Un très courte introduction à R

Matthieu Vignes, Emmanuel Rachelson

December 20, 2012

1 Un peu de contexte

R est un langage et un environnement de manipulation et d'analyse de données. Il va nous servir tout au long du module pour illustrer les différentes méthodes, en cours et en TP.

R désigne en fait l'environnement de manipulation des données et le langage de base. Les nombreuses fonctions d'analyse de données disponibles dans R sont implémentées au travers de packages. Les packages les plus courants sont installés à l'origine avec R, de nombreux autres peuvent être ajoutés. R est une implémentation du langage S, développée par Bell Labs. C'est un logiciel libre, distribué sous licence GNU GPL.

Le site du projet R : `http://www.r-project.org/`.

D'autres outils de manipulation de données et d'analyse statistique existent, tels que SAS, SPSS, JMPpro, etc.

2 But du TP

L'objectif ici est de se familiariser avec l'environnement de R, de connaître les commandes de base et de savoir trouver de l'aide. On va réutiliser R pendant toute la durée du cours, le but est que vous soyez aussi autonomes que possible par la suite.

3 Les bases

Pour démarrer : simple !

```
$ R
```

Séparateur d'instructions : `;`.

Symbole de commentaires : `#`.

Pour executer le code contenu dans un fichier, écrire les résultats dans un fichier ou les ramener dans le terminal :

```
> source(monfichiersource.R)
> sink(mesresultats)
> sink()
```

Pour obtenir de l'aide sur une commande :

```
> help(command)
> ?command
> help.start() # Aide au format HTML
> example(topic) # Exemples sur un sujet d'aide
```

R manipule des *objets*. Pour connaître la liste des objets existants à un moment :

```
> objects()
> ls()
```

Pour supprimer un objet nommé `obj` : `rm(obj)`.

A la fin d'une session, si on choisit de l'enregistrer, les objets sont stockés dans le fichier `.Rdata` et l'historique des commandes dans `.Rhistory`

Pour quitter une session :

```
> q()
```

Il existe un environnement graphique nommé *Rstudio* permettant d'écrire du code R, d'afficher des variables, des graphiques, etc. Ce TP peut-être mené directement dans la console de R ou via *Rstudio*.

4 Un exemple de session R commentée

4.1 Des nombres et des vecteurs

La stucture de données la plus simple en R est le vecteur.

Essayez les commandes suivantes :

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
> y <- c(x, 0, x)
```

Essayez les commandes suivantes :

```
> x
> y
> v <- 2*x+y+1
```

Attention : au cas où les dimensions des vecteurs ne concordent pas, les vecteurs les plus courts sont répétés en boucle pour arriver à la taille des vecteurs les plus longs.

Les opérations usuelles termes à terme : `+`, `-`, `*`, `/`, `^`, `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`...

Les opérations sur le vecteur tout entier : `min`, `max`, `range`, `length`, `sum`, `prod`, `mean`, `var`, `sort`...

Essayez les commandes suivantes :

```
> z <- 2*10:1 + 1
> z <- seq(length=11, from=10.1, by=-.2)
> z <- seq(along=x)
> x5 <- rep(x, times=5)
> x5 <- rep(x, each=5)
```

On peut manipuler des vecteurs de valeurs logiques. Par exemple :

```
> j <- x <= 10
> k <- x > 5.5
> j&k
> !(j|k)
> x + j&k
```

Les opérations logiques binaires : `<`, `<=`, `>`, `>=`, `==`, `!=`, `&`, `|`.

Attention aux conversions forcées :

```
> x + j&k
> x + (j&k)
```

Certains éléments d'un vecteur peuvent être inconnus. On parle alors de valeurs manquantes :

```
> z <- c(1:3, NA)
> is.na(z)
```

Attention, `NA` n'est pas une valeur mais un marqueur indiquant que la donnée n'est pas disponible. La commande suivante n'a donc pas de sens :

```
> z == NA
```

On peut aussi manipuler des chaînes de caractères comme des vecteurs :

