Université de Strasbourg
M1 - Mathématiques fondamentales + Magistère
Statistique

Ségolen Geffray
geffray@math.unistra.fr
Année 2020/2021

Prise en main du logiciel R

• Le logiciel `R` est né en 1995 des travaux de Robert Gentleman et Ross Ihaka (Département de Statistique de l'Université d'Auckland, Nouvelle-Zélande). C'est un logiciel libre (ou *Open Source* en anglais), sous licence GPL (General Public License), utilisable sous différents systèmes d'exploitation: Windows, MacOS, Unix, Linux. C'est un clone gratuit du logiciel S-Plus développé et commercialisé en 1988 par la start-up Statistical Sciences fondée et possédée par le professeur R. Douglas Martin (Département de Statistique de l'Université de Washington, Seattle, USA) et depuis 2008 par Tibco Software.
Le logiciel `R` est un interpréteur de commandes permettant d'effectuer des analyses statistiques et de produire des graphiques. C'est aussi un langage de programmation orienté objet (il existe trois classes d'objet: S3, S4 et R5).
Il est téléchargeable gratuitement à l'adresse suivante:
`http://cran.r-project.org/`
Pour faciliter l'écriture et le développement de programmes, il est recommandé d'utiliser un éditeur comme `RStudio` téléchargeable gratuitement à l'adresse suivante:
`http://www.rstudio.com/products/rstudio/download/`
Sous Windows, vous pouvez également utiliser l'éditeur `Tinn-R` téléchargeable à l'adresse suivante:
`http://nbcgib.uesc.br/lec/software/editores/tinn-r/en#h10-download`
NB: il est fortement recommandé d'installer l'éditeur choisi seulement après avoir installé `R`.
Vous créerez alors un fichier `mywork.R` dans lequel vous enregistrerez votre travail.
Pour la lisibilité du programme, il est impératif de le structurer selon les différentes étapes et de le commenter. Le sigle à taper pour mettre des lignes en commentaire est `#`. Pour quitter le logiciel, l'instruction à taper est `q()`.

• Par défaut, les commandes sont saisies en mode interactif dans la fenêtre `R Console` où elles apparaissent en rouge. A l'exécution des commandes, les résultats apparaissent en bleu, en mode texte. Le symbole `>` apparaît automatiquement au début de chaque ligne de commande. Le symbole `+` apparaît en début de ligne si la précédente est incomplète. Si plusieurs instructions sont tapées sur une même ligne, il faut les séparer par un point-virgule ";".

A l'exécution de `R`, des fichiers `.RData` et `.Rhistory` sont automatiquement créés dans le répertoire courant. Lorsqu'on quitte `R`, les objets créés pendant la session peuvent être sauvegardés dans le fichier `.RData` (fichier binaire) tandis que la suite des commandes saisies peut être sauvegardée dans le fichier `.Rhistory` (fichier texte). Ainsi, lorsqu'on relance `R` depuis le même répertoire, le fichier `.RData` est automatiquement chargé et l'on retrouve l'intégralité des objets que l'on a précédemment créés. On peut visualiser la suite de commandes que l'on a tapées à l'aide de l'instruction `history()` dans `R` ou en consultant simplement le fichier `.Rhistory`. Enfin, la commande `ls()` permet de visualiser la liste des objets créés et la commande `rm()`

permet de détruire des objets.

L'instruction `getwd()` permet de connaître le répertoire de travail utilisé par `mywork.R`. Pour changer de répertoire de travail, on peut utiliser l'instruction `setwd("C:/Data/mydocs")` (attention au sens de la barre oblique, inversé par rapport à l'écriture classique des chemins de répertoire).

Si les commandes sont stockées dans un fichier externe `mywork.R` situé dans le répertoire de travail utilisé par `R`, alors elles peuvent être exécutées avec la commande `source("mywork.r")`.

• Certains mots sont réservés et ne doivent pas être utilisés comme noms de variable ou de fonction. C'est le cas de:

```
FALSE TRUE NA NaN Inf NULL
break else for function if next pi repeat while
```

Il vaut mieux éviter de nommer un objet `T` ou `F` pour éviter la confusion avec la version abrégée de `TRUE` ou `FALSE`.

• Trouver de l'aide
La commande `help(dnorm)` ouvre une fenêtre externe d'aide qui renseigne sur la fonction `dnorm`. La commande `?dnorm` produit le même résultat.
Si l'on ne connaît pas le nom exact de la fonction, on peut utiliser la fonction `apropos` en indiquant une partie du nom de la fonction recherchée entre cotes par exemple `apropos('norm')`.
Si l'on ne connaît pas du tout le nom de la fonction, on peut utiliser la fonction `help.search` en indiquant un mot clé entre guillemets simples (ou doubles) et droits comme par exemple `help.search('normal')`.
Enfin, `R` fait la différence entre les majuscules et les minuscules. Ainsi, 'y' et 'Y' ne sont pas équivalents, pas plus que 'Car' et 'car'.

• Les librairies
Les librairies ("packages") de base fournies lors de l'installation de `R` permettent d'effectuer un certain nombre d'analyses. Cependant, il arrive que l'on souhaite ajouter de nouveaux outils pour des analyses plus poussées. Pour les rajouter, il faut cliquer sur l'onglet "Packages" et commencer par choisir un site miroir proche du lieu où vous trouvez (depuis la France, vous pouvez choisir par exemple l'un des deux miroirs lyonnais). Ensuite, recliquez sur l'onglet "Packages" et opter cette fois pour l'action "installer le package". Choisissez alors le package souhaité parmi la liste proposée des packages. Une fois que le package est installé, n'oubliez pas de le charger dans votre espace de travail via l'instruction à taper dans la console `library(mypackage)`.

