Institut de recherche
pour le développement

Formation R – Module 1

Emmanuel Paradis, Jean-Stéphane Bailly[1],

Bernard Palagos[2]

[1] Engref, [2] Cemagref

Montpellier, 15–17 octobre 2007

Sommaire

* Jean-Stéphane Bailly

I Premier contact avec R

R est à la fois un logiciel, un langage et un environnement de développement.

CRAN : Comprehensive R Archive Network
`http://cran.r-project.org/`

Utilisez les mirroirs (pour les IRDiens !) : Australie, Taïwan, Chili, . . .

Un « package » est un ensemble de « fonctions » documentées et testées pour marcher avec R.

Bref historique de R

1976 Première discussion sur S à Bell Labs

1978 S version 1

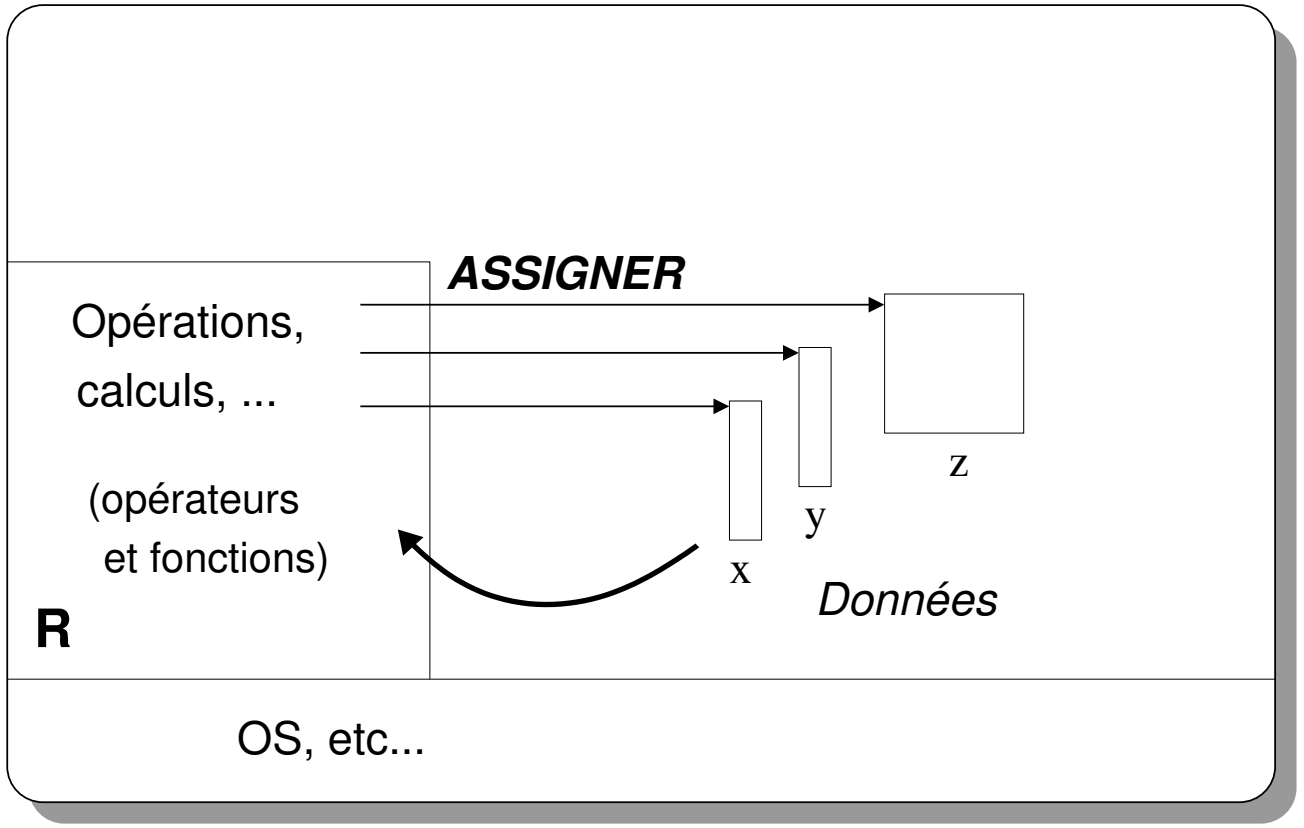
1993 Licence commerciale exclusive aboutit à S-Plus

1996 Ihaka R. & Gentleman R. 1996. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5 : 299–314. *(cité 2150 fois)*

2000 R 1.0

2007 (début octobre) R 2.6.0 ; $>$ 1000 « packages » sur CRAN ; 44 livres sur R ; 3000 citations (Web of Science) ; centaines de milliers d'utilisateurs

Comment R fonctionne



Mémoire vive (RAM)

Créer un objet avec l'opérateur « assigner » : `n <- 10`

Les opérateurs numériques : `+ - * / ^`.
Les parenthèses modifient les règles habituelles de priorité.

Gérer les objets en mémoire (1er contact avec les fonctions de R) : `ls() rm(x) str(x) ls.str()`

Générer des valeurs aléatoires (2ème contact) : `rnorm(10) runif(n)`

`ls()` pas besoin d'option, mais `ls(pattern = "x*")`

`rm(x, y, z)`, `rm(list = ls())`

Valeurs par défaut des arguments (options) : `rnorm runif`

Il y a deux façons de passer les arguments à une fonction : par position ou par les noms. Les commandes suivantes ont la même signification (mais pas le même résultat...) :

```
rnorm(5)
rnorm(5, 0, 1)
rnorm(mean = 0, sd = 1, n = 5)
```

Quels noms pour les objets ?

Un nom d'objet doit commencer par une lettre.

Les caractères autorisés sont : `A–Z` `a–z` `0–9` `.` `_`

Les caractères interdits sont :

- lettres accentuées
- tous les opérateurs : @, #, { }, ( ), ...
- \
- ~
- `;` (équivalent à "Enter")
- les 15 mots réservés du language : `for`, `in`, `if`, `else`, `while`, `next`, `break`, `repeat`, `function`, `NULL`, `NA`, `NaN`, `Inf`, `TRUE` et `FALSE`

Dans tous les cas un message d'erreur est affiché si vous tentez d'utiliser ces caractères.

Le nom d'un objet est limité à 256 caractères, alors n'hésitez pas à utiliser des noms explicites !

Les noms des fonctions peuvent être utilisés et cela ne pose généralement pas de problème...

```
> log <- 10.2
> log[1]
[1] 10.2
> log(1)
[1] 0
```

... mais autant l'éviter pour la lisibilité des commandes.De plus cela peut poser des problèmes si l'objet créé est une fonction.

Enfin, les objets dont le nom commence par un point sont cachés :

```
.a <- 1
ls()
ls(all.names = TRUE)
```

Le système d'aide

Les fonctions de R sont richement documentées, mais il peut être difficile pour un novice de s'y retrouver.

Deux problèmes sont fréquemment rencontrés.

