

Introduction au logiciel d'analyse statistique R

Denis Puthier

26 mai 2009

Laboratoire INSERM TAGC/ERM206, Parc Scientifique de Luminy case 928,13288
MARSEILLE cedex 09, FRANCE.
<http://tagc.univ-mrs.fr/puthier>
26 mai 2009

Table des matières

1	Préambules	4
2	Quelques mots à propos de R et des éditeurs	5
3	Premiers pas	5
3.1	Création, interrogation et destruction des objets.	5
3.2	Fonctions. Informations de base sur les objets	6
3.3	Les modes	8
3.4	L'aide dans R	8
3.4.1	La fonction <i>help</i>	8
3.4.2	La fonction <i>help.search</i>	9
3.4.3	La fonction <i>apropos</i>	9
3.4.4	La fonction <i>help.start</i>	10
4	Les objets dans R	10
4.1	Les objets de type vecteur	10
4.1.1	Création de vecteurs	10
4.1.2	Opérations de tri et de randomisation sur les vecteurs	12
4.1.3	Indexation des vecteurs	12
4.1.4	Operation mathématique sur les vecteurs	14
4.2	Les facteurs	15
4.3	Les matrices.	16
4.3.1	Création des matrices	16
4.3.2	Indexation des matrices	18

4.3.3	Trier les colonnes d'une matrice	19
4.4	Les listes	20
4.4.1	Création des listes	20
4.4.2	Indexation des listes	20
4.5	L'objet <i>data.frame</i>	21
4.5.1	Création des objets <i>data.frame</i>	22
4.5.2	Indexation des objets <i>data.frame</i>	22
5	Boucles for, vectorisation et fonctions de la famille apply	23
5.1	Les boucles for	23
5.2	La vectorisation	24
5.3	Fonctions de la famille <i>apply</i>	24
5.4	La fonction <i>apply</i>	24
5.4.1	Syntaxe de la fonction apply	25
5.4.2	Cas où la fonction à appliquer comporte plusieurs arguments	25
5.5	La fonction <i>tapply</i>	25
5.6	La fonction <i>lapply</i>	26
5.6.1	Syntaxe de la fonction <i>lapply</i>	26
5.6.2	Exemples d'utilisation de <i>lapply</i>	26
5.7	Autres fonctions de la famille <i>apply</i>	27
6	Gestion des valeurs manquantes	27
7	Structures conditionnelles	28
8	Fonctions de conversion	28
9	Les graphiques avec R	29
9.1	Gestion des fenêtres graphiques	29
9.2	Quelques exemples de fonctions graphiques	29
9.3	Paramètres graphiques	29
9.3.1	Quelques mots sur la fonction <i>par</i>	29
9.3.2	Gestion des couleurs	32
9.4	Un exemple de session graphique	34
10	Gestion des fichiers, lecture et écriture	36
10.1	Fonction <i>getwd</i>	36
10.2	Fonction <i>dir</i>	36
10.3	Fonction <i>setwd</i>	36
10.4	Fonction <i>dir.create</i>	38
10.5	Fonction <i>file.create</i>	38
10.6	Fonction <i>scan</i>	38

10.7	Fonction <code>read.table</code>	38
10.8	Fonctions <code>write.table</code>	39
11	Remarques à propos des objets dans R	39
11.1	Présentation	39
11.2	implantation d'une classe spécifique avec le formalisme "S4"	41
12	Installation d'une librairie dans R	43
13	Le fichier Rprofile	44

1 Préambules

L'avènement de la technologie des microarrays et de manière plus générale des techniques d'analyse biologique "hauts débits" (Librairies d'ESTs, SAGE, CAGE, Chip-chip...), conduit à la génération de données toujours plus volumineuses¹. Il est donc devenu courant pour le biologiste de manipuler quelques milliers (voire millions) de valeurs. Il convient donc de pouvoir utiliser des logiciels flexibles permettant l'automatisation des procédures de lecture, de traitement et d'analyse statistique des données.

R² est un système d'analyse statistique dont le développement a été initié par Robert Gentleman et Ross Ihaka. R fonctionne sur de nombreuses plate-formes (*Unix, Linux, Windows, Macintosh*). C'est un logiciel libre (sous licence *GPL*), développé dans le cadre du projet *GNU* (projet initié en 1984 visant à créer un système gratuit ressemblant à *UNIX*, <http://www.gnu.org/>). La notion de liberté sous-entend la liberté :

- D'exécuter le programme, pour tous les usages
- D'étudier son fonctionnement et de l'adapter à ses besoins (pour ceci l'accès au code source est une condition requise)
- De redistribuer des copies

Cette notion de liberté d'utilisation suggère que ce logiciel est en perpétuelle évolution notamment grâce à une communauté active de développeurs qui rendent publiques leurs améliorations. Sachant que ce logiciel est utilisé par des scientifiques d'horizons très divers (statisticiens, mathématiciens, biologistes, géographes...) cela permet un partage des outils développés et une convergence des disciplines vers des projets communs. C'est sans doute cela qui fait toute la force et le succès de cet outil. R est très souple et relativement facile d'utilisation. Au contraire des langages compilés (C/C++, Fortran, Pascal...) qui nécessitent la description d'un programme dans son ensemble pour son utilisation, R est un langage interprété, c'est à dire que les lignes de commandes sont exécutées directement par la machine. On pourra à la fois écrire des instructions simples, créer ses propres fonctions ou encore développer ses propres objets et librairies. Cependant on peut noter que son approche résolument tournée vers l'objet (au sens informatique) est parfois déroutante pour l'utilisateur néophyte et qu'une certaine expérience est nécessaire pour bien intégrer son fonctionnement.