• Objets et classes:
Les éléments de base de `R` sont des objets qui peuvent être des données (vecteurs, matrices), des fonctions, des graphiques... Les objets `R` diffèrent par leur mode (qui décrit leur contenu) et leur classe. Les différents modes sont:

```
NULL, logical, numeric, complex, character, NA (= Non Available),
NaN (= Not A Number)
```

Les modes `NULL` et `NA` désignent respectivement l'absence de valeur (ie un ensemble vide) et le codage par défaut d'une valeur manquante.
Les principales classes d'objets sont:

`vector, matrix, list, factor, array, time-series, data.frame`

Les vecteurs, matrices, tableaux, variables catégorielles et séries chronologiques sont de mode homogène. Par contre, les listes ou les tableaux de données peuvent être de mode hétérogènes. On peut demander à connaître la classe d'un objet `x` au moyen de l'instruction `class(x)`. De même, on peut demander à connaître le mode d'un vecteur `x` par exemple au moyen de l'instruction `is.numeric(x)` ou `is.na(x)`.
NB: attention, il existe un mode `integer` (encodé sur 32 bits) ainsi qu'un mode `double` pour des raisons de compatibilité avec le langage C et le Fortran. L'instruction `is.integer(x)` ne teste PAS si les composantes du vecteur `x` sont entières!

- **Opérations élémentaires**:
  - opérations élémentaires sur les scalaires: `+` (addition), `-` (soustraction),`*` (multiplication), `/` (division), `%/%` (division entière), `^` (puissance), `%%` (modulo)
  - opérations avec affectation: l'affectation est effectuée (préférentiellement) par `<-` ou de manière (presque) équivalente par `=`. Les chaînes de caractères sont rentrées entre guillemets droits doubles (`" "`) ou simples (`' '`).

```
> x<- 45*8
> y= 25-9/0.4
> z<-x+y
> nom<- "Toto"
> print(nom)    # pour demander l'affichage
[1] "Toto"
```

  - Evaluation d'expression logique:
    Les opérations logiques `<` (infériorité stricte), `>` (supériorité stricte), `<=` (infériorité large), `>=` (supériorité large), `!=` (différence), `==` (égalité) retournent `TRUE` ou `FALSE`. Ils peuvent être utilisés pour des comparaisons de scalaires, ou bien de vecteurs et de matrices auquel cas, l'opération logique est effectuée terme à terme.

```
> x<- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> y<-sample(1:10,10,replace=T)
> y
 [1]  5  4  4  4  2 10 10  8 10  9
> x<4
 [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> x<y
 [1]  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE
```

    Il est possible de définir plusieurs conditions à remplir au moyen des opérateurs `&` ou `&&` (ET), `!` (négation), `|` ou `||` (OU), `xor` (OU exclusif).

```
> x<-4.5
> x<10
[1] TRUE
> (x<10)&(x>5)
[1] FALSE
> (x<10)|(x>5)
[1] TRUE
```

A noter: `&&` s'arrête dès que l'un des termes de la comparaison est `FALSE` et `||` s'arrête dès que l'un des termes de la comparaison est `TRUE`. Les opérateurs logiques `&` et `|` s'emploie pour des comparaisons de vecteurs ou de matrices et procèdent élément par élément.

```
> x<- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> (x>4)&(x<8)
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
> (x>4)&&(x<8)
[1] FALSE
> (x>4)||(x<8)
[1] TRUE
> (x>4)|(x<8)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

• **Opérations ensemblistes**:

`union(A,B)`	union de deux ensembles `A` et `B` du même type
`intersect(A,B)`	intersection de deux ensembles `A` et `B` du même type
`setdiff(A,B)`	différence de deux ensembles `A` et `B` du même type
`is.element(x,A)`	évalue l'appartenance de `x` à `A`
`setequal(A,B)`	évalue l'égalité de `A` et `B`

• **Créer et manipuler des vecteurs**:

- Rentrer des valeurs à la main peut se faire au moyen de la fonction `c()` qui concatène des valeurs isolées ou des vecteurs.

```
> y1 <- c(1,3.2,56,0.11,-9,8.3,12.7)   # vecteur numérique de longueur 7
> x<-c(y1,2,3,45.02)                   # vecteur numérique de longueur 10
> y2 <- c("petit","moyen","grand")     # vecteur de chaînes de caractères
> y3 <- c(F,F,F,F,T,F,T,T,F,T)         # vecteur logique
```

- Les commandes `:`, `seq`, `rep`, `rev` permettent d'automatiser le travail

```
> y1<- 1:10                  # suite arithmétique de raison 1
> y0<- 1.4: 10               # suite arithmétique de raison 1 (qui s'arrête à 9.4)
> y2<-10:1                   # suite arithmétique de raison -1
> z1<- seq(1,10)             # z1=y1
> z2<-rev(y1)                # z2=y2, la fonction rev renverse l'ordre des
```

```
>                               # composantes de y1
> x1<-seq(1,10,by=0.5)          # suite arithmétique de raison 0.5
> x2<-seq(1,10,length.out=100) # le nombre de pas entre 1 et 10 est de 100
> x3<-rep(1,times=10)           # 1 est répété 10 fois
> x4<-rep(y1,times=10)          # le vecteur y1 est répété 10 fois
> x5<-rep(c(1,2),each=10)       # chacune des valeurs est répétée 10 fois
```

- Manipuler un vecteur