1. Trouver une fonction
   → Démarrer R sans avoir une idée, même approximative, des fonctions que l'on va utiliser est une perte de temps.
   Se reporter à la documentation pour débutants, aux supports de cours, conserver ses scripts commentés si on est utilisateur occasionnel (*cf.* III).
   Pour afficher le contenu d'un package : `library(help = stats)`.
   Pour démarrer l'aide au format HTML : `help.start()`.
   Les moteurs de recherche sur CRAN.
   Pour chercher une fonction dont le nom contient "xxx" : `apropos("xxx")`.
2. Trouver les détails sur une fonction
   `?` est l'opérateur pour accéder à une page d'aide.
   `?ls` et `help(ls)` sont équivalents.
   Pour afficher les arguments d'une fonction : `args(ls)`.

Fermer R

```
> q()
Save workspace image? [y/n/c]:
```

Répondre “y” aboutit à enregistrer deux fichiers dans le répertoire courant : ‘.Rhistory’ avec l’historique des commandes de la session, et ‘.RData’ avec les objets listés par `ls()`.

Lorsque R démarre, s’il y a un fichier ‘.RData’ dans le répertoire il est chargé en mémoire :

```
...
[Previously saved workspace restored]
```

Le format des fichier *.RData ou *.rda (nommé XDR) est binaire (illisible avec un éditeur texte). Il est possible de sauvegarder ainsi nimporte quel objet :

```
save(x, y, df, foo, file = "session_Aedes.RData")
```

Ces fichiers sont chargées en mémoire avec :

```
load("session_Aedes.RData")
```

⚠ Dans cet exemple, s'il y a des objets nommés `x`, `y`, `df` ou `foo` en mémoire, ils seront effacés par `load`. ***Utiliser des noms explicites.***

`save` ou `write.table` ?

`save` permet de sauvegarder tous les types d'objets (y compris des fonctions modifiées, des formules, ...) sans souci de formatage. Les fichiers *.RData sont plus compacts et chargés en mémoire plus rapidement qu'un fichier ASCII.

`write.table` permet d'écrire des fichiers lisibles par d'autres logiciels (et l'œil humain).

Exercices I

1. Créer un objet `x` avec la valeur 1. Créez ensuite un objet, également nommé `x`, avec une valeur aléatoire $\mathcal{N}(0, 1)$. Qu'est devenu le premier objet ?
2. Créer deux objets `x` et `y` identiques en deux commandes, puis avec une commande unique (il y a deux possibilités).
3. Afficher toutes les fonctions de R qui produisent des graphes ("plot" en anglais). Faites de même avec celles qui produisent un affichage.

Résumé I : premier contact avec R

- R manipule les données dans la mémoire vive de l'ordinateur.
- Une fonction a besoin des parenthèses pour être exécutée, même s'il n'y a pas d'arguments.
- Les arguments sont séparés par des virgules. Souvent, des arguments ont une valeur par défaut.
- Le résultat de l'exécution d'une fonction est un objet unique qui peut être stocké en mémoire avec `<-`.
- Les pages d'aide sont affichées dans R avec `'?'`.

II La structure des données

Vecteur

1
2
3
4
5

"Homo"
"Pan"
"Gorilla"

length (= 5)

mode (= "numeric")

length (= 3)

mode (= "character")

} **Attributs**

length : nombre d'éléments ***mode*** : numeric, character, logical (complex, raw)

Comment construire un vecteur ?

1. Séries régulières : ‘`:`’ `seq(from, to, by)` `rep` :

```
rep(1:2, 10)
rep(1:2, each = 10)
rep(1:2, each = 2, length.out = 20)
```

2. Séries aléatoires : `rloi(n, ...)`
3. Vecteurs “par défaut” : `numeric(2)` `logical(10)` `character(5)`
4. Concaténer des vecteurs : `c(x, y)`
5. Entrer direct au clavier avec `scan()` (numérique) ou `scan(what = "")`
6. Lecture de fichiers

Quelque soit le mode, une valeur manquante est indiquée `NA` (*not available*) mais est stockée de façon appropriée.

```
x <- c("NA", NA)
x
is.na(x)
```

Les valeurs numériques infinies sont indiquées `Inf` ; `NaN` signifie “not a number”.

```
x <- -5/0
x
exp(x)
Inf + Inf
Inf - Inf
```

Une matrice (***matrix***) est un vecteur arrangé de façon rectangulaire.

Comment construire une matrice ?

1. Avec la fonction `matrix(NA, nrow, ncol, byrow = FALSE)`
2. À partir d’un vecteur : `dim(x) <- c(nr, nc)`, si `length(x) == nr*nc` !
3. En joignant des vecteurs avec `rbind` ou `cbind`.

Facteur

Un facteur (***factor***) est un vecteur de mode numérique codant une variable qualitative (couleur, ...). L'attribut `"levels"` spécifie les noms des niveaux. Certains niveaux peuvent ne pas être présents.

Un facteur ordonné (***ordered***) a une hiérarchie dans ses niveaux (ex : TB, B, m, M, TM).

Comment construire un facteur ?

1. Séries régulières : `gl(n, k, n*k)` (*generate levels*)
2. Avec la fonction `factor(x, levels = )`
3. À partir d'un vecteur numérique `x` : `cut(x, breaks)` (*cf.* `?cut pour les détails)`)
4. Voir la fonction `stack` plus loin (à partir d'un tableau).
5. Lecture de fichiers

⚠ Les facteurs ne peuvent pas être concaténés avec `c(x, y)`

Tableau de données et liste

Un tableau de données (***data frame***) est un ensemble de vecteurs et/ou de facteurs tous de la même longueur.

Comment construire un tableau de données ?

1. Avec la fonction `data.frame`
2. Lecture de fichiers

Une liste (***list***) est un ensemble d'objets quelconques.

Comment construire une liste ?

1. Avec la fonction `list`
2. Lecture de fichiers (avec `scan`)

Exercices II

1. Quels sont le mode et la longueur de l'objet retourné par la commande `ls()` ?
2. Créer un vecteur avec votre nom et votre âge. Commentez sur les éventuelles erreurs rencontrées.
3. Créer un objet `x` prenant les valeurs entières de 20 à 1. Modifiez cette commande pour quelle marche avec un nombre entier `n` (égal à 20 dans le cas présent).
4. Créer un objet `y` avec des valeurs aléatoires uniformes $\mathcal{U}(0, 1)$; la longueur de `y` étant identique à celle de `x`.
5. Créer un vecteur `z` avec une valeur selon $\mathcal{U}(0, 1)$. Créez ensuite un tableau de données avec `x`, `y` et `z`. Nommez ce tableau `DF`.
6. Créer une liste `L` avec `DF` et `z`. Affichez `L` : voyez-vous une différence entre les deux `z` ?

Résumé II : la structure des données dans R

- Les vecteurs (*vector*) sont les briques fondamentales des données. Le ***mode*** d'un vecteur peut être *numeric*, *character* ou *logical*.
- Les facteurs (*factor*) sont des vecteurs de mode numeric servant à coder les variables qualitatives (= catégorielles).
- Les tableaux de donfactornées (*data frame*) sont des collections de vecteurs et de facteurs de même longueur.
- Une liste (*list*) est une collection d'objets quelconques.

III Gérer ses scripts de commandes

Un bon éditeur est utile dès qu'on débute avec R. Il permet notamment de :

1. colorer la syntaxe,
2. "allumer" les parenthèses, crochets et accolades,
3. ajouter et éditer des commentaires,
4. éventuellement envoyer des lignes ou des blocs de commandes directement vers R.

