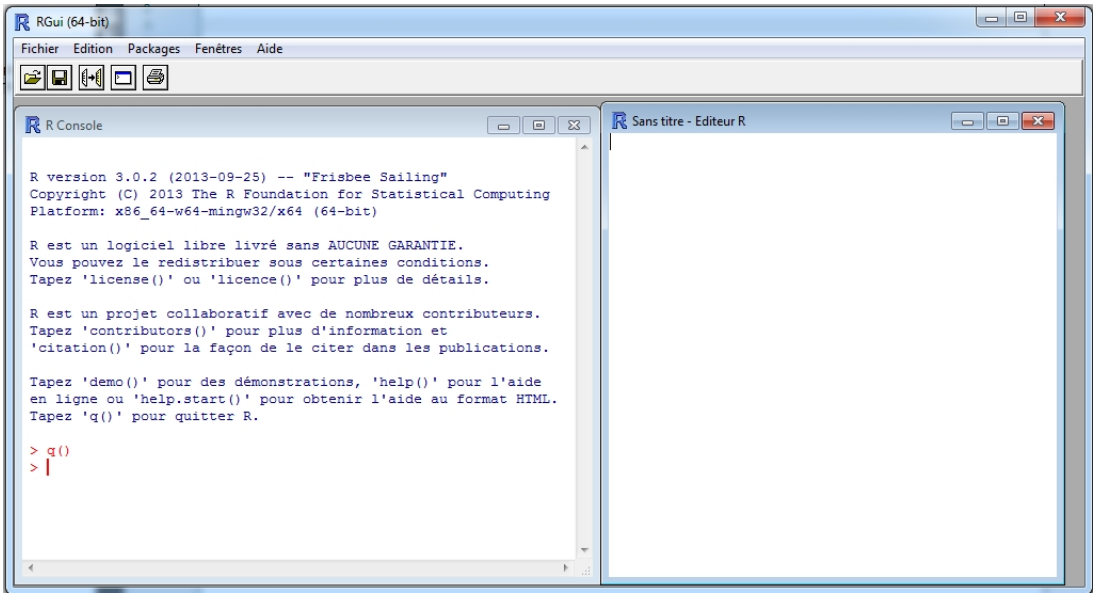


Figure 1 – RGui sous windows

Il existe de nombreuses interfaces graphiques pour le langage R. Nous allons utiliser celle fournie par défaut à savoir *RGui*. Avec cette interface, on a un terminal ou console (à gauche Figure 1) et un éditeur de texte (à droite Figure 1). Notons toutefois que l'éditeur de texte est mauvais (pas de couleur, pas d'indentation automatique, ...), c'est pourquoi il est plus conseillé d'utiliser un autre éditeur comme *Notepad++*. Cependant, en faisant ceci on perd une fonctionnalité de l'éditeur de R, à savoir pouvoir exécuter son script ou une partie avec la commande **ctrl+R**.

On observe un premier symbole dans le terminal, le prompt, noté `>`. Il signifie qu'il attend une commande. Vous pouvez aussi avoir le symbole `+`, qui signifie que la commande précédente est incomplète et donc R attend encore des instructions. Vous pouvez revenir au symbole précédent en utilisant la touche **echap**.

Si jamais vous êtes sous Linux, vous pouvez lancer R dans le terminal à l'aide de commande *R*. Vous aurez les mêmes symboles que sous windows et si vous voulez quitter, utiliser la commande `q()`.

Quant on quitte, R demande si on souhaite sauvegarder. Les sauvegardes se font en deux fichiers. Le premier en **.Rdata** contient les informations sur les variables utilisées. Le deuxième en **.Rhistory** contient les commandes utilisées.

Quelques remarques importantes : R tient compte de la casse (minuscules, majuscules), les décimales sont séparées par un point et on peut revenir à l'instruction précédente avec la touche flèche vers le haut.

2.2 L'aide

Voici les principales façons d'accéder à l'aide.

- Pour consulter l'aide pour une fonction, on peut utiliser soit `help(nom_fonction)` soit `?nom_fonction`. Par exemple pour consulter l'aide sur la fonction `mean` on peut utiliser `help(mean)`.
- On peut également consulter l'aide au format HTML avec la commande `help.start()`.
- On peut également accéder à l'aide sur le site du CRAN par la commande `RSiteSearch("calcul de la moyenne")`.
- Pour avoir l'aide sur un package il faut utiliser la commande `help(package="nomdupackage")`. Pour charger un package il faut utiliser la commande `install.packages("nomdupackage")`.

Ne pas hésiter à regarder sur Internet d'autres sites, à utiliser des livres, ...

2.3 Les scripts

Travailler avec le terminal suffit pour de petits programmes. Mais dès que l'on a besoin de plusieurs instructions, on utilise des scripts. Ils ont de plus de nombreux avantages :

- On gagne du temps. Les scripts sont facilement réutilisables et adaptables. Ils évitent donc de reprendre à zéro des lignes de commande déjà tapées.
- Si on a besoin de nombreuses lignes de code, elles seront plus simples à modifier par exemple en cas d'erreur.
- Les scripts sont faciles à échanger.
- Ils évitent également les pertes de données.