```
> x<- sample(1.4:20.4,size=10,replace=T) # création du vecteur x par tirage
>                                        # uniforme avec remise dans {1.4,..,20.4}
> x
 [1] 18.4 13.4 20.4 10.4 17.4  7.4  4.4  7.4 12.4  9.4
> length(x)             # fournit la taille du vecteur x
[1] 10
> x[1]                  # extraction de la première composante de x
[1] 18.4
> x[c(2,4,8)]           # extraction des composantes \no 2, 4 et 8 de x
[1] 13.4 10.4  7.4
> x[-2]                 # tous les éléments de x sauf le 2ème
[1] 18.4 20.4 10.4 17.4  7.4  4.4  7.4 12.4  9.4
> x[-(3:6)]             # tous les éléments de x sauf ceux d'indice 3 à 6
[1] 18.4 13.4  4.4  7.4 12.4  9.4
> x[x>8]                # extraction des composantes de x qui sont >8
[1] 18.4 13.4 20.4 10.4 17.4 12.4  9.4
> x[(x>4)&(x<10)]       # extraction des composantes de x qui sont >4 et <10
[1] 7.4 4.4 7.4 9.4
> x[(x>14)|(x<8)]       # extraction des composantes de x qui sont >14 ou <8
[1] 18.4 20.4 17.4  7.4  4.4  7.4
> unique(x)             # extraction des composantes deux à deux différentes de x
[1] 18.4 13.4 20.4 10.4 17.4  7.4  4.4 12.4  9.4
> sort(x)               # composantes de x affichées dans l'ordre croissant
 [1]  4.4  7.4  7.4  9.4 10.4 12.4 13.4 17.4 18.4 20.4
> which(x==10.4)        # retourne l'indice des coordonnées égales à 10.4
[1] 4
> which(x>15)           # retourne l'indice des coordonnées >15
[1] 1 3 5
> which.min(x)          # retourne l'indice des coordonnées du/des minimum(s) de x
[1] 7
> which.max(x)          # retourne l'indice des coordonnées du/des maximum(s) de x
[1] 3
> rank(x)               # retourne le rang des éléments de x
 [1]  9.0  7.0 10.0  5.0  8.0  2.5  1.0  2.5  6.0  4.0
> diff(x)               # calcule le vecteur de longueur (length(x)-1) des
                        # différences entre chaque élément de x et le précédent
[1]  -5   7 -10   7 -10  -3   3   5  -3
```

- Opérations sur des vecteurs:

Les opérations `+,-,*,/` entre deux vecteurs de même taille sont des opérations terme à terme. De même, toute fonction appliquée sur un vecteur est évaluée terme à terme.

```
> x<- 1:5
> x
[1] 1 2 3 4 5
> y<- c(rep(0,3),rep(1,2))
> y
[1] 0 0 0 1 1
> x+y
[1] 1 2 3 5 6
> x*y
[1] 0 0 0 4 5
> z<-(1:10)^2
> z
 [1]   1   4   9  16  25  36  49  64  81 100
> ifelse(z>15,1,0) # créé un vecteur d'indicatrices qui vaut 1 si z>15 et 0 sinon
 [1] 0 0 0 1 1 1 1 1 1 1
```

L'instruction `crossprod(x,y)` calcule le produit scalaire de deux vecteurs x et y de même longueur à savoir `t(x)%*%y`.
L'instruction `outer(x,y,"*")` calcule le produit externe de deux vecteurs x et y à savoir `x%*%t(y)`.

- **Créer et manipuler des matrices**:
  - La fonction `matrix` permet de créer des matrices au nombre de colonnes fixés par `ncol` et au nombre de lignes fixés par `nrow`. Par défaut, une matrices est remplie colonne par colonne. Pour la remplir ligne par ligne, on rajoute l'option `byrow=T`. Attention, si la dimension du vecteur contenant les valeurs des composantes de la matrice n'est pas égale au produit `nrow`×`ncol`, alors il y a recyclage des valeurs et affichage d'un message d'avertissement.

```
> x<- matrix(0,nrow=2,ncol=3)
> x
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
> x<- matrix(NA,nrow=2,ncol=3)
> x
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> x<- matrix(1:10,nrow=2)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> x<- matrix(1:10,ncol=5)
```

```
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> x<- matrix(1:10,nrow=2,byrow=T)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
> x<- matrix(1:10,nrow=3)
Message d'avis :
In matrix(1:10, nrow = 3) :
  la longueur des données [10] n'est pas un diviseur ni un multiple du nombre
de lignes [3]
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8    1
[3,]    3    6    9    2
> x<- matrix(1:10,nrow=3,ncol=5)
Message d'avis :
In matrix(1:10, nrow = 3, ncol = 5) :
  la longueur des données [10] n'est pas un diviseur ni un multiple du nombre
de lignes [3]
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10    3
[2,]    2    5    8    1    4
[3,]    3    6    9    2    5
```

- Une matrice peut être créée par concaténation de vecteurs de même taille.

```
> x <- 1:10
> y <- x^2
> rbind(x,y)
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
x    1    2    3    4    5    6    7    8    9    10
y    1    4    9   16   25   36   49   64   81   100
> cbind(x,y)
       x   y
 [1,]  1   1
 [2,]  2   4
 [3,]  3   9
 [4,]  4  16
 [5,]  5  25
 [6,]  6  36
 [7,]  7  49
 [8,]  8  64
```