Le principal intérêt d'un script de commandes est de pouvoir répéter les analyses (intérêt pratique mais aussi fondamental). Un autre intérêt, mais non moins négligeable, est d'ajouter des commentaires.

Pour répéter des analyses à partir d'un script :

1. copier/coller vers la console ;
2. envoyer les commandes vers R (si possible) ;
3. `source("script_Aedes_morpho_Dakar.R")`
4. `R CMD BATCH script_Aedes_morpho_Dakar.R`

L'avantage de `R CMD BATCH` est que les commandes et les résultats sont dans le même fichier ('script_Aedes_morpho_Dakar.Rout').

Avec `source` ou `R CMD BATCH`, chaque ligne de commentaire doit être précédée par `#`. Malheureusement, R n'a pas de syntaxe pour exclure un bloc de lignes de l'exécution. Cependant, il y a quelques trucs.

```
"
..........
... bloc exclu de l'exécution
........
"
```

⚠ Le bloc exclu ne doit pas contenir de "double quotes". S'il y en a, mettre des "simple quotes" (mais il ne faut pas qu'il y en ait dans le bloc).

Autre inconvénient, R doit lire la chaîne de caractères et créer la variable correspondante (sans la stocker toutefois).

Si le bloc est syntactiquement correct :

```
if (FALSE) {
..........
... bloc exclu de l'exécution
........
}
```

Ajouter des commandes dans un script

1. copier/coller depuis la console ;
2. `savehistory("R_script_today.R")` sauvegarde toutes les commandes de la session en cours sur le disque. Si aucun nom de fichier n'est précisé, il sera nommé '.Rhistory' et sera donc caché (et pas forcément associé avec votre éditeur de scripts R).

⚠ Il est vivement conseillé "d'aérer" les opérateurs :

```
x > -1          x>-1
x < -1          x<-1   # :(
```

Ajouter des messages dans un script

Le plus simple : `message("Debut des calculs...")`
Plus intéressant : `cat("n =", n, "\n")`
`stop("......")` : arrête l'exécution et affiche le message d'erreur `"......"`

Sweave et odfWeave

Un fichier Sweave mêle les commandes LaTeX avec des commandes R. Par exemple le fichier 'AedesDakar.Rnw' :

```
[...]
We sampled two populations of {\it Aedes} sp.:

<<>>=
AedesDakar <- read.table("AedesDakar.txt", header = TRUE)
summary(AedesDakar)
@

The distributions of the variables look like this:

<<fig=true>>=
pairs(AedesDakar)
@
[....]
```

Dans R, la commande `Sweave("AedesDakar.Rnw")` exécute les instructions R, enregistre les éventuels graphiques au format EPS et PDF sur le disque, crée un fichier 'AedesDakar.tex' avec le texte, les commandes R et les résultats dans une police spéciale, et les instructions appropriées pour inclure les graphiques. Il suffit ensuite de compiler ce fichier avec LaTeX.

Le package **odfWeave** offre des fonctionnalités similaires pour les document au format ODF (Open Document Format) utilisé notamment par OpenOffice (*R News* 6/4, October 2006).

Résumé III : scripts de commandes

Il est important de garder ses commandes dans des fichiers scripts '.R' pour la répétabilité des analyses.

VI Manipulation de données

Le système d'indexation des vecteurs : `[ ]`

1. Numérique : positif (extraire, modifier et/ou 'allonger') OU négatif (extraire uniquement).
2. Logique : le vecteur d'indices logiques est éventuellement recyclé (sans avertissement) pour extraire, modifier et/ou allonger.
3. Avec les noms (***names*** = vecteur de mode character) ; pour extraire ou modifier

Extraction et “subsetting” des matrices, tableaux et listes

1. `[, ]` (les 3 systèmes) pour matrices et tableaux mais :
   - allongement impossible,
   - `drop = TRUE` par défaut.

   Il n’y a pas de names mais `colnames` et/ou `rownames`.
2. Extraction à partir d’un tableau ou d’une liste : `$` (avec noms) `[[` (numérique ou avec noms).
3. Subsetting à partir d’un tableau ou d’une liste : `[` (les 3 systèmes).

   `subset` est une fonction qui permet de faire le même genre d’opération de sélection de lignes et/ou colonnes d’une matrice ou d’un tableau.

Les conversions

R a 105 fonctions `as.`*XXX* pour convertir les objets. Cela peut concerner le mode (`as.numeric`, `as.logical`, ...), le type de données (`as.data.frame`, `as.vector`, ...), ou même des objets du language (`as.formula`, `as.expression`, ...).

⚠ R effectue parfois des conversions implicites (***coercions***) :

```
"0" == 0
"0" == FALSE
0 == FALSE
```

Manipulation des vecteurs logiques

Les vecteurs logiques sont le plus souvent générés avec les opérateurs de comparaison : `==` `!=` `<` `>` `<=` `>=`.

L'opérateur `!` inverse les valeurs d'un vecteur logique (avec d'éventuelles coercions : `!0` `!1`).

L'opérateur `&` compare deux vecteurs logiques élément par élément et retourne un vecteur logique avec `TRUE` si les deux éléments le sont aussi, `FALSE` sinon.

L'opérateur `|` fait la même opération mais retourne `TRUE` si au moins un des éléments l'est aussi.

La fonction `xor` fait la même opération mais retourne `TRUE` si un seul des éléments l'est aussi.

R a 55 fonctions de la forme `is.`*XXX* (`is.numeric`, `is.logical`, `is.factor`, `is.na`, `is.data.frame`, ...)

Trois fonctions utiles pour manipuler les vecteurs logiques :

- `which` retourne les indices des valeurs `TRUE`,
- `any` retourne `TRUE` s'il y a au moins une valeur `TRUE`,
- `all` retourne `TRUE` si elles le sont toutes.

⚠ L'opérateur `==` n'est pas toujours approprié pour comparer des valeurs numériques (sensibilité à la précision numérique) : utiliser plutôt la fonction `all.equal`.

Manipulation des facteurs

Un facteur est en fait stocké sous forme d'entiers avec un attribut qui spécifie les noms des niveaux (*levels*) :

```
> f <- factor(c("a", "b"))
> f
[1] a b
Levels: a b
> str(f)
 Factor w/ 2 levels "a","b": 1 2
```

Donc si on traite les facteurs comme des vecteurs ordinaires, on manipule les codes numériques.

```
> c(f) # == as.integer(f)
[1] 1 2
> as.vector(f) # == as.character(f)
[1] "a" "b"
```

Les niveaux peuvent être extraits ou modifiés avec la fonction `levels` :

```
> levels(f) <- c(levels(f), "c")
> f
[1] a b
Levels: a b c
> table(f)
f
a b c
1 1 0
> g <- factor(c("a", "b", "c", "d"))
> levels(f) <- levels(g)
> f
[1] a b
Levels: a b c d
```

Pour supprimer les niveaux absents :

```
> factor(f)
[1] a b
Levels: a b
```

Pour concaténer des facteurs :

```
> h <- factor(c(as.vector(f), as.vector(g)))
> h
[1] a b a b c d
Levels: a b c d
```

Note : l'indexation d'un facteur préserve les niveaux :

```
> h[1]
[1] a
Levels: a b c d
```

Exercices VI

1. Construire un vecteur `x` avec 30 valeurs selon $\mathcal{N}(0, 1)$. Sélectionner les éléments d'indices impairs. Trouver une solution qui marche quelque soit la longueur de `x`. Extraire les valeurs de `x` positives.
2. Créer un vecteur avec trois noms de taxons de votre choix. Afficher le mode de ce vecteur. Créer un vecteur numérique avec les tailles (approximatives) de ces taxons. Trouver comment extraire une taille avec un nom de taxon.
3. Créer un tableau de données avec trois observations (lignes) et deux variables numériques de votre choix. Extraire les noms des rangées de ce tableau avec `rownames`. Qu'observez-vous ? Modifier ces noms avec les noms de taxons choisis ci-dessus.
4. Extraire la première colonne de ce tableau avec `[` puis avec `[[`. Comparer les résultats.
5. Effacer cette première colonne.