Dans le cas du langage R, il faut les enregistrer avec l'extension **.R**. Pour utiliser les instructions du script, il faut tout d'abord être dans le bon répertoire de travail (le répertoire où est sauvegardé le script). Sous windows, on va dans l'onglet **fichier** puis **Changer le répertoire courant ...**. Sous linux (fonctionne aussi sous windows), on utilise la fonction `setwd()`, la fonction `getwd()` permettant de connaître son répertoire de travail. On appelle ensuite le script avec l'instruction `source("fichier.R")`. Si vous voulez une exécution ligne à ligne, il faut ajouter l'option `echo` : `source("fichier.R",echo=T)`.

Il est important de prendre de bonnes habitudes quant à la rédaction des scripts. Premièrement, il ne faut pas hésiter à commenter ses programmes. Sous R, on utilise le `#`, c'est-à-dire que toutes les instructions après le `#` ne seront pas exécutées. Ensuite, sous R, on commence ses scripts par les instructions `rm(list=ls())` et `graphics.off()`. La première sert à effacer toutes les variables en mémoire et la seconde ferme toutes les fenêtres graphiques. Enfin, rendez votre programme lisible en regroupant les instructions de mêmes natures.

Il y a un exemple de script ci-dessous. Il correspond à certains exercices de régression linéaire traités en L1. La structure est identique pour chaque exercice, à savoir rentrer les données puis calculer les coefficients de régression et enfin tracer les courbes.

```
#Exercices de la feuille 2 (Regression linéaire, L1S2)
rm(list=ls()) #effacer les variables en mémoire
graphics.off() #fermer toutes les fenêtres graphiques ouvertes

pdf('td2.pdf') #enregistrer l'image
par(mfrow=c(3,3)) #afficher les 9 figures à venir sur 3 lignes, 3 colonnes

#-------------------------------------------------------------#
#                          EXERCICE 2                         #
#-------------------------------------------------------------#
#données
   rang=0:6
   montant=c(75,260,820,1650,2300,4000,5300)
   z=sqrt(montant)
#Calcul des coefficients de regression
   reg11=lm(montant~rang)
   reg12=lm(z~rang)
#afficher
   plot(montant~rang)
   abline(reg11)
   plot(z~rang)
   abline(reg12)
   plot(montant~rang)
   curve(reg12$coefficients[2]^2*x^2+2*reg12$coefficients[1]*reg12$coefficients[2]*x+reg12$
     coefficients[1],from=0,to=6,add=T)

#-------------------------------------------------------------#
#                          EXERCICE 3                         #
#-------------------------------------------------------------#
#données
   rang2=c(0,5,10,15,20,25,28)
   nb_pass=c(21.9,26.4,36.9,44.7,67,82,97.9)
   z2=log(nb_pass)
#Calcul des coefficients de regression
   reg21=lm(nb_pass~rang2)
   reg22=lm(z2~rang2)
#afficher
   plot(nb_pass~rang2)
   abline(reg21)
   plot(z2~rang2)
   abline(reg22)
   plot(nb_pass~rang2)
   curve(exp(reg22$coefficients[1]+reg22$coefficients[2]*x),from=0,to=28,add=T)

#-------------------------------------------------------------#
#                          EXERCICE 4                         #
#-------------------------------------------------------------#
#données
   volume=c(620,890,1013,1186,1454,1944,2313,3179)
   pression=c(6.7,4.3,3.48,2.644,1.997,1.35,1.1,0.7)
   ln_vol=log(volume)
   ln_press=log(pression)
#Calcul des coefficients de regression
   reg3=lm(ln_press~ln_vol)
   gamma=-reg3$coefficients[2]
   C=exp(reg3$coefficients[1])
#afficher
   plot(pression~volume)
   plot(ln_press~ln_vol)
   abline(reg3)
   plot(pression~volume)
   curve(C/(x^gamma),from=620,to=3179,add=T)
```

2.4 Les fonctions

Le but de cette partie est de montrer comment utiliser les fonctions sous R. On ne verra pas ici comment créer ses propres fonctions.

Le but d'une fonction est de réaliser une (ou plusieurs) tâche(s) précise. Par exemple la fonction `mean` calcule la moyenne d'une série statistique. Dans R, comme pour tout langage, les fonctions ont des entrées et des sorties. S'agissant des entrées, certaines sont obligatoires, d'autres sont optionnelles. Pour les connaître, il ne faut pas hésiter à regarder l'aide. Généralement, les optionnelles ont des valeurs par défaut. Pour les modifier, il n'y a pas d'ordre spécifique (comme cela peut être le cas pour d'autres langages comme Matlab) mais il faut préciser quel paramètre on modifie par son nom. Prenons un exemple. On a vu que pour exécuter un script on utilise la fonction `source`. Le plus simplement, on utilise la commande `source("nom_fichier.R")`. Mais cette fonction possède un paramètre optionnel appelé `echo` qui permet d'exécuter le script comme s'il était écrit ligne par ligne dans le terminal. Par défaut ce paramètre est désactivé, i.e., mis à la valeur `FALSE`. Pour l'activer, on le met à la valeur `TRUE` (ou `T`, les deux sont valables). Cela donne donc la commande `source("nom_fichier.R",echo=TRUE)` ou `source("nom_fichier.R",echo=T)`. Quand vous exécuterez vos scripts, n'hésitez pas à tester avec le paramètre `echo` activé et désactivé.