```
 [9,]  9  81
[10,] 10 100
```

- Extraire des éléments d'une matrice

```
> y<- sample(1:22,size=15,replace=T)
> x<-matrix(y,nrow=3)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    8   21   21    1   10
[2,]    6    9    3   18   18
[3,]   11    7   10    3   21
> x[2,3]      # extraire l'élément situé en 2ème ligne, 3ème colonne
[1] 3
> x[,1]
[1]  8  6 11  # extraire la 1ère colonne
> x[3,]       # extraire la 3ème ligne
[1] 11  7 10  3 21
> x[1:2,2:4]
> # extraire la sous-matrice composée des lignes 1 et 2 sur les colonnes 2 à 4
     [,1] [,2] [,3]
[1,]   21   21    1
[2,]    9    3   18
> x[-2,]        # x sans sa seconde ligne
     [,1] [,2] [,3] [,4] [,5]
[1,]    8   21   21    1   10
[2,]   11    7   10    3   21
> x[,-c(2,4)]   # x sans ses colonnes 2 et 4
     [,1] [,2] [,3]
[1,]    8   21   10
[2,]    6    3   18
[3,]   11   10   21
```

- Extraire de la matrice M les lignes telles que les éléments sur la première colonne sont supérieurs à 0.1

```
> M<-matrix(rnorm(20),nrow=4,ncol=5)
> M
            [,1]       [,2]        [,3]       [,4]      [,5]
[1,]  0.59186891  0.8630190  0.06212799  0.4355270 1.4282895
[2,] -0.28584743  0.7802716  3.56221003  0.9709235 0.9000507
[3,] -0.01265327 -2.0097372 -1.22889626 -0.9055665 1.2105219
[4,]  0.51772740 -0.9042950  0.33737161 -0.2337437 1.7191870
> M[,1]>0.1
[1]  TRUE FALSE FALSE  TRUE
> M[M[,1]>0.1,]
          [,1]      [,2]       [,3]       [,4]     [,5]
[1,] 0.5918689  0.863019 0.06212799  0.4355270 1.428289
[2,] 0.5177274 -0.904295 0.33737161 -0.2337437 1.719187
```

- Concaténer des matrices:
  Les fonctions `rbind` et `cbind` permettent de concaténer des matrices pourvu que leurs dimensions soient compatibles.

```
> x<- matrix(1:6,nrow=2,ncol=3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> y<- matrix(7:12,nrow=2,ncol=3)
> y
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
> rbind(x,y)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[3,]    7    9   11
[4,]    8   10   12
> cbind(x,y)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
```

- Opérations sur des matrices:
  Les opérations `+,-,*,/` entre deux matrices de même taille sont des opérations terme à terme. De même, toute fonction numérique appliquée sur une matrice est évaluée terme à terme.

```
> x<- matrix(1:6,nrow=2,ncol=3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> dim(x)
[1] 2 3
> y<- matrix(7:12,nrow=2,ncol=3)
> y
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
> x+y
     [,1] [,2] [,3]
[1,]    8   12   16
[2,]   10   14   18
> x*y
     [,1] [,2] [,3]
```

```
[1,]    7   27   55
[2,]   16   40   72
> x\%*\%t(y)
     [,1] [,2]
[1,]   89   98
[2,]  116  128
> det(x\%*\%t(y))
[1] 24
> solve(x\%*\%t(y))
```

`dim(A)`	donne la dimension d'une matrice `A`
`t(A)`	calcule la transposée
`Conj(A)`	calcule la conjuguée
`det(A)`	calcule le déterminant
`diag(A)`	extrait la diagonale d'une matrice A
`diag(x)`	créée la matrice diagonale dont la diagonale est le vecteur `x`
`diag(4)`	créée la matrice identité de dimension 4×4
`bdiag(A)`	créée une matrice diagonale par blocs
`solve(A)`	détermine l'inverse d'une matrice carrée inversible
`solve(A,B)`	calcule $A^{-1}.B$
`A%*%B`	calcule le produit matriciel `A.B`
`eigen(A)`	détermine les valeurs et les vecteurs propres d'une matrice `A` carrée et diagonalisable. Le résultat de cette fonction est une liste à deux composantes nommées `values` et `vectors`. Ainsi, la commande `eigen(S)$val` fournit le vecteur des valeurs propres et la commande `eigen(S)$vec` fournit la matrice des vecteurs propres correspondants.
`lower.tri(A)`	extrait la partie triangulaire inférieure de `A`
`upper.tri(A)`	extrait la partie triangulaire supérieure de `A`
`vec(A)`	concatène les colonnes de `A` en un grand vecteur
`trace(A)`	calcule la trace de `A`
`kappa(A)`	calcule le conditionnement de `A`
`kronecker(A,B)`	calcule le produit de Kronecker $A \otimes B$
`svd(A)`	calcule la décomposition en valeurs singulières de A matrice $m \times n$ en $A = U.D.V^t$ où D est la matrice diagonale $m \times n$ des valeurs singulières, U et V sont des matrices unitaires de taille respective $m \times m$ et $n \times n$.
`chol(A)`	calcule la décomposition de Cholesky d'une matrice réelle symétrique et définie positive `A` en $A = U^t.U = L.L^t$ NB: `chol2inv` calcule A^{-1} en utilisant la décomposition de Cholesky
`qr(A)`	calcule la décomposition QR de A en $A = QR$ où Q est une matrice orthogonale et R est une matrice triangulaire supérieure
`backsolve(R,b)`	résout $Rx = b$ où R est une matrice triangulaire supérieure
`forwardsolve(L,b)`	résout $Lx = b$ où L est une matrice triangulaire inférieure

• **La fonction** `apply`:
La fonction `apply` permet d'appliquer une fonction sur les lignes ou les colonnes d'une matrice.

```
> x<-matrix(seq(1:12),ncol=4)
```

```
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> apply(x,1,mean)   # moyenne sur les lignes
[1] 5.5 6.5 7.5
> apply(x,2,mean)   # moyennne sur les colonnes
[1]  2  5  8 11
```

• **La fonction** `sweep`:
L'instruction `sweep(A,1,x,FUN='/')` permet de diviser (multiplier, soustraire,...) chacun des lignes (code 1) ou des colonnes (code 2) de la matrice `A` par un élément du vecteur `x` (par exemple le vecteur des écarts-types par colonne pour réduire une matrice).

• **Généralisation des matrices à plus de 2 dimensions**:
La fonction `array` permet de créer de tels tableaux. La fonction `apply` reste utilisable.