Résumé VI : manipulation de données

- Usage typique des opérateurs suivants :
  - `[ ]` indexation numérique, logique ou par noms des vecteurs
  - `[ , ]` indexation des matrices et tableaux
  - `[[ ]]` extraction d'éléments d'un tableau ou d'une liste
  - `$` extraction par le nom d'éléments d'un tableau ou d'une liste
  - `[ ]` "subsetting" d'un tableau (colonnes) ou d'une liste
- Les facteurs doivent être manipulés avec précaution.

VII Transformation de variables

Transformation

Les fonctions suivantes transforment un vecteur numérique et retournent un vecteur de même longueur :

```
sqrt abs sign log exp [a]sin[h] [a]cos[h] [a]tan[h]
```

```
sign(x) == x/abs(x)
```

Pas d'option sauf `log(x, base = exp(1))` ; `log10(x)` est un raccourci pour `log(x, base = 10)`.

Deux opérateurs spéciaux :

`x %% y` : x modulo y
`x %/% y` : combien de fois *entière* y dans x

Des fonctions spécialisées dont (liste exhaustive dans `?Special`) :

`gamma(x)`	$\Gamma(x)$
`choose(n, k)`	C_n^k (souvent noté $\binom{k}{n}$)
`factorial(x)`	$\prod_{i=1}^{x} i \equiv x!$; équivalent à `prod(1:x)`
`lfactorial(x)`	$\sum_{i=1}^{x} \ln i$; équivalent à `sum(log(1:x))` mais comparer :

```
lfactorial(1e7)
sum(log(1:1e7))
```

Arrondis

⚠ Il est important de distinguer la précision d'un nombre tel qu'il est stocké en mémoire (et utilisé dans les calculs) et la précision affichée à l'écran (souvent tronquée).

```
pi
print(pi, digits = 1)
print(pi, digits = 16)
```

`ceiling(x)` : arrondit à l'entier supérieur ou égal

`floor(x)` : arrondit à l'entier inférieur ou égal

`trunc(x)` : supprime les décimales ; identique à `floor(x)` si `x >= 0`

`round(x, digits = 0)` : arrondit à `digits` décimales

`signif(x, digits = 6)` : arrondit à `digits` chiffres ; comparer :

```
print(round(pi*1000, 6), digits = 18)
print(signif(pi*1000, 6), digits = 18)
# mais:
print(round(pi/10, 6), digits = 18)
print(signif(pi/10, 6), digits = 18)
```

`zapsmall(x, digits = getOption("digits"))` : appèle `round(x, 7 - log10(max(x)))`

Tri de variables

`rev(x)` inverse :

- les éléments de `x` si c'est un vecteur ou une matrice (qui sera convertie en vecteur)
- les colonnes de `x` si c'est un tableau
- les éléments de `x` si c'est une liste

`sort(x)` trie les éléments de `x` et retourne le vecteur trié. L'option `decreasing = TRUE` permet de trier dans le sens décroissant (plus efficace que `rev(sort(x))`).

`order` réalise un tri multiple "hiérarchisé" sur un ensemble de vecteurs : un tri est fait sur le 1er vecteur ; s'il y a des égalités de rang elles sont résolues avec le 2nd vecteur, et ainsi de suite. Cette fonction retourne les indices ainsi triés.

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

Il est possible de mélanger les modes :

```
> order(c(1, 2, 1, 2), c("b", "b", "a", "a"))
[1] 3 1 4 2
```

```
x <- 10:1
sort(x)
order(x)
all(sort(x) == x[order(x)])
```

Exemple de tri des lignes d'un tableau :

```
o <- order(df$x, df$y, df$z)
df[o, ]
```

`order` a aussi l'option `decreasing = FALSE` (par défaut).

Avec `sort` ou `order` il n'y a pas d'ex-aequo ("tie").

`rank` retourne les rangs d'un vecteur ; il y a plusieurs méthodes pour résoudre les ties (*cf.* `?rank` pour les détails).

```
x <- rep(1, 4)
rank(x)
order(x)
# mais:
order(x, 4:1)
```

Exercices VII

1. On sait que $\sqrt[n]{x} = x^{1/n}$. Comment tester si R retourne le même résultat pour ces deux opérations ?
2. Calculer le reste d'une division entière (il y a au moins deux possibilités).
3. Simuler 40 valeurs selon $\mathcal{U}(0, 3)$; les stocker dans `x`. Créer un facteur indiquant les trois classes d'intervalles 0–1, 1–2 et 2–3 de `x` (il y a au moins trois solutions).

Résumé VII : les transformations de variables

- La plupart des transformations mathématiques se font via des fonctions appliquées directement au vecteur.
- Les fonctions `round` et `signif` affectent le stockage des décimales et des chiffres, respectivement, alors que `print(x, digits)` affecte l'affichage uniquement.

VIII Description et résumé

Résumé et valeurs manquantes

Nous avons vu jusqu'ici des fonctions qui agissent sur chaque élément d'un vecteur.

```
> x <- c(1, NA)
> log(x)
[1]  0 NA
```

Mais quelle est le résultat de l'addition de 1 avec une valeur manquante ?

```
> sum(x)
[1] NA
```

`sum`, comme de nombreuses fonctions du même type, a une option `na.rm = TRUE`.

```
> sum(x, na.rm = TRUE)
[1] 1
```

`mean`, `var`, `median`, `quantile`, `max`, `min`, `range`, `prod`

`cumsum` et `cumprod` n'ont pas cette option mais, logiquement, retourne `NA` dès qu'une valeur manquante est rencontrée.

`summary` affiche, pour les vecteurs numériques, un résumé (minimum, maximum, quartiles, moyenne) avec un comptage des valeurs manquantes.
Si l'objet est une matrice ou un tableau, le résumé est fait pour chaque colonne.

Pour les calculs de matrices de corrélation ou de covariance, il y a une option `use` qui spécifie l'action à prendre en présence de valeurs manquantes :

- `use = "all.obs"` : erreur s'il y a au moins une valeur manquante dans la matrice,
- `use = "complete.obs"` : les lignes avec au moins une valeur manquante sont supprimées,
  ⚠ les corrélations sont influencées par l'échantillonnage (lignes et colonnes)
- `use = "pairwise.complete.obs"` : la suppression des lignes se fait au cas par cas pour chaque couple de variables,
  ⚠ les corrélations ne sont pas toutes calculées avec le même effectif.