3 Les objets sous R

A la vue du titre de la section, on se pose la question : qu'est-ce-qu'un objet sous R ? C'est un espace dans lequel on peut stocker ce qui nous intéresse. On va ici s'attacher à présenter les principaux objets de R et les opérations basiques qu'on peut effectuer avec ceux-ci.

3.1 Création, affichage, suppression

La création par affectation s'effectue à l'aide d'un de ces trois opérateurs : `<-`, `->` ou `=`. Par exemple :

```
> b <- 25.6 # création de b
> x <- b # x reçoit la valeur de b
> x = b # x reçoit la valeur de b
> b -> x # x reçoit la valeur de b
```

Remarquons que R possède des jeux de données déjà existants. Ces données sont dans des packages. Chaque package contient des données spécifiques que l'on récupère à l'aide de la fonction `data`, par exemple `data(package="datasets")`.Notons la présence du fichier **iris** dont les données sont célèbres. On obtient des renseignement à l'aide des commandes `?iris` ou `help(iris)`. On peut le visualiser en entier à l'aide de la commande `iris`. On peut également lire des données à partir d'un fichier externe (par exemple excel, text,...) avec la fonction `read.table()`.

Il y a deux façons d'afficher un objet `x` à partir du terminal :

```
> x = 10
> print(x)
[1] 10
> x
[1] 10
```

Par contre, seule la fonction `print` permet d'afficher l'objet quand on écrit son programme dans un script.

R garde en mémoire les objets définis précédemment, il est donc utile de connaître les objets en mémoire ainsi que de supprimer ceux qui sont devenus inutiles. Pour connaître les objets de la session, deux commandes : `object()` et `ls()`. Pour supprimer des objets, on utilise la fonction `rm()` :

- `rm(x)` supprime l'objet `x`.
- `rm(objet1,objet2,objet3)` supprime plusieurs objets.
- `rm(list=ls(pattern=".*a.*"))` supprime une liste d'objets qui possèdent une partie de leur nom en commun, ici la lettre a.
- `rm(list=ls())` supprime tout les objets.

3.2 Le type d'un objet

Avant d'aborder les différents objets de R, on commence par présenter les différents types ou modes d'objets dans R. Il y en a cinq :

1. null (objet vide) : `NULL`
2. logical (booléen) : `TRUE, FALSE` ou `T, F`
3. numeric (réel) : `1, 5.431, pi, 1e-10`
4. complex (complexe) : `2+0i, 2i`
5. character (chaîne de caractères) : `'bonjour', "P"`

Pour connaître le mode d'un objet `x`, on utilise la commande `mode(x)`. On peut tester l'appartenance d'un objet `x` à un mode (le résultat est un booléen) :

`is.null(x)`	Teste si `x` est un objet vide
`is.logical(x)`	Teste si `x` est un booléen
`is.numeric(x)`	Teste si `x` est un réel
`is.complex(x)`	Teste si `x` est un complexe
`is.character(x)`	Teste si `x` est une chaîne de caractères

On peut également convertir des objets par les commandes `as.logical(x)`, `as.numeric(x)`, `as.complex(x)` et `as.character(x)`.

3.3 La valeur manquante

Il se peut que certaines données soient manquantes. Elles ne sont donc pas disponibles et R les note `NA` (Not Available, valeur manquante en français). Elle a ses propres règles de calcul. Par exemple

```
> x = NA
> print(x+1)
[1] NA
```

Pour savoir où sont les valeurs manquantes d'un objet `x`, on utilise la commande `is.na(x)` qui retourne un booléen de même longueur que `x`.

3.4 Les vecteurs

Un vecteur est composé d'un ensemble de valeurs appelées composantes, coordonnées ou éléments. Dans R, toutes les composantes d'un vecteur doivent être du même type. La fonction `length` donne le nombre d'éléments du vecteur.

3.4.1 Les vecteurs numériques

Voici quelques façons pour les construire :

- Par la fonction collecteur `c` :

```
> x = c(1.6,-8.3,45,16)
> x
[1]  1.6 -8.3 45.0 16.0
> x = c(x,6,c(15,9))
> x
[1]  1.6 -8.3 45.0 16.0  6.0 15.0  9.0
```

- Par l'opérateur séquence ": " :

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

- Par la fonction `seq` :

```
> seq(1,5,by=0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> seq(1,6,length=5)
[1] 1.00 2.25 3.50 4.75 6.00
```

- Par la fonction `rep` :

```
> rep(1,4)
[1] 1 1 1 1
> rep(c(1,2),each=3)
[1] 1 1 1 2 2 2
> rep(1:3,times=1:3)
[1] 1 2 2 3 3 3
```

- Par la fonction `scan`. On rentre les éléments au fur et à mesure. Un argument de type `n=5` spécifie le nombre d'éléments à collecter. Si ce paramètre n'est pas précisé, on arrête de collecter en mettant une valeur vide.