```
> A=array(1:24,c(2,4,3))
> A
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]    9   11   13   15
[2,]   10   12   14   16

, , 3

     [,1] [,2] [,3] [,4]
[1,]   17   19   21   23
[2,]   18   20   22   24

> A[,,1]
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> A[,1,1]
[1] 1 2
> A[1,,1]
[1] 1 3 5 7
> apply(A,1,max)
[1] 23 24
```

```
> apply(A,2,max)
[1] 18 20 22 24
> apply(A,3,max)
[1]  8 16 24
```

La fonction `aperm` généralise la transposition des matrices en permutant les dimensions, et si nécessaire, en modifiant adéquatement la taille. Ainsi `A[i,j,k]` devient `A[k,j,i]`.

• Créer et manipuler une liste:
Une liste est une structure qui contient des objets pas nécessairement de même type, ni de même taille.

```
> mylist<- list(serie=sample(1:200,size=100,replace=T),taille=100,
+ type="random sampling with replacement",extra="coucou")
> names(mylist)        # affiche le nom des différentes entrées de la liste
[1] "serie"  "taille" "type"   "extra"
> summary(mylist)
# résume quelques informations sur les différentes entrées de la liste
       Length Class  Mode
serie  100    -none- numeric
taille   1    -none- numeric
type     1    -none- character
extra    1    -none- character
> mylist$taille       # affiche l'entrée appelée taille de la liste
[1] 100
> mylist[[2]]          # affiche la 2ème entrée de la liste
[1] 100
```

La fonction `lapply` permet d'appliquer une fonction sur les éléments d'une liste.
La fonction `sapply` permet d'appliquer une fonction sur les éléments d'une liste et simplifie la sortie.

• La fonction `replicate`:
L'instruction `replicate(n,expr)` répète une opération faisant intervenir de l'aléa et renvoie un tableau multidimensionnel résultant de cette suite d'opérations.

```
> A<-replicate(100,rnorm(30))
> class(A)
[1] "matrix"
> dim(A)
[1]  30 100
```

• Créer un tableau croisé
La fonction `outer` appliquée sur un couple de vecteurs `(x,y)` et une fonction `f` retourne une matrice M de taille $n_x \times n_y$ où n_x et n_y sont respectivement la taille de `x` et de `y` et dont la composante située en ligne `i` et colonne `j` vaut $M(i,j) = f(x_i, y_j)$.

```
> f=function(x,y) log(x^2+y^4)
> x=seq(1:10,by=0.5)
> y=seq(-10:10,by=0.5)
> M=outer(x,y,'f')
```

• **Tables de contingence et de proportions**:
L'instruction `table` compte les différents niveaux d'un facteur apparaissant dans les données.

```
> x<- rbinom(50,30,0.4)
> x
 [1] 11 12 13 10 13 11 13 12 15 17 14 12 10 14 10 14 12 10 16 13 12 15  9 17 13 16 10
[28] 15 16 15 16 14 10 14 12 14 11 14 14 13 14 10 13  7 10 12 11 15 15 18
> table(x)
x
 7  9 10 11 12 13 14 15 16 17 18
 1  1  8  4  7  7  9  6  4  2  1
> y<- c(rep("yes",10),rep("no",30),rep("yes",10))
> y
 [1] "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "no"  "no"  "no"  "no"
[15] "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"
[29] "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "yes" "yes"
[43] "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes"
> table(x,y)
    y
x    no yes
  7   0   1
  9   1   0
  10  5   3
  11  1   3
  12  4   3
  13  3   4
  14  8   1
  15  3   3
  16  4   0
  17  1   1
  18  0   1
> z<-sample(1:2,50,replace=T)
> z
 [1] 1 1 1 1 1 2 1 2 1 1 2 2 1 1 1 1 2 1 1 2 2 2 1 2 1 1 1 2 2 1 1 2 1 1 2 1 1 2 2 2 2 2 1
> table(x,y,z)
, , z = 1

    y
x    no yes
  7   0   1
  9   1   0
  10  5   2
  11  1   1
  12  0   1
  13  1   4
  14  4   0
  15  1   3
  16  3   0
```

```
  17  0   1
  18  0   0

, , z = 2

    y
x    no yes
  7   0   0
  9   0   0
  10  0   1
  11  0   2
  12  4   2
  13  2   0
  14  4   1
  15  2   0
  16  1   0
  17  1   0
  18  0   1
```

L'instruction `ftable` retourne une table de contingences sous la forme d'une table dit plate (en anglais *flat*, d'où le f dans le nom de la fonction) dans le cas d'un croisement de 3 variables ou plus, plutôt que sous la forme d'un array comme le fait l'instruction `table`.

```
> ftable(x,y,z)
       z 1 2
x  y
7  no    0 0
   yes   1 0
9  no    1 0
   yes   0 0
10 no    5 0
   yes   2 1
11 no    1 0
   yes   1 2
12 no    0 4
   yes   1 2
13 no    1 2
   yes   4 0
14 no    4 4
   yes   0 1
15 no    1 2
   yes   3 0
16 no    3 1
   yes   0 0
17 no    0 1
   yes   1 0
18 no    0 0
   yes   0 1
```

L'instruction `xtabs` fournit des tables de contingence, tout comme les instructions `table` et `ftable`. Attention à la différence de synthaxe : `xtabs` prend en entrée une formule!