Pour les facteurs, `table` calcule les effectifs observés pour chaque niveau, ou les combinaisons de niveaux pour plusieurs facteurs :

```
g1 <- gl(2, 10); g2 <- gl(2, 1, 20); g3 <- gl(2, 2, 20)
table(g1)
table(g1, g2)
table(g1, g2, g3)
```

`ftable` (*flat table*) présente les résultats de `table` sous forme tableaux plats :

```
ftable(table(g1, g2, g3))
```

Opérations sur les tableaux

`stack` reformatte un tableau en empilant les colonnes numériques pour faire un vecteur unique et un facteur indiquant les colonnes d'origine :

```
df <- data.frame(x = rnorm(10), y = rnorm(10, 1))
stack(df)
```

Les colonnes non-numériques sont ignorées. Le résultat est un tableau. À partir d'une matrice, utiliser `as.data.frame` au préalable.

`unstack` fait l'opération inverse.

```
all.equal(unstack(stack(df)), df)
```

Faire une opération sur toutes les lignes et/ou colonnes d'une matrice ou d'un tableau :

```
apply(df, 1, sum)
```

Le `1` indique que la fonction est appliquée à chaque ligne de `df` : le changer par `2` pour que ce soit sur les colonnes.

Les arguments éventuellement nécessaires à la fonction donnée en argument sont ajoutés à la suite (comme `na.rm = TRUE`).

Appliquer une fonction à chaque élément d'une liste :

```
lapply(L, sum)
```

Les résultats sont retournés sous forme de liste. `sapply` fait la même opération mais retourne les résultats sous forme plus "conviviale" (vecteur ou matrice avec `*names`).

`rapply` est une variante “récursive” de `lapply` (dans l’exemple suivant `lapply` retourne `NA`) :

```
> L <- list(list(rnorm(10), rnorm(100)),
+           list(rnorm(10, 1), rnorm(100, 1)))
> rapply(L, mean)
[1] 0.08877853 0.07190452 0.90469646 1.17776931
```

Deux fonctions permettent d’appliquer une fonction à un ou plusieurs vecteurs (sous forme de matrice ou de tableau) pour chaque niveau d’un facteur.

```
by(data, INDICES, FUN, ...)
aggregate(x, by, FUN, ...)
```

Le premier argument est identique : un vecteur, une matrice ou un tableau. Pour `by` le second argument est soit un facteur ou une liste de facteurs, alors que pour `aggregate` les facteurs sont obligatoirement passés sous forme de liste, même s’il n’y en a qu’un.

La principale différence est dans la façon dont les résultats sont retournés :

```
> mean0 <- c(rnorm(10), rnorm(10, 0, 1))
> mean1 <- c(rnorm(10, 1), rnorm(10, 1, 1))
> gr <- gl(2, 10, labels = c("var=0", "var=1"))
> dfNorm <- data.frame(mean0, mean1)
> by(dfNorm, gr, mean)
gr: var=0
     mean0      mean1
-0.2322668  1.1473740
------------------------------------------------------------
gr: var=1
     mean0      mean1
0.09352467 1.09244319
> aggregate(dfNorm, list(gr), mean)
  Group.1       mean0    mean1
1   var=0 -0.23226678 1.147374
2   var=1  0.09352467 1.092443
```

À signaler une autre fonction :

```
> args(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

L'argument principal est ici un vecteur, et `INDEX` est, comme pour `aggregate`, une liste. Le résultat est retourné sous forme de vecteur ou de matrice avec des `*names` correspondants aux niveaux de `INDEX`.

Exercices VIII

1. Reprendre l'exemple avec `dfNorm` et refaire l'analyse réalisée avec `aggregate` mais avec `tapply`.
2. Lire le fichier 'Mammal_lifehistories_v2.txt' [a]. Afficher les noms des colonnes. Combien d'observations par ordre ?
3. Extraire les colonnes 5 et 12 et les stocker dans des vecteurs ayant des noms appropriés. Faire un descriptif de ces deux vecteurs.
4. Calculer les moyennes et variances de ces deux variables pour chaque ordre.

[a] source : www.esapubs.org/archive/ecol/E084/093/default.htm

Résumé VIII : description et résumé

- La plupart des fonctions produisant un résumé d'un vecteur ont l'option `na.rm = FALSE` par défaut, consuisant à retourner `NA` s'il y a au moins une valeur manquante.
- Les fonctions `cov` et `cor` retournent, par défaut, une erreur s'il y a au moins une valeur manquante dans la matrice ou le tableau de données.

IX Les tests statistiques

Les fonctions sont de la forme `xxx.test`.

Exemple : `t.test`.

Sauriez-vous utiliser cette fonction ?

Essayez cette fonction dans le cas où l'hypothèse nulle est vraie, et dans le cas où elle est fausse.

Le test de t est-il robuste à une variance hétérogène ?

Exercices IX

1. Afficher la liste des fonctions de R faisant des tests statistiques.
2. Les masses corporelles moyennes des artiodactyles et des carnirores sont-elles différentes ? Quel test préliminaire peut-on faire sur ces deux variables avant de comparer leurs moyennes ?
3. Répéter cette analyse pour les tailles de portées.

X Les modèles linéaires

Le modèle : $E(y) = \beta_1 x_1 + \beta_2 x_2 + \cdots + \alpha$

y :	réponse
$x_1, x_2, \ldots$:	prédicteurs
$\beta_1, \beta_2, \ldots, \alpha$:	paramètres

Les observations : $y_i = \beta_1 x_{1i} + \beta_2 x_{2i} + \cdots + \alpha + \epsilon_i$

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

Pour une variable qualitative (*aka* catégorielle) z, il faut utiliser un codage numérique sensé : les ***contrastes***.

Exemple 1 : $z = \{R, B\}$, z est substitutée par x_z :

$z = R \rightarrow x_z = 0$
$z = B \rightarrow x_z = 1$

Le modèle (numérique) : $E(y) = \beta x_z + \alpha$

$z = R \rightarrow E(y) = \alpha$
$z = B \rightarrow E(y) = \beta + \alpha$

Exemple 2 : $z = \{R, B, V\}$, z est substitutée par x_{z_1} et x_{z_2} :

	x_{z_1}	x_{z_2}	
R	0	0	$E(y) = \alpha$
B	1	0	$E(y) = \beta_1 + \alpha$
V	0	1	$E(y) = \beta_2 + \alpha$

$$E(y) = \beta_1 x_{z_1} + \beta_2 x_{z_2} + \alpha$$

Pour une variable avec n catégories, $n - 1$ variables $0/1$ sont créées ; il y a donc $n - 1$ paramètres supplémentaires associés à l'effet de cette variable. D'où les $n - 1$ ddl associés au test de cet effet, par opposition à 1 ddl avec un prédicteur continu.

Formulation générale des modèles linéaires

"There is no essential distinction between linear models we happen to call *Regression* and those we call *Analysis of Variance* models"
W. N. Venables

$E(y) = \beta x$	Régression linéaire
$E(y) = \beta z$	Analyse de variance (ANOVA)
$E(y) = \beta_1 x + \beta_2 z$	Analyse de covariance (ANCOVA)

Dans tous les cas les erreurs sont normalement distribuées autour de la moyenne : ces modèles sont ajustés par la méthode des moindres carrés (minimiser $\sum(y_i - \widehat{y}_i)^2$).

Dans R, `aov` et `lm` produisent le même ajustement ; la différence est dans l'affichage des résultats.

Les interactions entre variables

Deux cas : continue $\times$ qualitative, qualitative $\times$ qualitative

1. continue $\times$ qualitative ($x \times z$)

   Pour coder cette interaction, une nouvelle variable est créée par le produit de x et du codage numérique de z (x_z) ; le modèle devient :

   $E(y) = \beta_1 x + \beta_2 x_z + \beta_3 (x x_z) + \alpha$ (Le modèle reste linéaire !)

   $z = M \rightarrow E(y) = \beta_1 x + \alpha$
   $z = F \rightarrow E(y) = \beta_1 x + \beta_2 + \beta_3 x + \alpha = (\beta_1 + \beta_3) x + \beta_2 + \alpha$

   Si pas d'interaction ($\beta_3 = 0$), la pente est la même pour les deux catégories.

   Pour n catégories, $n - 1$ nouvelles variables sont créées pour coder l'interaction.

2. qualitative $\times$ qualitative ($z_1 \times z_2$)

   De nouvelles variables sont créées avec le produit de toutes les combinaisons 2 à 2 possibles entre les codages numériques des deux variables.

$$z_1 = M \rightarrow x_{Z_1} = 0$$
$$z_1 = F \rightarrow x_{Z_1} = 1$$
$$z_2 = bleu \rightarrow x_{Z_2} = 0$$
$$z_2 = vert \rightarrow x_{Z_2} = 1$$

$$E(y) = \beta_1 x_{Z_1} + \beta_2 x_{Z_2} + \beta_3 (x_{Z_1} x_{Z_2}) + \alpha$$

M	$bleu$	$E(y) = \alpha$
	$vert$	$E(y) = \beta_2 + \alpha$
F	$bleu$	$E(y) = \beta_1 + \alpha$
	$vert$	$E(y) = \beta_1 + \beta_2 + \beta_3 + \alpha$

Si pas d'interaction ($\beta_3 = 0$) :

$$\begin{array}{lll} M & bleu & E(y) = \alpha \\ & vert & E(y) = \beta_2 + \alpha \\ F & bleu & E(y) = \beta_1 + \alpha \\ & vert & E(y) = \beta_1 + \beta_2 + \alpha \end{array}$$

Le "contraste" entre $bleu$ et $vert$ est le même pour M et F (et vice versa).

Pour le cas de deux variables avec n_1 et n_2 catégories, respectivement, $(n_1 - 1)(n_2 - 1)$ nouvelles variables $0/1$ sont créées.

Pour les interactions d'ordre supérieur (entre trois variables ou plus), les combinaisons 3 à 3, 4 à 4, etc, sont utilisées.

Les formules

`y ~ x1 + x2`	effets additifs
`y ~ x1 * x2`	effets additifs et interaction
	identique à `y ~ x1 + x2 + x1:x2`

Les formules sont des objets comme les autres :