```
> labels <- c("poids", "densité", "volume", "population")
> paste(labels, 1:5, sep="XX")
```

Essayez de sélectionner des sous-vecteurs avec les commandes :

```
> x[3]
> x[x<9]
> x[3:6]
> x[!is.na(x[3:6])] # un peu tiré par les cheveux
> x[-(1:3)]
```

On peut nommer les éléments d'un vecteur :

```
> val <- c(12.0, 1.3, 7, 10000)
> names(val) <- labels
> val[c("densité", "population")]
```

Les types atomiques, appelés `mode` sont `logical`, `numeric`, `complex`, `character`, `raw`. Le mode d'un vecteur est le mode de ses éléments. On peut créer des vecteurs par défaut de ces modes :

```
> numeric(3)
> character(5)
> logical(0)
> mode(x)
> mode(labels)
> length(x)
```

On peut convertir certaines expressions :

```
> as.character(x)
> as.integer(x)
> as.integer(labels)
```

Un vecteur peut être redimensionné en accédant à des éléments d'indice inconnu :

```
> e <- numeric()
> e[3] <- 17
> e[5] <- 3
> length(e) <- 4
```

4.2 Des objets, des tableaux, des listes, des data frames...

4.2.1 Des objets

Toutes les éléments créés lors d'une session R (y compris les fonctions) sont des *objets*. Ils possèdent tous les attributs intrinsèques de *mode* et *longueur*. Certains objets ont d'autres attributs, dont on peut obtenir la liste via la fonction `attributes(variable)` et que l'on peut modifier via la fonction `attr(variable, attribut)`.

```
> mode(x)
> length(x)
> attributes(x)
```

4.2.2 Des collections de catégories

Les `factor` sont des vecteurs un peu particuliers utilisés pour stocker des collections de catégories. Par exemple, imaginons qu'on ait une liste de 30 notaires de différentes villes de France. On peut créer le vecteur `villes` des villes où les notaires sont installés, puis le vecteur `revenus` de leurs revenus. Le factor `villesf` est la structure qui contient le vecteur des villes comme une collection de catégories discrètes, il a, à ce titre, un attribut `levels` supplémentaire.

```
> villes <- c("tls", "bdx", "par", "lyn", "lyn", "msl", "lil", "lil",
              "par", "bst", "lyn", "bst", "par", "par", "bdx", "tls",
              "bdx", "msl", "lil", "bst", "par", "lyn", "lyn", "lil",
              "bdx", "sbg", "lyn", "bst", "bst", "sbg")
> revenus <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56, 61, 61,
               61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46, 59, 46, 58, 43)
> villesf <- factor(villes)
> attributes(villesf)
> mode(villesf)
> class(villesf)
> levels(villesf)
```

On peut appliquer une fonction à chaque catégorie (chaque facteur) d'un vecteur `factor` :

```
> tapply(revenus, villesf, mean)
```

4.2.3 Des listes

Une `list` est une collection d'éléments de types différents.

```
> list1 <- list("Jean", 48, 74.5, c(12,7,5))
> list2 <- list(nom="Jean", age=22, poids=74.5, age.enfants=c(12,7,5))
> list1[[4]]
> list2$age.enfants
> list2[[4]] # le 4ème élément de la liste
> list2[4] # une liste contenant le 4ème élément de la liste
```

La concaténation de listes est une liste :

```
> list3 <- c(list1, list2)
> list3[[5]]
```

4.2.4 Des tableaux

Un `array` est un `vector` auquel on accède par un ensemble d'indices. Pour cela, un `array` dispose d'un attribut `dim`. Par exemple :

```
> z <- 1:30
> dim(z) <- c(3,2,5)
```

On peut également créer un `array` directement :

```
> array(1:13, dim=c(3,5))
> temp <- 1:13; dim(temp) <- c(3,5)
```

On accède aux éléments d'un `array` en donnant leur indice, un ensemble d'indices, ou une matrice d'indices :

```
> z[2,2,4]
> z[,,1]
> z[1:2,,1]
> indices <- array(0, dim=c(2,3));
> indices[1,] <- c(3, 1, 2); indices[2,] <- c(1,2,2)
> z[indices]
```

Attention, les opérations arithmétiques se font terme à terme dans les `array`. De façon générale, il faut veiller à ce que les `array` aient la bonne taille (la règle exacte est en fait un peu plus complexe).