```
> xtabs(~x+y+z)
, , z = 1

    y
x    no yes
  7   0   1
  9   1   0
  10  5   2
  11  1   1
  12  0   1
  13  1   4
  14  4   0
  15  1   3
  16  3   0
  17  0   1
  18  0   0

, , z = 2

    y
x    no yes
  7   0   0
  9   0   0
  10  0   1
  11  0   2
  12  4   2
  13  2   0
  14  4   1
  15  2   0
  16  1   0
  17  1   0
  18  0   1
```

L'instruction `prop.table` convertit les comptages d'une table de contigences en proportions relatives à la table entière ou à ses marginales (lignes ou colonnes).

```
> prop.table(table(x))
x
   7    9   10   11   12   13   14   15   16   17   18
0.02 0.02 0.16 0.08 0.14 0.14 0.18 0.12 0.08 0.04 0.02
> prop.table(table(x,y))
    y
x      no  yes
  7  0.00 0.02
  9  0.02 0.00
  10 0.10 0.06
```

```
  11 0.02 0.06
  12 0.08 0.06
  13 0.06 0.08
  14 0.16 0.02
  15 0.06 0.06
  16 0.08 0.00
  17 0.02 0.02
  18 0.00 0.02
> prop.table(table(x,y),1)
    y
x           no       yes
  7  0.0000000 1.0000000
  9  1.0000000 0.0000000
  10 0.6250000 0.3750000
  11 0.2500000 0.7500000
  12 0.5714286 0.4285714
  13 0.4285714 0.5714286
  14 0.8888889 0.1111111
  15 0.5000000 0.5000000
  16 1.0000000 0.0000000
  17 0.5000000 0.5000000
  18 0.0000000 1.0000000
> prop.table(table(x,y),2)
    y
x            no        yes
  7  0.00000000 0.05000000
  9  0.03333333 0.00000000
  10 0.16666667 0.15000000
  11 0.03333333 0.15000000
  12 0.13333333 0.15000000
  13 0.10000000 0.20000000
  14 0.26666667 0.05000000
  15 0.10000000 0.15000000
  16 0.13333333 0.00000000
  17 0.03333333 0.05000000
  18 0.00000000 0.05000000
```

Notons que l'instruction `expand.grid` peut servir à énumérer toutes les combinaisons des niveaux des différents facteurs impliqués dans une étude.

```
> expand.grid(categorie=c("A","B","C"), reponse=c("yes","no"), genre=c("M","F"))
   categorie reponse genre
1          A     yes     M
2          B     yes     M
3          C     yes     M
4          A      no     M
5          B      no     M
6          C      no     M
7          A     yes     F
```

```
8          B     yes     F
9          C     yes     F
10         A      no     F
11         B      no     F
12         C      no     F
```

• Structures de contrôle et itérations:
La syntaxe `if (condition) instructions` permet d'effectuer les instructions uniquement dans le cas où la condition est remplie. La syntaxe `if (condition) instructions A else instructions B` permet d'effectuer les instructions A dans le cas où la condition est remplie et les instructions B sinon. Afin d'effectuer des boucles, on peut utiliser les instructions `for` ou `while`. Noter que dans une boucle effectuée avec l'instruction `for`, le compteur est incrémenté automatiquement, ce qui n'est pas le cas si l'on utilise l'instruction `while`. Sur l'exemple suivant, les deux boucles produisent le même résultat.