```
form <- list(y ~ x1, y ~ x2, y ~ x1 + x2,  y ~ x1 * x2)
lapply(form, lm, data = DF)
```

Usage typique de `lm` :

```
Aedes2002.mod1 <- lm(surv ~ locality * age, data = Aedes2002)
# == lm(Aedes2002$surv ~ Aedes2002$locality * Aedes2002$age)
```

`data` peut être un tableau ou une liste. Par défaut les variables sont cherchées dans l'espace de travail.

```
# modèle additif avec toutes les colonnes de Aedes2002
# (sauf 'surv'):
lm(surv ~ ., data = Aedes2002)
# ... et avec les interactions de degré 2:
lm(surv ~ .^2, data = Aedes2002)
# etc...
```

⚠ Un prédicteur entier est traité comme une variable continue :

```
y <- rnorm(40)
x <- rep(1:4, each = 10)
lm(y ~ x)
lm(y ~ factor(x))
```

Comment faire la régression par l'origine ?

```
lm(surv ~ locality * age - 1, data = Aedes2002)
```

Comment ajouter un terme constant au modèle ?

```
lm(surv ~ locality + offset(2.05 * age), data = Aedes2002)
# == lm(surv ~ locality, offset = 2.05 * age, data = Aedes2002)
# == lm(surv - 2.05 * age ~ locality, data = Aedes2002)
```

Et les données manquantes ?

`lm` a l'option `na.action` qui est par défaut `na.omit` (les observations avec au moins une valeur manquante sont éliminées).

`na.action = na.fail` résulte en une erreur s'il y a au moins une valeur manquante.

`model.matrix` permet de visualiser le modèle numérique créé par une formule.

```
g1 <- gl(2, 10)
g2 <- gl(2, 2, 20)
g3 <- gl(2, 1, 20)
table(g1, g2, g3)
y <- rnorm(20)
model.matrix(y ~ g1 * g2 * g3)
```

Résumés et tests statistiques