```
> A <- array(1:12, c(3,4))
> B <- array(1:20, c(4,5))
> A*B
> A*A
> A^2+1
```

4.2.5 Des matrices

Des outils de calcul tensoriel comme la transposition généralisée `aperm(z, indices)` ou le produit tensoriel `z %o% x` ou `outer(z, x, *)` sont disponibles.

Une matrice est un `array` à deux dimensions. Beaucoup de fonctions de manipulations de matrices sont disponibles en R avec un nom évident (`det()`, `t()`, `tr()`...). Essayez les commandes suivantes :

```
> D <- matrix(c(1, 1, 0, 0, 3, 0, 0, 0, 2), nrow=3, ncol=3)
> A %*% B
> C = t(B) %*% B
> x %*% C %*% x # Notez l'abus sur les dimensions
> t(C)
> crossprod(x, C %*% x)
> diag(x)
> diag(C)
> diag(3)
> solve(2*diag(5), 1:5) # Résoud 2Ix=b avec b=1:5
> solve(2*diag(5))
> solve(C, 1:5)
> ev <- eigen(D);
> svd(A)
```

On peut également concaténer des matrices :

```
> cbind(1:3, 1:3, matrix(1:12, 3, 4), -1:1) % colonne par colonne
> cbind(1:3, 1:3, matrix(1:12, 3, 4), -1:1, 0:1)
> rbind(1, matrix(1:6, 2, 3))
```

4.2.6 Des tableaux de données

Les `data.frame` sont des listes un peu particulières qui décrivent les tableaux de données. On peut les voir comme des super-matrices dont les colonnes correspondent à un type de données. Chaque élément (chaque colonne) d'un `data.frame` a le même nombre d'éléments.

```
> D = cbind(1:30, rep(1:3,times=10))
> comptables <- data.frame(lieu=villesf, rev=revenus, D)
```

Pour alléger le code et faciliter l'accès aux données des `data.frame`, on peut utiliser `attach` / `detach`, qui ajoutent la `data.frame` au chemin de recherche des noms (attention aux conflits de noms).

```
> attach(comptables)
> lieu
> X2
> detach(comptables)
```

La fonction `search()` indique les packages et les `data.frame` utilisés pour définir les noms des objets accessibles. La fonction `ls()` est en fait équivalente à la fonction `ls(1)` qui liste les objets de l'environnement R. Si une `data.frame` a été attachée, alors elle apparait en seconde position et on peut retrouver ses éléments avec `ls(2)`.

Généralement, on rassemble toutes les données d'un même problème dans un unique `data.frame` pour y accéder en sécurité sans risquer de conflit de nommage, puis on attache/détache cette `data.frame` pour y travailler.

4.3 Lire dans des fichiers

En général, les données que l'on manipule sont fournies dans des fichiers. Par défaut, la fonction `read.table` de R attend des fichiers dont la première ligne indique les noms des variables décrites, puis, sur chaque ligne, un numéro de ligne et la valeur de chaque variable. On crée ainsi le `data.frame` correspondant aux données. Il existe aussi des fonctions de conversion comme `read.csv`. Par exemple :

```
> irisdf <- read.table("iris.data") # fichier au format R
> irisdf <- read.csv("iris.csv") # fichier au format csv
```

Pour écrire un `data.frame` dans un fichier :

```
> write.table(comptables, "comptables.data")
```

Notez qu'un certain nombre de jeux de données classiques (dont `iris`) sont disponibles dans R. Pour connaître leur liste et une courte description : `data()`.

R fournit également un éditeur très simple de tableaux en mode graphique.

```
> iris.new <- edit(iris)
> fix(comptables)
```

4.4 Blocs d'instructions et boucles

On peut regrouper une séquence d'insctructions au sein d'accolades `{}`. Dans ce cas, la valeur renvoyée par le bloc est la valeur renvoyée par sa dernière ligne.

On peut également construire des expressions conditionnelles et des boucles comme dans l'exemple suivant.