3.4.2 Les vecteurs de caractères

Comme précédemment, on peut créer des vecteurs de caractères en utilisant les fonctions `c` ou `rep`. Par exemple :

```
> x = c("A","AB","C2")
> x
[1] "A"  "AB" "C2"
> x = rep("A",5)
> x
[1] "A" "A" "A" "A" "A"
```

On peut également manipuler différents objets et les concaténer à l'aide de la fonction `paste` :

```
> paste("X",1:5,sep="-")
[1] "X-1" "X-2" "X-3" "X-4" "X-5"
> paste(c("X","Y"),1:5,sep=".")
[1] "X.1" "Y.2" "X.3" "Y.4" "X.5"
> paste(c("X","Y"),1:5,sep=".",collapse="+")
[1] "X.1+Y.2+X.3+Y.4+X.5"
```

Pour l'extraction on utilise la fonction `substr` :

```
> substr("statistiques",5,9)
[1] "istiq"
```

3.4.3 Les vecteurs logiques

On obtient généralement des vecteurs booléens à l'aide des opérateurs logiques : `>`, `>=`, `<`, `<=`, `==`, `!=`, etc. On peut aussi les générer avec les fonctions `seq`, `rep` et `c`. Ils sont utiles car ils permettent d'effectuer des sélections ou des opérations de conditions. Voici quelques exemples :

```
> 1>0
[1] TRUE
> x = c(-1,0,2)
> test = x>1
> test
[1] FALSE FALSE  TRUE
> (1+x^2)*(x>1)
[1] 0 0 5
```

La fonction `all` retourne `TRUE` si tous les éléments satisfont la condition et `FALSE` sinon. La fonction `any` retourne `TRUE` si un des éléments satisfait la condition et `FALSE` sinon :

```
> x = c(-1,0,2)
> all(x>1)
[1] FALSE
> any(x>1)
[1] TRUE
```

3.4.4 Sélection d'une partie d'un vecteur

On utilise pour cela l'opérateur `[ ]` et un vecteur de sélection : `x[vecteurdeselection]`.

Pour le vecteur de sélection, on peut choisir soit un vecteur d'entiers positifs, soit un vecteur d'entiers négatifs, soit un vecteur de logique. Dans le cas des entiers positifs, ce sont les indices des éléments à sélectionner. Ils doivent donc être compris entre 1 et `length(x)`. Le choix des entiers négatifs permet d'enlever les éléments du vecteur que l'on ne souhaite pas conserver. On peut également procéder par des vecteurs de logiques. On peut ainsi extraire des éléments particuliers. Voici quelques exemples :

```
> v = 1:100
> v[6] # donne le sixième élément de x
[1] 6
> v[6:8] # donne les 6ème, 7ème et 8ème éléments de v
[1] 6 7 8
> v[c(6,6,1,2)] # donne les 6ème, 6ème, 1er et 2ème éléments de v
[1] 6 6 1 2
> v[10:1] # donne les 10ème, ..., 1er éléments de v
 [1] 10  9  8  7  6  5  4  3  2  1
> w = 1:10
> w[-(1:5)] # donne w sans ses 5 premiers éléments
[1]  6  7  8  9 10
> w[-c(1,5)] # donne w sans le premier et le cinquième élément
```

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

```
> x = 1:15
> x[(x<5)]
```

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

```
> x[(x<5) & (x>=12)] # & signifie "et"
```

```
integer(0)
```

```
> x[(x<5) | (x>=12)] # | signifie "ou"
```

```
[1]  1  2  3  4 12 13 14 15
```

```
> # Sélection à partir des valeurs d'un autre vecteur
> temp = c(23,28,24,32)
> y = c(80,102,87,124)
> y[temp>25]
```

```
[1] 102 124
```

3.5 Les matrices

Comme pour les vecteurs, les matrices doivent avoir tout leurs éléments de même type. Remarquons que dans R, les vecteurs ne sont pas considérés comme des matrices. Dans le cas des matrices, la fonction `length` retourne le nombre d'éléments de la matrice et la fonction `mode` le mode des éléments. Pour connaître le nombre de lignes et de colonnes, on utilise la fonction `dim`. La façon la plus utilisée pour créer une matrice est l'utilisation de la fonction `matrix`. Par défaut, les valeurs sont rangées par colonne mais l'argument `byrow` permet de les ranger par lignes :

```
> m = matrix(c(1,2,8,45,16,9),ncol=2)
> m
     [,1] [,2]
[1,]    1   45
[2,]    2   16
[3,]    8    9
```

```
> m = matrix(1:9,nrow=3)
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
> m = matrix(1:9,nrow=3,byrow=TRUE)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Si la longueur du vecteur est différente du nombre d'éléments de la matrice, deux cas se présentent. Tout d'abord, s'il y a trop d'éléments, R ne prend que les premiers. Ensuite, s'il n'y en a pas assez, R répète le vecteur. Il est également possible de remplir une matrice avec un unique élément :

```
> m = matrix(1:5,nrow=3,ncol=3)
> m

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

```
> m = matrix(1,nrow=4,ncol=2)
> m

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

3.5.1 Sélection

Pour les matrices, la sélection d'éléments ou d'une partie de la matrice se fait de façon similaire aux vecteurs. Voici quelques exemples qui illustrent ces différentes méthodes :

