SY09 Printemps 2016
TP 0 — Introduction à R

1 Prise en main de R

1.1 Installation

Les fichiers relatifs à R (fichier exécutable pour l'installation, sources, packages additionnels) sont tous accessibles via le CRAN (*Comprehensive R Archive Network*). La mise à jour des packages peut se faire manuellement ou via le gestionnaire et l'installateur disponible dans le menu déroulant du logiciel.

1.2 Lancement et utilisation

Une fois lancé, le logiciel ouvre un certain nombre de fenêtres, dont la console qui est utilisée pour exécuter les commandes. Après le lancement, la console affiche un message d'accueil et une invite de commande :

```
>
```

Une aide est disponible dans le menu déroulant du logiciel, et peut également être lancée via la commande

```
> help()
```

Remarquons que l'usage des parenthèses est obligatoire pour exécuter une fonction, même si elle ne nécessite pas d'arguments (leur omission entraîne l'affichage du code de la fonction dans la console).

D'une manière générale, l'exécution de commandes peut se faire de deux manières :
- en les tapant directement dans la console, les unes après les autres (il est possible d'accéder aux commandes précédemment tapées via les flèches haut et bas du clavier) ;
- en les écrivant dans un script, et en exécutant le script sous R (voir paragraphe 1.3).

Exercice : manipulation de nombres

Taper les instructions suivantes :

```
> 8*5/(2*2)
> x<-4
> y<-5
> z<-x*y
> sin(pi*x)
> ls()
```

Exercice : manipulation de vecteurs et de matrices

Taper les instructions suivantes et observer le résultat ; à quoi sert l'argument `byrow` ?

```
> A<-matrix(1:9,nrow=3,byrow=T)
> B<-matrix(c(5,3,7,4,6,3,1,6,3,2,8,5),nrow=4,byrow=T)
> A
> B

> dim(A)
> na<-dim(A)[1]
> pa<-dim(A)[2]

> v<-1:3   # ou v<-c(1,2,3)
> v
```

Il est possible de concaténer des matrices en les agrégeant par lignes ou par colonnes :

```
> C<-cbind(A,v)
> D<-rbind(A,B)
> C
> D
```

On peut extraire des éléments, des lignes ou des colonnes d'une matrice :

```
> dim(C)
> dim(D)
> D[3,3]
> D[,2]
> D[5,]
> D[2:4,2:3]
```

Que fait la commande `diag` ?

```
> diag(3)
> diag(v)
> diag(A)
```

Tester les opérations matricielles suivantes :

```
> matrix(1,3,2)
> t(A)
> A[,2]<-v
> A
```

```
> B%*%A
> A%*%A
> A*A
> A+A
```

Quelle est la différence entre les opérations `%*%` et `*` ?

La fonction `array` (et la classe correspondante) permet de créer et manipuler des tableaux de nombres de dimension quelconque (potentiellement supérieure à 2) :

```
> E <- array(1,c(4,3,2))
> E[,,2] <- E[,,2]*2
> E
```

Notons que le « liage » de matrices selon la troisième dimension, de manière à former un tel tableau de nombres, nécessite l'installation et le chargement du package `abind` :

```
> library(abind)
> F1 <- matrix(1,4,3)
> F2 <- matrix(2,4,3)
> F <- abind(F1, F2, along=3)
> F
```

1.3 Scripts et fonctions

Scripts

Dans le cadre des TDs de SY09, les traitements à appliquer aux données peuvent être conséquents, et l'usage des mêmes commandes devra parfois être répété sur différents jeux de données. Il est donc *vivement conseillé* d'utiliser des scripts (qui permettent en outre de garder une trace du travail effectué d'une séance sur l'autre).

Un script R est un fichier d'extension « `.R` » (par exemple « `mon-script.R` »), dans lequel sont écrites une suite de commandes, et que l'on peut exécuter de la manière suivante :

```
> source("mon-script.R")
```

L'exécution d'un script renvoie un message d'erreur si le fichier correspondant ne se trouve pas dans le répertoire courant. Il est possible de passer le chemin complet en argument de la fonction `source` (ou de se placer dans le répertoire de travail adéquat).