```
frm <- surv ~ locality * age
res.lm  <-  lm(frm, data = Aedes2002)
res.aov <- aov(frm, data = Aedes2002)
```

1. `summary(res.aov)` : tableau d'ANOVA (= tests sur les effets $\sim F$)

   `summary(res.lm)` : tests sur paramètres ($\sim t$)

2. `anova`

   Si un modèles : tableau d'ANOVA en incluant les effets dans l'ordre de la formule.

   Si plusieurs modèles : tableau d'ANOVA entre les modèles.

   (a) `anova(res.lm)` et `summary(res.aov)` sont identiques.

   (b) L'ordre des termes dans la formule est important s'il y a plusieurs prédicteurs qualitatifs et que les effectifs retournés par `table(...` sont inégaux ("unbalanced design").

3. `drop1` : teste les effets individuels *vs.* le modèle complet.

   (a) Le principe de marginalité est respecté.

   (b) `drop1(res.lm)` et `summary(res.lm)` sont identiques si tous les préditeurs sont continus ou avec deux catégories (car chaque effet a 1 ddl) et qu'il n'y a pas d'interaction dans le modèle.

4. `add1` teste l'ajout d'un ou plusieurs effets : `add1(res.lm, ~ habitat)`.

   Si le modèle initial n'inclue pas d'interaction : `add1(res, ~.^2)` testera l'addition de chaque interaction individuellement.

`plot(obj.lm)`

- valeurs prédites (en x) *vs.* résidus (en y)
- idem avec les résidus standardisés (variance homogène)
- quantiles des résidus standardisés en fonction des valeurs prédites
- distance de Cook :

$$D_i = \frac{\sum_j (\widehat{y}_j - \widehat{y}_{j(-i)})^2}{k\text{RSS}}$$

$\widehat{y}_j$: valeur prédite pour j avec toutes les données
$\widehat{y}_{j(-i)}$: valeur prédite pour j en enlevant i
k : nombre de paramètres
RSS : “residual sum of squares”

Autres fonctions utiles pour l’évaluation des modèles linéaires :

`termplot` (nombreuses options. . .) malheureusement, ne permet pas de visualiser les interactions

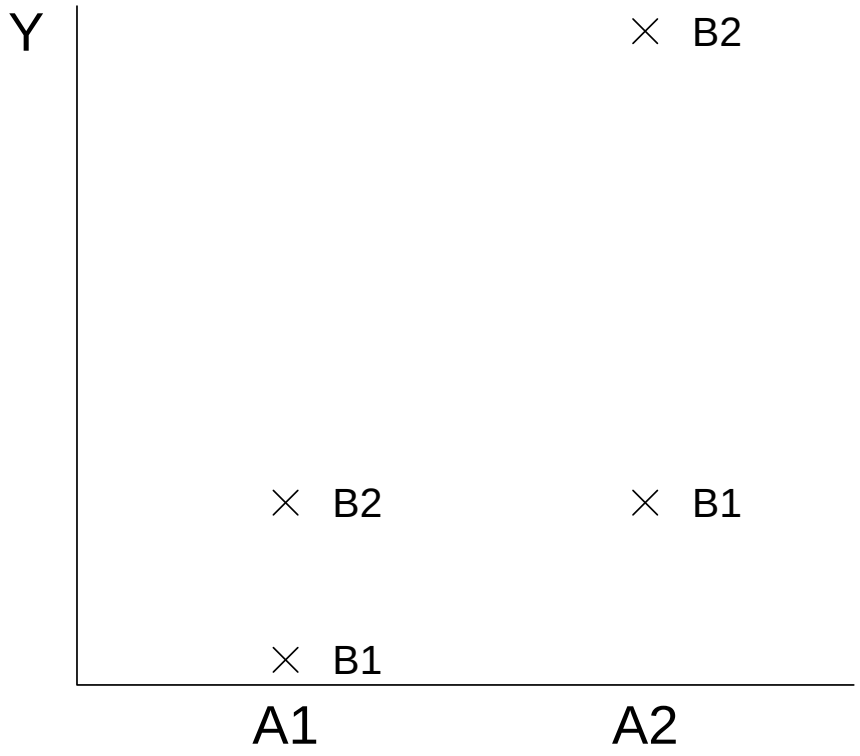
`interaction.plot`

1. `step` : cherche un modèle par sélection de variables pas-à-pas
2. `predict` : calcule les valeurs prédites par le modèle, éventuellement pour de nouvelles données ; possibiité de calculer les intervalles de confiance sur les prédictions (*cf.* `?predict.lm`)
3. `update` : ré-ajuste un modèle avec de nouveaux effets ou de nouvelles données

Mais où sont les “type III SS” dans R ?

“Nowhere, it seems to me, is this SAS *coup d’état* more evident than in the way Analysis of Variance concepts are handled” W. N. Venables

Les “type III SS” testent une hypothèse qui n’a, généralement, pas de sens.



Il ne peut pas y avoir d’interaction sans effets principaux, puisque, par définition, l’interaction implique un “contraste”. C’est le ***principe de marginalité***.

`y ~ x * A` avec $A = \{A_1, A_2\}$

$$E(y) = \beta_1 x + \beta_2 x_A + \beta_3 x x_A + \alpha$$

$$A_1 \rightarrow E(y) = \beta_1 x + \alpha$$
$$A_2 \rightarrow E(y) = (\beta_1 + \beta_3) x + \beta_2 + \alpha$$

$H_0 : \beta_1 = 0$ (hypothèse testée par les “type III SS”)

$$A_1 \rightarrow E(y) = \alpha$$
$$A_2 \rightarrow E(y) = \beta_3 x + \beta_2 + \alpha$$

`y ~ A * B` $A = \{A_1, A_2\}, B = \{B_1, B_2\}$

$$E(y) = \beta_1 x_A + \beta_2 x_B + \beta_3 x_A x_B + \alpha$$
$$A_1, B_1 \rightarrow E(y) = \alpha$$
$$A_1, B_2 \rightarrow E(y) = \beta_2 + \alpha$$
$$A_2, B_1 \rightarrow E(y) = \beta_1 + \alpha$$
$$A_2, B_2 \rightarrow E(y) = \beta_1 + \beta_2 + \beta_3 + \alpha$$

$$H_0 : \beta_1 = 0$$
$$A_1, B_1 \rightarrow E(y) = \alpha$$
$$A_1, B_2 \rightarrow E(y) = \beta_2 + \alpha$$
$$A_2, B_1 \rightarrow E(y) = \alpha$$
$$A_2, B_2 \rightarrow E(y) = \beta_2 + \beta_3 + \alpha$$

```
drop1(obj, test = "F")
drop1(obj, .~., test = "F")
library(car)
Anova(obj, type = "III")
```

Les “type II SS” respectent le principe de marginalité.

```
Anova(obj)
anova(aov(y ~ x1 * x2))
anova(aov(y ~ x2 * x1))
```

`model.matrix` peut être utile pour “vérifier” un modèle.

À lire : Venables W. N. (2000) *Exegeses on linear models.*
`www.stats.ox.ac.uk/pub/MASS3/Exegeses.pdf`

XI Miscellanea

Les options

Plusieurs options permettent de personnaliser l'apparence et le comportement de R. Elles sont affichées par `options()` et les options individuelles sont affichées avec, par exemple, `options("editor")` qui sera modifiée avec `options(editor = "emacs")`.

En voici quelques unes :

`prompt`	l'attente de commande, `"> "` par défaut
`warn = -1`	supprime les message d'avertissement
`width`	largeur de la console
`na.action`	le défaut pour les fonctions comme `lm`
`show.signif.stars`	(logique) affiche les étoiles associées aux test ? (`TRUE` par défaut)
`browser`	le navigateur appelé par `help.start()`
`editor`	l'éditeur appelé par `fix()` ou `edit()`

Personnaliser le démarrage de R

Le fichier '.Rprofile' situé dans le répertoire HOME de l'utilisateur contient des commandes R qui sont exécutées au démarrage. Exemple d'un tel fichier :

```
#options(width = 60)
options(editor = "emacs")
options(browser = "mozilla")
options(show.signif.stars = FALSE)
cat("Bon travail sur R !\n")
.Last <- function() cat("Penser à relire le papier !\n")
```

La fonction `.Last` est exécutée quand R est fermé.

Débuts de programmation sous R

La boucle `for`

```
for (x in A) {
    ........
    # expressions, commandes...
    ........
}
```

`A` peut être un vecteur (le plus souvent), un tableau ou une liste. À chaque itération de la boucle, `x` est remplacé par l'élément suivant de `A`, mais x peut ne pas être utilisé dans la boucle.

Si une seule instruction dans la boucle : pas besoin d'accolades.