```
> m = matrix(1:9,nrow=3,byrow=TRUE)
> m[2,3] # sélectionne l'élément présent à la 2ème ligne et 3ème colonne
[1] 6

> m[2,] # retourne sous forme d'un vecteur la 2ème ligne

[1] 4 5 6

> m[2, ,drop=F] # retourne sous forme d'une matrice la 2ème ligne

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

> m[,c(2,2,1)] # donne la 2nde, la 2nde et la 1ère colonne

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

> m[-1,] # supprime la 1ère ligne

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

> m[2:3,-2] # sélectionne la 2ème et la 3ème ligne sans la 2ème colonne

     [,1] [,2]
[1,]    4    6
[2,]    7    9

> # Sélectionner les colonnes pour lesquelles la valeur sur la première ligne
> # est strictement supérieure à 1 :
> m[,m[1,]>1]

     [,1] [,2]
[1,]    2    3
[2,]    5    6
[3,]    8    9

> m[m>3] # retourne un vecteur contenant les valeurs supérieures à 3

[1] 4 7 5 8 6 9

> m[m>3] = 0 # remplace les valeurs supérieures à 3 par 0
> m
```

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

3.5.2 Calculs sur les matrices

Dans R, les opérations sur les matrices sont effectuées élément par élément :

```
> m = matrix(1:4,ncol=2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> n = matrix(3:6,ncol=2,byrow=T)
> n
     [,1] [,2]
[1,]    3    4
[2,]    5    6
```

```
> m+n

     [,1] [,2]
[1,]    4    7
[2,]    7   10

> m*n

     [,1] [,2]
[1,]    3   12
[2,]   10   24
```

Voici d'autres fonctions utiles en algèbre linéaire :

Fonction	Description
`X%*%Y`	produit de matrices
`t()`	transposition d'une matrice
`diag(5)`	matrice identité d'ordre 5
`diag(vec)`	matrice diagonale avec les valeurs du vecteur `vec` sur la diagonale
`crosspod(X,Y)`	produit croisé (`t(X)%*%Y`)
`det()`	déterminant de la matrice
`eigen()`	diagonalisation d'une matrice

3.5.3 Opérations sur les lignes et les colonnes

Introduisons quelques fonctions utiles. Tout d'abord, on rappelle que les fonctions `dim`, `nrow` et `ncol` donnent des informations sur la dimension de la matrice. Ensuite, on peut concaténer deux matrices : par colonne avec la fonction `cbind` et par ligne avec la fonction `rbind` :

```
> cbind(c(1,2),c(3,4))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Une autre fonction intéressante est la fonction `apply`. Elle permet d'appliquer une fonction aux lignes (`MARGIN=1`) ou au colonnes (`MARGIN=2`) de la matrice. Par exemple :

```
> X = matrix(1:6,ncol=3)
> X
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
> apply(X,MARGIN=2,sum) # sommes par colonne
[1]  3  7 11
> apply(X,1,mean) # moyennes par ligne
[1] 3 4
```

3.6 Autres objets

3.6.1 Les facteurs

Pour pouvoir manipuler des données qualitatives, on utilise les facteurs. Leur longueur est donnée par la fonction `length`, leur mode par la fonction `mode` et leurs modalités par la fonction `levels`. Les facteurs peuvent être non ordonnés (couleur,sexe,...) ou ordonnés (niveaux, taille,...). Trois fonctions permettent de créer les facteurs :

- La fonction `factor`

```
> sexe = factor(c("M","M","F","F","F","M","F"))
> sexe
[1] M M F F F M F
Levels: F M

> sexe = factor(c(2,2,1,1,1,2,1),labels=c("femme","homme"))
> sexe

[1] homme homme femme femme femme homme femme
Levels: femme homme
```

- La fonction `ordered`

```
> taille = ordered(c("petit","petit","grand","grand","grand",
+ "moyen","grand","moyen"),
+ levels=c("petit","moyen","grand"))
> taille
[1] petit petit grand grand grand moyen grand moyen
Levels: petit < moyen < grand
```

- La fonction `as.factor`

```
> x = c(1:5,5:1)
> x
 [1] 1 2 3 4 5 5 4 3 2 1

> x.f = as.factor(x)
> x.f

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

On peut connaître les niveaux, le nombre de niveaux et l'effectif de `x.f` :

```
> levels(x.f)

[1] "1" "2" "3" "4" "5"

> nlevels(x.f)

[1] 5

> table(x.f)

x.f
1 2 3 4 5
2 2 2 2 2
```

3.6.2 Les listes

Une liste est un ensemble ordonné d'objets qui n'ont pas forcément le même type. On appelle ces objets composantes de la liste. Ces composantes peuvent avoir un nom. On peut connaître leur taille avec la fonction `length` et leur mode avec la fonction `mode`. Par rapport aux autres objets, les listes ont un attribut supplémentaire qu'on obtient avec la fonction `names`. Notons que toutes les fonctions qui retournent plusieurs objets le font sous la forme d'une liste.

On utilise la fonction `list` pour créer une liste :