```
> x2 <- logical(length(x))
> for(i in 1:length(x)) {
+   if(x[i] < 7) {
+     x2[i] <- FALSE
+   }
+   else {
+     x2[i] <- TRUE
+   }
+ }
> x2
```

Les boucles `repeat{...}` et `while(condition) {...}` sont également possibles. L'instruction `break` permet d'arrêter une boucle. L'instruction `next` permet de sauter à l'itération suivante sans finir les instructions de l'itération courante.

4.5 Les fonctions

Parmi les objets que R peut manipuler, il y a ceux dont le `mode` est `function`. Ce sont des fonctions au même titre que `cos`, `mean`, `ls` et toutes celles disponibles dans l'environnement R (d'ailleurs la plupart des fonctions du langage sont elles-mêmes écrites en R). En soi, une fonction est déclarée comme :

```
> name <- fonction(arg1, arg2, ...) {instructions}
```

Par exemple :

```
> moyenne.empirique <- function(vec=x) {
+   x.temp <- x/length(x)
+   s <- sum(x.temp)
+   s
+ }
> moyenne.empirique(x)
> moyenne.empirique(vec=x)
```

On peut également définir des opérateurs binaires comme `+` ou `%*%`.

```
> a <- c(TRUE, FALSE, FALSE, FALSE, TRUE)
> b <- c(TRUE, TRUE,  FALSE, TRUE, FALSE)
> "%et%" <- function(a, b) {
+   a & b
+ }
> a %et% b
```

Les fonctions en R conservent la notion de portée : les variables définies au sein d'une fonction n'existent plus une fois l'appel à la fonction terminé, de même pour les affectations effectuées au sein de la fonction.

4.6 Graphiques

R propose des fonctions de construction de graphiques de "haut niveau" dont la principale est `plot`. Essayez les commandes suivantes :

```
> demande <- read.csv("demande.csv") # 100 jours de demande électrique française
> plot(comptables$lieu)
> plot(comptables$lieu, comptables$rev)
> plot(comptables)
> plot(seq(length=96, from=0.5, by=0.5), demande$jour_0)
```

D'autres fonctions d'affichage : `qqnorm`, `hist`, `persp`...

Les fonctions d'affichage de haut niveau, qui génèrent un nouveau graphique, peuvent prendre certains arguments permettant de préciser les axes, le style, les labels, etc.

```
>  plot(seq(length=96, from=0.5, by=0.5), demande$jour_0, type="l", ylab="demande",
+           xlab="heures", main="jour 0")
```

De façon générale, les (nombreux) paramètres graphiques sont fixés par la commande `par(...)` et l'aide (ou les divers tutoriaux sur R) fournit la liste complète de ces paramètres.

Il est également possible de faire appel à des fonctions de "bas niveau" qui ne créent pas de nouveau graphique mais modifient le graphique courant. Parmi celles-ci, `lines` rajoute des lignes, `points` rajoute des points. Parmi celles fréquemment utilisées : `abline`, `text`, `legend`, `title`...

Exercice : essayez d'afficher les 10 premiers jours de `demande` sur le même graphique (attention aux axes !). Pour s'entraîner, cela peut faire appel à une fonction, une boucle, des vecteurs, des fonctions d'affichage... Une fois cela fait, vous pouvez recommencer en regardant l'aide de `ts.plot` (et incidemment de `ts` et de `as.ts`).

Il existe enfin des fonctions graphiques "interactives" qui permettent de récupérer de l'information directement depuis le graphique. Essayez par exemple :

```
> locator(1)
> text(locator(1), "ici !")
```

5 Pour aller plus loin

L'introduction à R fournie par [1] (dont s'inspire largement ce tutoriel) est un bon point de départ pour aller plus loin dans la maîtrise du langage.

References

[1] *An Introduction to R.* W. N. Venables, D. M. Smith and the R Core Team, Version 2.15.2 (2012-10-26), available from `http://www.r-project.org/`