Fonctions

Une fonction est une séquence d'instructions que l'on souhaite appeler via une commande spécifique. On stocke généralement ces fonctions dans un fichier, que l'on charge à l'aide de la fonction `source` (voir ci-dessus) pour qu'elles soient en mémoire, de manière à pouvoir les utiliser.

Par exemple, supposons que l'on souhaite créer deux fonctions, `somme` et `produit`, stockées dans le fichier « `mes-fonctions.r` » :

```
somme <- function(a, b)
{
    resultat <- a + b;
}

produit <- function(a, b)
{
    resultat <- a * b;
}
```

Le chargement du fichier puis l'exécution des fonctions peut se faire via les commandes

```
> source("mes-fonctions.r")
> res1 <- somme(2, 3)
> res2 <- produit(2, 3)
```

Notons qu'il est possible d'affecter une valeur par défaut à un argument dans l'en-tête de la fonction, qu'il prendra en l'absence de valeur renseignée par l'utilisateur. Par exemple : `produit <- function(a, b=2)`.

Par défaut, une fonction retourne le résultat de la dernière instruction qu'elle exécute. Pour retourner plusieurs arguments, on pourra créer un objet contenant différents champs dans lesquels on stockera les arguments, que l'on retournera enfin à l'utilisateur :

```
mafonction <- function(..., ...)
{
    ...
    objet <- NULL
    objet$champ1 <- ...
    objet$champ2 <- ...
    objet$champ3 <- ...
    objet
}
```

L'appel de la fonction permet ainsi de récupérer l'ensemble des arguments srtockés dans un même objet :

```
monobjet <- mafunction(..., ...)
objet$champ1
objet$champ2
objet$champ3
```

Répertoire de travail

Dans un certain nombre de situations (exécution de scripts, chargement de fonctions ou de données « personnelles » (non disponibles dans des packages), sauvegarde d'informations dans un fichier, etc, il est nécessaire de connaître le répertoire courant (dans lequel on travaille) voire de le changer. Cela peut être fait via le menu déroulant du logiciel, ou en utilisant les commandes

```
> getwd()
> setwd("/Users/myname/Documents/R/TP/TP0")
```

Exercice : programmation

Programmer une fonction nommée `prodtrans` qui calcule le produit $X^{\mathrm{t}}X$ d'une matrice X par sa transposée. On stockera la fonction dans un fichier « `prodtrans.R` ». On chargera la fonction et on l'exécutera sur la matrice `A`, puis sur la matrice `B`.

2 Analyse de données avec R

2.1 Opérations de base

Indicateurs numériques, graphiques de base

Taper les instructions suivantes et observer le résultat ; à quoi sert la fonction `apply` ?

```
> x<-c(2,4,3,7,1)
> A<-matrix(c(1,2,5,3,0,9),nrow=3,byrow=T)
> max(x)
> max(A)
> apply(A,1,max)
> apply(A,2,max)
> apply(E,3,sum)
> mean(x)
> mean(A)
> apply(A,2,mean)
```

Que calculent les fonctions `var` et `sd` ?

```
> var(x)
> var(A)
> sd(x)
> sd(A)
```

On comparera les résultats des fonctions `var` et `cov.wt` :

```
> var(A)
> cov.wt(A, method='unbiased')
> cov.wt(A, method='ML')
```

Les fonctions `hist` et `plot` permettent de faire des affichages simples :

```
> x<-c(2,2,2,1,3,4,1,1)
> hist(x)
> x<-c(1,2,3,4,5)
> y<-c(1,4,9,16,25)
> plot(x,y)
> plot(x,y,pch=22)  # taper ?points pour les informations sur les codes associés à pch
> plot(x,y,pch=19,col='blue')
> plot(x,y,type='l',col='blue')
```

Si l'on veut faire plusieurs affichages de graphiques, il est possible de créer de nouvelles fenêtres en utilisant la fonction `x11` (ou `quartz` sous OS X).