```
> vecteur = seq(1,6,by=2)
> matrice = matrix(1:9,ncol=3)
> facteur = factor(c("M","M","F","F","F","M","F"))
> ordonne = ordered(c("petit","petit","grand","grand","grand",
+ "moyen","grand","moyen"),
+ levels=c("petit","moyen","grand"))
> maliste = list(vecteur,matrice,facteur,ordonne)
> length(maliste)
[1] 4
> mode(maliste)
[1] "list"
> maliste
[[1]]
[1] 1 3 5

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

[[3]]
[1] M M F F F M F
Levels: F M

[[4]]
[1] petit petit grand grand grand moyen grand moyen
Levels: petit < moyen < grand
```

On peut nommer les composantes de la liste :

```
> names(maliste)
NULL
> names(maliste) = c("vec","mat","sexe","taille")
> names(maliste)
[1] "vec"    "mat"    "sexe"   "taille"
```

Pour extraire une composante de la liste, on a deux solutions. On peut utiliser `[[ ]]` ou utiliser le nom de la liste avec le symbole `$`. Il est également possible d'extraire plusieurs éléments d'une même liste. On utilisera alors `[ ]` plutôt que `[[ ]]` :

```
> maliste[[1]]
[1] 1 3 5
> maliste[[3]]
[1] M M F F F M F
Levels: F M
> maliste$sexe
[1] M M F F F M F
Levels: F M
> maliste[["sexe"]]
[1] M M F F F M F
Levels: F M
> maliste[c(2,3)]
$mat
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$sexe
[1] M M F F F M F
Levels: F M
```

Il y a quelques fonctions utiles pour les listes. Tout d'abord, la fonction `lapply` permet d'applique une fonction successivement à chaque composante. Ensuite, la fonction `c` concatène deux listes. Il existe un autre type de liste : les listes `dimnames`. Elles donne un nom aux lignes et colonnes d'une matrice :

```
> X = matrix(1:9,ncol=3)
> nomligne = c("ligne1","ligne2","ligne3")
> nomcol = c("col1","col2","col3")
> dimnames(X) = list(nomligne,nomcol)
> X
       col1 col2 col3
ligne1    1    4    7
ligne2    2    5    8
ligne3    3    6    9
```

Il existe des listes particulières appelées data-frames. Dans ce cas, toutes les composantes ont même longueur mais peuvent être de types différents. En statistiques, les tableau de données peuvent être assimilés à des data-frames. On les crée à partir de la fonction `data.frame` :

```
> v1 = c(1,6,8,5)
> v2 = c("b","a","b","a")
> df = data.frame(nom.var1 = v1, nom.var2 = v2)
> df
  nom.var1 nom.var2
1        1        b
2        6        a
3        8        b
4        5        a
```

4 Les structures de contrôle

On va juste rappeler les structures classiques de la programmation.

4.1 Les boucles (for ou while)

On retrouve les boucles classiques sous R. Tout d'abord, la boucle `for` s'écrit :

```
for (i in vecteur) {
      expr1
      expr2
      ...
}
```

Des exemples très simples pour illustrer la boucle `for` sont la somme des éléments d'un vecteur ou la calcul du factoriel d'un nombre :

```
> # Calcul de la somme d'un vecteur x
> x = 1:5
> s=0 # initialisation
> for (i in 1:length(x)) {
+ s=s+x[i]
+ }
> s
[1] 15
> # Calcul du factoriel d'un nombre n supérieur à 2
> n = 5
> fact=1 # initialisation
> for (j in 2:n) {
+ fact=fact*j
+ }
> fact
[1] 120
```

Ensuite, on a la boucle `while` :

```
while (condition) {
      expr1
      expr2
      ...
}
```

Les ordres `expr1`, `expr2`, ... sont effectués tant que la condition est vraie, celle-ci étant évaluée en début de boucle. Dès que la condition est fausse, la boucle est arrêtée. Voici un exemple :

```
> i = 0
> while (i<5) {
+ print(i)
+ i=i+1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
```

Il existe un autre type de boucle : `repeat`. La sortie est assurée par la commande `break`.

4.2 Les conditions (if, else)

On veut ici exécuter une commande sous condition c'est-à-dire la commande est exécutée si et seulement si la condition est vraie. On l'écrit ainsi :

```
if (condition) {
        expr1
        expr2
        ...
}
```

Voici un exemple :

```
> i = 1
> repeat {
+ print(i)
+ i=i+1
+ if (i>3) break }
[1] 1
[1] 2
[1] 3
```

A la suite du `if`, on peut ajouter la condition `else`. Si la condition est vraie, on exécute les commandes après le `if`, si la condition n'est pas vérifiée, on exécute les commandes après le `else` :