```
> for (i in 1:10)
+  {
+    print(i)
+    }

> i=1
> while(i<=10)
+ {
+  print(i)
+  i<- i+1
+  }
```

• Fonctions mathématiques

`min`	calcule le minimum
`cummin`	fonction minimum cumulé: le résultat est un vecteur de même taille que celui d'entrée
`pmin(x,y)`	fonction minimum entre deux vecteurs de même longueur: le résultat est un vecteur de même taille que ceux d'entrée dont le $i^{\text{ème}}$ élément est $\min(x[i], y[i])$
`max`	calcule le maximum
`cummax`	fonction maximum cumulé: le résultat est un vecteur de même taille que celui d'entrée
`pmax(x,y)`	fonction maximum entre deux vecteurs de même longueur: le résultat est un vecteur de même taille que ceux d'entrée dont le $i^{\text{ème}}$ élément est $\max(x[i], y[i])$
`sum`	fonction somme
`cumsum`	fonction somme cumulée: le résultat est un vecteur de même taille que celui d'entrée
`prod`	fonction produit
`cumprod`	fonction produit cumulé: le résultat est un vecteur de même taille que celui d'entrée
`abs`	fonction valeur absolue d'un réel ou module d'un nombre complexe
`sign`	fonction signe (ne fonctionne pas pour un nombre complexe)
`Arg`	fonction argument (angle) d'un nombre complexe
`Conj`	fonction nombre conjugué d'un nombre complexe

`sqrt`	fonction racine carrée
`exp`	fonction exponentielle
`log`	fonction logarithme naturel
`log10`	fonction logarithme en base 10
`log2`	fonction logarithme en base 2
`sin`	fonction sinus
`cos`	fonction cosinus
`tan`	fonction tangente
`asin`	fonction arcsinus
`acos`	fonction arccosinus
`atan`	fonction arctangente
`sinh`	fonction sinus hyperbolique
`cosh`	fonction cosinus hyperbolique
`tanh`	fonction tangente hyperbolique
`floor`	fonction partie entière inférieure
`ceiling`	fonction partie entière supérieure
`round`	fonction arrondi
`trunc`	fonction troncature
`x%%y`	reste de la division de `x` par `y`
`factorial`	calcule la factorielle
`choose`	calcule le coefficient binômial
`beta`	fonction beta
`lbeta`	logarithme de la fonction beta
`gamma`	fonction gamma
`lgamma`	logarithme de la fonction gamma
`digamma`	dérivée première de la fonction gamma
`trigamma`	dérivée seconde de la fonction gamma
`psigamma`	dérivée d'ordre quelconque (>0) de la fonction gamma

• **Evaluations numériques**

`integrate`	intègre numériquement
`grad`	package `numDeriv`, évalue numériquement le vecteur gradient en un point fixé
`hessian`	package `numDeriv`, évalue numériquement la matrice hessienne en un point fixé
`optimize`	détermine numériquement le minimum (ou le maximum) et son argument d'une fonction unidimensionnelle dans un intervalle fixé, `optimise` est un alias
`optim`	détermine numériquement le minimum (ou le maximum) et son argument d'une fonction multidimensionnelle, éventuellement dans un intervalle fixé
`nlm`	détermine numériquement le minimum (ou le maximum) et son argument d'une fonction multidimensionnelle, éventuellement dans un intervalle fixé
`nlminb`	détermine numériquement le minimum (ou le maximum) et son argument d'une fonction multidimensionnelle dans un cube
`ConstrOptim`	détermine numériquement le minimum (ou le maximum) et son argument d'une fonction multidimensionnelle sous des contraintes linéaires

`maxLik`	package `maxLik`, détermine le maximum de vraisemblance
`optimx`	package `optimx`, tentative d'unification des différentes fonctions `R` d'optimisation
`uniroot`	détermine numériquement les zéros d'une fonction unidimensionnelle dans un intervalle fixé
`polyroot`	détermine numériquement tous les zéros (éventuellement complexes) d'un polynôme
`fft`	détermine la transformée de Fourier rapide
`fft( ,inverse=T)`	détermine l'inverse de la transformée de Fourier

• **Lois de probabilités**

loi	nom	paramètres
beta	beta	shape1, shape2
binomiale	binom	size, prob
binomiale négative	nbinom	size, prob
Cauchy	cauchy	location, scale
χ^2	chisq	df, ncp
exponentielle	exp	rate
Fisher	f	df1, df2, ncp
gamma	gamma	shape, scale
géométrique	geom	prob
hypergéométrique	hyper	m, n, k
log-normale	lnorm	meanlog, sdlog
logistique	logis	location=0,scale=1
normale	norm	mean=0, sd=1
Poisson	pois	lambda
Student	t	df, ncp
uniforme	unif	min=0, max=1
Weibull	weibull	shape, scale

Simuler un échantillon i.i.d. de taille n peut se faire avec l'instruction `rloi(n,paramètres)` en remplaçant `loi` par l'un des noms dans le tableau et `paramètres` par les paramètres associés. Déterminer le quantile théorique d'ordre `p` est possible avec l'instruction `qloi(p,paramètres)` en remplaçant `loi` par l'un des noms dans le tableau et `paramètres` par les paramètres associés. Déterminer la valeur de la densité théorique en un point x peut se faire avec l'instruction `dloi(x,paramètres)` en remplaçant `loi` par l'un des noms dans le tableau et `paramètres` par les paramètres associés. Déterminer la valeur de la fonction de répartition théorique en un point x peut se faire avec l'instruction `ploi(x,paramètres)` en remplaçant `loi` par l'un des noms dans le tableau et `paramètres` par les paramètres associés.

• **Fonctions statistiques**:
Certaines de ces fonctions effectuent les calculs demandés et tracent des graphiques.

`range`	détermine l'étendue d'un vecteur de valeurs
`mean`	calcule la moyenne empirique
`median`	calcule la médiane empirique
`var`	calcule la variance empirique
`sd`	calcule l'écart-type empirique
`skewness`	calcule le coefficient d'asymétrie empirique package moments
`kurtosis`	calcule le coefficient d'aplatissement empirique package moments
`summary`	produit quelques statistiques descriptives
`cov`	calcule la covariance empirique entre deux vecteurs
`cor`	calcule la corrélation empirique entre deux vecteurs
`quantile(x,probs)`	détermine le quantile empirique d'ordre `probs` du vecteur x
`boxplot`	trace la boîte à moustaches
`ecdf`	détermine la fonction de répartition empirique
`hist(x,probability=T,nclass=NULL)`	trace l'histogramme de x (avec `nclass` nombre de classes)
`density(x)`	détermine à partir de x l'estimateur à noyau de la densité
`qqnorm(x)`	trace les quantiles empiriques de x en fonction des valeurs attendues sous l'hypothèse gaussienne

En anticipant un peu sur la section "Fonctions graphiques", voici quelques exemples à faire tourner et à méditer...

Exemple 1: un histogramme avec le tracé de la densité de la loi $\mathcal{N}(0,1)$:

```
> x <- rnorm(150)
> hist(x, breaks=20, freq=F, col="cyan")
> curve(dnorm(x), add=T,col="darkblue")
> curve(dnorm(x,mean=0,sd=0.8), add=T,col="darkred")
>
> hist(x, breaks=20, freq=F)$breaks
 [1] -2.8 -2.6 -2.4 -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.2  0.0  0.2
[17]  0.4  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.6
> length(hist(x, breaks=20, freq=F)$breaks)
[1] 28
> hist(x, breaks=20, freq=F)$counts
 [1]  1  0  2  3  4  4  1 11  5  4 10  7 18 13 13 10  2 12  6  7  4  5  2  1  2  2  1
```

Exemple 2: variations autour d'un boxplot:

```
x <- rnorm(100)
par(mfcol=c(2,2))
boxplot(x)
boxplot(x,horizontal=T)
boxplot(x, col="red")
boxplot(x, col="yellow",border="darkblue",lwd=2)
```

Exemple 3: plusieurs boxplot en parallèle:

```
x <- rnorm(100)
```

```
y <- rnorm(200,mean=1,sd=2)
z <- rnorm(50, mean=10, sd=0.2)
boxplot(x,y,z,col=c("blue","white","red"),border=c("black","darkblue"),lwd=1.5)
```

• **Fonctions graphiques**:

`par(mfrow=c(i,j))`	permet de couper une fenêtre graphique en i lignes et j colonnes
`plot`	trace un nouveau graphique
`lines`	ajoute une courbe sur un graphique déjà existant
`points`	ajoute des points sur un graphique déjà existant
`title`	ajoute un titre sur un graphique déjà existant
`legend`	ajoute une légende sur un graphique déjà existant
`abline`	permet d'ajouter des droites verticales, horizontales ou obliques sur un graphique déjà existant
`par(new=TRUE)`	trace un graphique sur un graphique déjà existant
`contour(x,y,z)`	place les données de la matrice `z` en fonction des valeurs stockées dans les vecteurs `x` et `y` puis trace des lignes de niveau. NB: Il faut donc que `dim(z)=c(length(x),length(y))`
`filled.contour`	idem mais les aires entre les contours sont colorées
`image(x,y,z)`	trace une grille de rectangles dont la couleur dépend de la valeur de `z`
`persp(x,y,z)`	trace et interpole en 3D les données de `z` en fonction des valeurs stockées dans les vecteurs `x` et `y`.
`pairs(X)`	si X est une matrice, produit un graphique pour chaque paire de colonnes de X
`coplot(y x|z)`	si `x` et `y` sont des vecteurs numériques et si `z` est un vecteur catégoriel (resp numérique) produit un graphique de `y` en fonction de `x` pour chaque modalité (resp pour chaque intervalle de valeurs) de `z`
`coplot(y x|z1+z2)`	produit un graphique pour chaque couple de valeurs (intervalles) de `(z1,z2)`

Exemple d'usage:

```
> aux<-function(x,y){cos(x*y)}
> image(1:10,1:10,outer(1:10,1:10,aux))
> contour(1:10,1:10,outer(1:10,1:10,aux))
> persp(1:10,1:10,outer(1:10,1:10,aux))
```

La fonction `plot` peut prendre de nombreux arguments dont les suivants:

- `type`: type de graphique ('p'=points, 'l'=ligne, 'h'= diagramme en bâtons, 's' et 'S'=escalier)
- `pch`: type de points (de 0 à 25)
- `lty`: type de traits (de 1 à 6)
- `lwd`: largeur des traits
- `col`: couleur de la courbe
- `cex`: character expansion = taille des caractères
- `ylim=c(0,10)` et `xlim=c(-5,5)` pour fixer les limites des axes du graphique

• **Ecrire ses propres fonctions**:
La structure générale d'une fonction est la suivante:

```
myfunction <- function(arg1,arg2,arg3=mydefault)
 {  instructions à effectuer
    résultats à renvoyer placés en dernière ligne
   }
```

Les accolades définissent le début et la fin de la fonction. La dernière ligne doit comporter **uniquement** l'appel de l'objet renvoyé par la fonction. Les arguments d'entrée sont ici `arg1`, `arg2` et `arg3`, ce dernier ayant une valeur par défaut égale à `mydefault`. Les deux exemples ci-dessous illustrent la définition d'une fonction.

```
foo1<-function(x)
 { x^2+x*sin(2*pi*x) }

foo2<-function(x,k=2)
 { x^k+x/k*sin(2*pi*x) }
```

• **Importer et exporter des données**:
La passerelle liant `R` à un autre logiciel scientifique (ou tableur) est le format texte (ASCII). Ainsi, `R` peut importer et exporter du format texte.
La fonction `read.table` permet de lire des données stockées au format `txt` ou `csv` dans un fichier externe.
L'instruction `read.table("myfile.txt")` lit les données contenues dans le fichier `myfile.txt`.
L'instruction `read.table("myfile.txt",header=TRUE)` lit les données contenues dans le fichier `myfile.txt` mais cette fois-ci, la première ligne contient les noms de variables (qui ne doivent pas être confondues avec des données).
L'instruction `read.table("myfile.csv",sep="",dec=",")` lit les données contenues dans le fichier `myfile.csv`, le séparateur est l'espace, les chiffres décimaux figurent derrière le caractère ','.

A l'inverse, la fonction `write.table` permet d'écrire des données stockées au format txt ou csv dans un fichier externe.
La commande `sink` permet la redirection du résultat des commandes (non-graphiques) vers un fichier (auquel cas il n'y a pas d'affichage à l'écran). Ne pas oublier de fermer le fichier en rappelant `sink()` sans argument.

```
A=rnorm(50)
write.table(A,"myA.txt")
sink("my-analysis-A.txt")
A
summary(A)
sink()
```

La fonction `png` permet d'enregistrer des graphiques au format `png`. Ne pas oublier de fermer le fichier en appelant `dev.off()` sans argument.

```
> png(filename="my-image.png",width=10,height=21,units="cm",res=512)
> boxplot(A)
> dev.off()
```

De même, la fonction `postscript` permet d'enregistrer des graphiques au format `ps`. Ne pas oublier de fermer le fichier en appelant `dev.off()` sans argument.

```
postscript(file="my-plot.eps")
x<-seq(-10,10,0.5)
y<-x^2
plot(x,y,type="l",main="f:x->x^2",col="red")
dev.off()
```

Des packages spécifiques sont disponibles et parfois nécessaires pour permettre et simplifier la communication avec d'autres logiciels comme Minitab, S, SAS, SPSS, Stata, Octave, Python... Le package `xlsReadWrite` contient les fonctions `read.xls` et `write.xls` qui permettent respectivement de lire et d'écrire des données au format `xls`.