Exercice : centrage en colonne

Écrire une fonction nommée `centre` qui centre une matrice X en colonne. On rappelle qu'une matrice centrée en colonne est telle que les moyennes de ses colonnes sont toutes nulles. Lorsque la matrice X correspond à un tableau individus-variables, cette opération consiste à effectuer une translation du nuage de points associé de manière à faire coïncider son centre de gravité avec l'origine du repère.

Vérifier que l'on peut calculer les matrices de covariance empirique (ou empirique corrigée) en utilisant les fonctions `centre` et `prodtrans`.

2.2 Structures de données et analyse exploratoire

Analyse exploratoire des Iris de Fisher

Charger le jeu de données `iris` en mémoire, puis observer le résultat des instructions suivantes :

```
> data(iris)
```

```
> class(iris)
> names(iris)
> iris[,1]
> iris$Sepal.Length
> class(iris[,1])
> class(iris$Species)

> summary(iris)
> apply(iris[,1:4],2,mean)
> cor(iris[,1:4])
> print(cor(iris[,1:4]),digits=3)
> plot(iris)
> boxplot(iris)
```

Il est possible d'afficher plusieurs graphiques sur la même fenêtre :

```
> def.par <- par(no.readonly=T)
> par(mfrow=c(2,2))
> for(i in 2:5)
> {
>   hist(iris[,i])
> }
> par(def.par)
```

Que font les commandes suivantes ?

```
> pie(summary(iris$Species))
> barplot(summary(iris$Species))
```

Que font les commandes suivantes ? Comment fonctionnent-elles ?

```
> quartz() # ou x11()
> plot(iris[,1:4], col=c("red","green","blue")[iris$Species])
> quartz() # ou x11()
> pairs(iris[,1:4],main="Iris de Fisher",pch=21,bg=c("red","green3","blue")[iris$Espèce])
```

La séquence d'instructions suivante permet de faire un histogramme plus élaboré de manière à représenter la distribution d'un caractère en distinguant les espèces :

```
> attach(iris)

> # Histogrammes avec les espèces
> inter<-seq(min(Petal.Length),max(Petal.Length),by=(max(Petal.Length)-min(Petal.Length))/10)
> h1<-hist(plot=F,Petal.Length[Species=='setosa'],breaks=inter)
> h2<-hist(plot=F,Petal.Length[Species=='versicolor'],breaks=inter)
> h3<-hist(plot=F,Petal.Length[Species=='virginica'],breaks=inter)
> barplot(rbind(h1$counts,h2$counts,h3$counts),space=0,
>   legend=levels(Species),main="LoPe",col=c('blue','red','yellow'))

> # Graphique sur un fichier Postscript
> postscript('exemple.eps',horizontal=F,width=12/2.5,height=12/2.5)
> pairs(iris[2:5],main="Les Iris",pch=21,bg=c("red","green3","blue")[Species])
> dev.off()

> detach(iris)
```

Exercice : fonction `hist.factor`

Sur la base du code précédent, écrire une fonction nommée `hist.factor` qui, à partir d'une variable quantitative et d'une variable qualitative, affiche un histogramme permettant de visualiser dans chaque « bin » les effectifs selon les modalités de la variable qualitative.

Application : notes médian SY02 printemps 2014

Charger le jeu de données contenu dans le fichier `median-sy02-p2014.csv` et supprimer les lignes correspondant aux absents :

```
> notes <- read.csv("median-sy02-p2014.csv", header=F, na.strings=c("NA","ABS"))
> names(notes) <- c("branche","note")
> notes <- notes[-which(is.na(notes$note)),]
```

Convertir l'information de branche en chaîne de caractère, agréger l'information de branche, et reconvertir en variable qualitative :

```
> notes$branche <- as.character(notes$branche)
> notes$branche <- substr(notes$branche,1,2)
> notes$branche <- as.factor(notes$branche)
```

Visualiser les résultats de notes en fonction de la branche au moyen de la fonction `hist.factor`, puis au moyen de la fonction `boxplot`. Semble-t-il y avoir une influence de la branche sur les notes ?