```
if (condition) {
        expr1
        expr2
        ...
} else {
        expr3
        expr4
        ...
}
```

Il est obligatoire que le `else` soit sur la même ligne que l'accolade fermante "}" du `if`.

5 Représenter les données

On donne ici uniquement les principales fonctions pour représenter les données. Si on a besoin de développer cette partie lors d'un prochain TP (peut-être pour la régression linéaire), on le fera à cette occasion.

Fonction	Description
`plot(x,y)`	Trace le nuage de point de `y` en fonction de `x`
`barchart`	Trace un diagramme
`barplot(x)`	Trace un diagramme en bâtons de `x`
`pie(x)`	Trace un diagramme circulaire de `x`
`hist`	Trace un diagramme avec des options pour définir les classes : • `nclass=...` pour fixer un nombre de classes • `br=c(...,...,...)` pour donner les bornes des classes Attention à ne pas utiliser `nclass` et `br` en même temps
`boxplot(x)`	Trace une boîte à moustache de `x`

A noter que par défaut R n'ouvre qu'une seule fenêtre graphique. Donc si on veut pouvoir afficher différents graphiques, il faut utiliser soit la commande `X11()` soit la commande `dev.new()` (et `dev.off()` pour fermer la fenêtre graphique active). Rappelons que la commande `graphics.off()` ferme toutes les fenêtres graphiques ouvertes.

6 Quelques fonctions statistiques

Le tableau ci-dessous donne les fonctions de base en statistiques :

Fonction	Description
`summary`	Donner divers paramètres statistiques
`cumcum`	Calculer les effectifs cumulés
`sum`	Calculer l'effectif total
`mean`	Calculer la moyenne
`max`	Calculer la valeur maximum
`min`	Calculer la valeur minimum
`range`	Calculer les valeurs minimum et maximum
`median`	Calculer la médiane
`quantile`	Calculer un quantile
`var`	Calculer la variance
`sd`	Calculer l'écart-type

Voici un autre tableau pour les lois de probabilités usuelles :

Nom de la distribution	Nom sous R	Paramètres	Valeurs par défaut
Binomiale	`binom`	`size`, `prob`	
Exponentielle	`exp`	1/`mean`	1
Fisher	`f`	`df1`, `df2`	
Géométrique	`geom`	`prob`	
Hypergéométrique	`hyper`	`m, n, k`	
Khi-deux	`chisq`	`df`	
Normale	`norm`	`mean, sd`	0, 1
Poisson	`pois`	`lambda`	
Student	`t`	`df`	
Uniforme	`unif`	`min, max`	0, 1

Pour toute ces distributions, on a quatre commandes préfixées par les lettres `d, p, q, r` suivi du nom de la distribution :

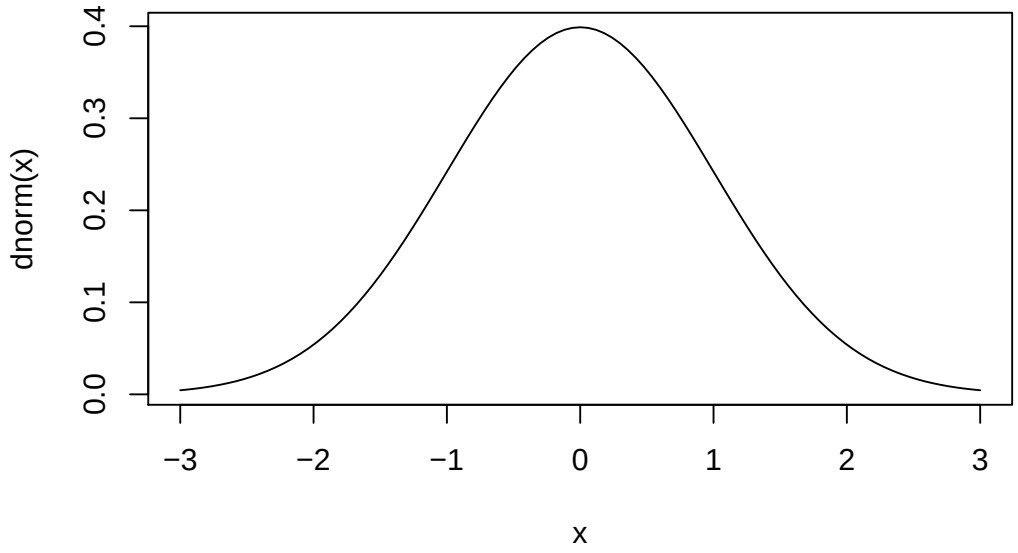
- `dnomdist` : C'est la fonction de densité $f_X(x)$ pour une distribution de probabilités continue et c'est $\mathbb{P}(X = k)$ pour une loi discrète.
- `pnomdist` : Il s'agit de $\mathbb{P}(X \leq k)$.
- `qnomdist` : C'est la fonction des quantiles, c'est-à-dire la valeur pour laquelle la fonction de répartition atteint une certaine probabilité.
- `rnomdist` : génère des réalisations aléatoires indépendantes de la distribution `nomdist`.

Remarque : Si on veut tracer la fonction `f` de `x=from` à `x=to`, on peut utiliser la fonction `curve` : `curve(f(x),from,to)`.

Par exemple prenons la loi normale :

```
> pnorm(1.64)
[1] 0.9494974
> qnorm(0.975)
[1] 1.959964
> rnorm(5,mean=2,sd=0.5)
```

```
[1] 2.403865 2.017267 1.698504 1.320467 2.101327
> curve(dnorm(x),from=-3,to=3)
```



7 Exercices

Exercice 1 :

1. Créer les trois vecteurs ci-dessous avec la fonction `rep` :

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

2. Créer le vecteur ci-dessous avec la fonction `paste` :

```
[1] "A1)" "A2)" "A3)" "A4)" "A5)" "A6)" "A7)" "A8)" "A9)"
```

3. Le vecteur `letters` contient les 26 lettres de l'alphabet. Trouver la position de la lettre `q` (à l'aide de la fonction `which` et `==`) dans l'alphabet et créer le vecteur de coordonnées `a1, b2, ...` jusqu'à `q??` où `??` désigne la position de `q`.

Exercice 2 :

1. Créer la matrice `mat` suivante (pour les noms de ligne et colonne, utiliser les fonctions `rownames` et `colnames`).