```
FICHIER <- c("Aedes.dat", "Culex.dat", "Anopheles.dat")
for (FILE in FICHIER) {
    DF <- read.table(FILE, header = TRUE)
    mod <- lm(prevalence ~ ., data = DF)
    print(summary(mod))
}
```

⚠ Dans les boucles les résultats ne sont pas imprimés par défaut.

Exemple avec une liste :

```
selvar <- list(1:3, -6, c(1, 4:6, 8, 12))
for (sel in selvar)
  print(lm(prevalence ~ ., data = DF[, sel]))
```

De nombreuses variantes sont possibles à partir de cet exemple :

```
selvar <- list(1:3, -6, c(1, 4:6, 8, 12))
res <- list()
for (i in 1:length(selvar)) {
   sel <- selvar[[i]]
   res[[i]] <- lm(prevalence ~ ., data = DF[, sel])
   # variante à une ligne:
   # res[[i]] <- lm(prevalence ~ ., data = DF[, selvar[[i]]])
}
lapply(res, anova) # ou lapply(res, drop1) etc...
```

Souvent, il y a plusieurs objets concernés par l'itération :

```
FICHIER <- c("Aedes.dat", "Culex.dat", "Anopheles.dat")
SPECIES <- c("Aedes albopictus", "Culex pipiens",
             "Anopheles sp.")
pdf("Mosquito.pdf")
for (i in 1:length(FICHIER)) {
    DF <- read.table(FICHIER[i], header = TRUE)
    pairs(DF, main = SPECIES[i])
}
dev.off()
```

Très souvent efficacité et lisibilité vont ensemble, alors n'hésitez à faire appel à votre bon sens, votre intuition et votre bon goût (et votre imagination. . .).

Évitez les répétitions inutiles. . . :

☹

```
tab1[sp == "Aedes", ]
tab2[sp == "Aedes", ]
tab3[sp == "Aedes", ]
```

⇒

☺

```
sel <- sp == "Aedes"
tab1[sel, ]
tab2[sel, ]
tab3[sel, ]
```

. . . surtout si elles sont dans des boucles !

```
for (i in 1:ncol(DF)) {
  sel <- DF$sp == "Aedes"
  hist(DF[sel, i])
}
```

⇒

```
sel <- DF$sp == "Aedes"
DFsel <- DF[sel, ]
for (i in 1:ncol(DF))
  hist(DFsel[, i])
```

```
x1 <- seq(-1, 1, 0.001)
x2 <- seq(-1, 1, 0.001)
```

⇒

```
x2 <- x1 <- seq(-1, 1, 0.001)
```

ou

```
x1 <- seq(-1, 1, 0.001)
x2 <- x1
```

Les boucles peuvent être imbriquées :

```
for (x in SPECIES) {
    DFsel <- DF[sp == x, ]
    for (i in ncol(DF)) {
        hist(DFsel[, i], main = x)
    }
}
```

```
x <- rnorm(1e6)
s <- 0
for (i in 1:length(x)) s <- s + x[i]
```

$\longrightarrow$ temps de calcul = 3,1 sec

`sum(x)` $\longrightarrow$ temps de calcul = 0.004 sec

⚠ Pour bien programmer en R, ne pas programmer !

La boucle `if` : `else` est facultatif

```
for (i in 1:ncol(DF)) {
    x <- df[[i]]
    if (mode(x) == "numeric") {
        ...
    } # else message("colonne non numérique")
}
```

La boucle `while`

```
while (i < n) {
    ...
    i <- i +1
    ...
}
```

Ré-échantillonnage

`sample(x)` : une permutation aléatoire des éléments de `x`.
`x` : un vecteur, un tableau ou une liste.

`sample(x, replace = TRUE)` : échantillon de “bootstrap”

Pour `n` une valeur entière unique :
`sample(n)` : une permutation du vecteur `1:n`.

Si `X` est une matrice ou un tableau :

```
n <- nrow(X)
o <- sample(n)
X[o, ] # permutation aléatoire des lignes de X
n <- ncol(X)
o <- sample(n)
X[, o] # permutation aléatoire des colonnes de X
```

Trois entiers au hasard entre 1 et 8 (sans remise) :

```
> sample(8, size = 3)
[1] 5 1 6
> sample(8, size = 9)
Error in sample(8, size = 9) :
  cannot take a sample larger than the population when
      'replace = FALSE'
> sample(8, size = 9, replace = TRUE)
[1] 5 1 3 7 7 3 8 8 6
```

Un bootstrap :

```
B <- 1000
res <- numeric(B)
for (i in 1:B) {
    # éventuellement x <- DF[, 6]
    res[i] <- mean(sample(x, replace = TRUE))
}
hist(res)
abline(v = mean(x), col = "blue")
sd(res) # à vérifier...
```

à essayer avec `x <- runif(10)`

```
> sd(res) # à vérifier...
[1] 0.07923807
> mean(x)
[1] 0.3255205
```

Un test par permutations aléatoires (H_0 : les moyennes entre deux groupes sont égales, H_1 : les moyennes sont différentes) :

```
x <- runif(10, 0, 2)
y <- runif(10)
vec <- c(x, y)
g <- gl(2, 10)
ms <- tapply(vec, g, mean)
ms[2] - ms[1] # == diff(ms)

B <- 1000
stat <- numeric(B)
for (i in 1:B)
  stat[i] <- diff(tapply(sample(vec), g, mean))
hist(stat)
abline(v = diff(ms), col = "blue")
2*sum(stat > abs(ms[2] - ms[1]))/B # = P-value...
         # à ajuster si H1 est différente
```

Exemple :

```
> ms[2] - ms[1] # == diff(ms)
         2
-0.1421764
> 2*sum(stat > abs(ms[2] - ms[1]))/B # = P-value
[1] 0.432
> t.test(x, y)

        Welch Two Sample t-test

data:  x and y
t = 0.7864, df = 13.069, p-value = 0.4457
alternative hypothesis: true difference in means is not equal to
95 percent confidence interval:
 -0.2481968  0.5325497
sample estimates:
mean of x mean of y
0.7682938 0.6261174
```

Les dates

`as.Date` : transforme une chaîne de caractères en objet “Date”.

```
> x <- "2007-10-17"
> d <- as.Date(x, "%Y-%m-%d")
> x; d
[1] "2007-10-17"
[1] "2007-10-17"
> str(x)
 chr "2007-10-17"
> str(d)
Class 'Date'  num 13803
> b <- as.Date("17/10/07", "%d/%m/%y")
> b
[1] "2007-10-17"
> str(b)
Class 'Date'  num 13803
```

à essayer avec :

```
x <- c("14071789", "17102007")
x <- as.Date(x, "%d%m%Y")
plot(x, c(10, 100)) # != plot(x)
```

La syntaxe résumée :

`%d`	jour du mois (01–31)
`%m`	mois (01–12)
`%Y`	année (4 chiffres)
`%y`	année (2 chiffres) à éviter !
`%a`	nom du jour abrégé*
`%A`	nom du jour complet*
`%b`	nom du mois abrégé*
`%B`	nom du mois complet*

*système-dépendant (*locale*) mais “partial matching”

Les éléments manquants dans l'entrée sont pris dans l'horloge de la machine :

```
> as.Date("", "")
[1] "2007-10-17"
> as.Date("", "") == as.Date("17", "%d")
[1] TRUE
```

Plus compliqué pour le temps, mais l'idée se généralise :

```
> d1 <- strptime("17-10-2007 13:30:00", "%d-%m-%Y %H:%M:%S")
> d2 <- strptime("17-10-2007 13:30:30", "%d-%m-%Y %H:%M:%S")
> d2 - d1
Time difference of 30 secs
```

Pour l'opération inverse :

```
> str(x)
Class 'Date'  num [1:2] -65914  13803
> format(x, "%A %d %B %Y")
[1] "Tuesday 14 July 1789"      "Wednesday 17 October 2007"
```

Plus de détails sous R :

```
?strptime # détails sur syntaxe...
?Date
?as.Date
```