```
        col_1 col_2 col_3 col_4
ligne 1     1     5     5     0
ligne 2     0     5     6     1
ligne 3     3     0     3     3
ligne 4     4     4     4     2
```

2. Créer le vecteur contenant les éléments diagonaux de `mat` (utiliser la fonction `diag`).
3. Créer la matrice contenant uniquement les deux premières lignes de `mat`.
4. Créer la matrice contenant uniquement les deux dernières colonnes de `mat`.
5. Créer la matrice contenant toutes les colonnes de `mat` sauf la troisième.
6. Calculer le déterminant (`det`) puis inverser la matrice (fonction `solve`).

Exercice 3 :

1. On va prendre ici le jeu de données `ethanol` :

```
> library(lattice)
> data(ethanol)
> head(ethanol)
```

```
    NOx  C     E
1 3.741 12 0.907
2 2.295 12 0.761
3 1.498 12 1.108
4 2.881 12 1.016
5 0.760 12 1.189
6 3.120  9 1.001
```

   Calculer les statistiques de base des trois variables du jeu de données `ethanol` (utiliser la fonction `summary`).

2. Calculer les quartiles de chacune des trois variables (utiliser la fonction `apply` avec la fonction `quantile`).
3. Toujours avec la fonction `apply`, calculer tous les déciles de chacune des trois variables en utilisant l'argument `probs` de la fonction `quantile`.

Exercice 4 :
On reprend ici la matrice `mat` de l'exercice 2.

1. Créer une nouvelle matrice qui ne contient que les colonnes de `mat` dont tous les éléments sont strictement inférieurs à 6 (Utiliser les fonctions `apply` et `all`).
2. Créer une nouvelle matrice qui ne contient que les lignes de `mat` qui ne présentent aucun 0.

Exercice 5 :
On va utiliser ici le jeu de données `iris`. On va voir ce que fait la fonction `var`. Considérons la première colonne de `iris` que l'on notera `x`.

1. Calculer la moyenne $\bar{\texttt{x}}$.
2. Calculer $\sigma_1^2 =$ `var(x)`.
3. Calculer $\sigma_2^2 = \frac{1}{n}\sum_{i=1}^{n} \texttt{x}_i^2 - \bar{\texttt{x}}^2$.
4. Calculer $\sigma_3^2 = \frac{n}{n-1}\sigma_2^2$.
5. Conclure.

Exercice 6 :

1. Tracer la fonction sinus entre 0 et 2π (utiliser `pi`) et ajouter un titre (fonction `title`).
2. Tracer la courbe de la loi normale entre -4 et 4 (`dnorm`).
3. Tracer sur le même graphe les loi de Student à 5 et 30 degrés de liberté. Utiliser la fonction `curve` et une couleur différente pour chaque courbe. Ajouter une légende (optionnel).

Exercice 7 : (Théorie des grands nombres)

1. On peut fixer la graine du générateur aléatoire avec la fonction `set.seed`. Fixer la à 123. Simuler un échantillon $(X_1, \ldots, X_{1000})$ provenant d'une loi de Bernouilli de paramètre $p = 0.6$.
2. Calculer les moyennes successives $M_k = S_k/k$ où $S_k = \sum_{i=1}^{k} X_i$ (utiliser la fonction `cumsum`). Tracer M_k en fonction de k puis ajouter la droite horizontale d'équation $y = 0.6$ (`abline`).

Exercice 8 : (Théorème Central Limite)

1. Soit $X_1, X_2, \ldots, X_N$ indépendantes et suivant la loi de Bernouilli $\mathcal{B}(p)$. Rappeler la loi suivie par $S_N = \sum_{i=1}^{N} X_i$. Donner sa moyenne et son écart-type.
2. Prenons $p = 0.5$ et $N = 10$. Simuler $n = 1000$ réalisations $S_1, \ldots, S_{1000}$ d'une loi binomiale (fonction `rbinom`) de paramètres N et p. Ranger dans un vecteur `U10` les quantités $\frac{S_i - N\times p}{\sqrt{N\times p\times(1-p)}}$. Faire de même avec $N = 30$ et $N = 1000$ pour obtenir deux nouveaux vecteurs `U30` et `U1000`.
3. Représenter sur une même fenêtre (`par(mfrow=...)`) les histogrammes de `U10`, `U30` et `U1000` en superposant à chaque fois la densité de la loi normale centrée réduite (`dnorm`).
4. Refaire les questions précédentes mais en prenant une loi de Poisson.