Néanmoins l'effort sera rapidement récompensé. En effet, de très nombreux outils sont disponibles sous forme de librairies ("packages") qui contiennent de nombreuses fonctions de haut niveau dédiées à des tâches spécifiques. Le biologiste pourra s'appuyer sur la richesse de ce langage pour effectuer l'analyse statistique de données issues de méthodes biologiques "classiques" ou de méthodes "haut débits" (transcriptome, protéome, interactome,...).

Enfin, le logiciel R est doté d'une palette d'outils graphiques très évolués permettant d'obtenir des sorties totalement paramétrables. Ainsi, Les données analysées peuvent être exportées sous des formats divers (.jpg, .png, .bmp, .ps, .pdf,.html, .txt...).

¹**kpdf**. Si le chargement des images s'avère difficile avec *acrobat reader* vous pouvez utiliser *kpdf*.

²**Sweave**. Ce document a été généré avec la fonction **Sweave** issue de la librairie *tools* du logiciel R.

2 Quelques mots à propos de R et des éditeurs

Pour lancer R sous *Windows*, cliquez, sur le raccourci présent sur le bureau. Sous *Linux*, tapez “R” dans un terminal.

Il est souvent préférable de garder une trace des commandes passées à R. Si vous vous trompez ou n’êtes pas complètement satisfait d’un résultat, il sera très facile de relancer le script après avoir modifié quelques lignes ou paramètres. A moins que vous ne maîtrisiez parfaitement les macro commandes, cette opération n’est pas possible dans un tableur (Excel, Oocalc...). En effet, ceux-ci ne conservent pas de traces des opérations effectuées (hormis dans le cas de “macros”) et il est donc souvent très difficile de revenir, *a posteriori*, sur une analyse. Pour conserver et modifier les commandes, nous passerons par un éditeur. Sous windows, il en existe peu qui soient évolués et gratuits... Vous pourrez utiliser *notepad*, *notepad++* (en choisissant la coloration syntaxique du *C/C++*) ou encore *winEdt* (payant mais utilisable avec la librairie *RWinEdt*). Sous *linux*, on aura le choix entre *kwwrite*, *kate*, *rkward* et *emacs* (avec le module *ESS*) pour citer certains des plus utilisés. Sous *linux* les fichiers portant l’extension “.R” seront reconnus par ces éditeurs comme du langage *R* et ils adopteront la coloration syntaxique *ad hoc*. Enfin citons un éditeur, JGR, basé sur *JAVA* (et donc indépendant du système d’exploitation) que vous pourrez télécharger à l’adresse <http://stats.math.uni-augsburg.de/JGR/>. Dans le cadre de ces TP nous utiliserons *kate*.

3 Premiers pas

3.1 Création, interrogation et destruction des objets.

R est un langage basé sur des objets (*vector*, *matrix*, *data.frame*, *list*, *factor*,...). Nous reviendrons sur cette notion dont la compréhension est nécessaire pour la bonne maîtrise de ce langage. Pour commencer, créons un objet simple de type “vecteur” qui contiendra une valeur numérique.

```
> x <- 15
> x
```

```
[1] 15
```

L’opérateur d’assignation “=” peut se substituer à l’opérateur “<-”

```
> x = 22
> x
```

```
[1] 22
```

Cependant l'opérateur “<-” est généralement préféré car il permet d'avoir un code plus lisible (pour des raisons que nous expliciterons par la suite).

Le code pourra être commenté à l'aide du caractère “#”. Toute commande à la suite de ce caractère ne sera pas interprétée.

```
> #x <- 23
```

```
> x
```

```
[1] 22
```

Les instructions peuvent être séparées par un retour à la ligne ou par le caractère “;”.

```
> x <- 12; y <- 13
```

Les objets créés sont stockés dans la mémoire vive de l'ordinateur (ils ne sont pas, à ce stade, stockés sur le disque). On peut lister les objets disponibles en mémoire avec la fonction `ls`. Si vous quittez R ces objets seront détruits. Une autre méthode pour les détruire consiste à utiliser la fonction `rm` (**r**emove).

```
> ls()
```

```
[1] "x" "y"
```

```
> rm(x)
```

```
> rm(y)
```

```
> ls()
```

```
character(0)
```

3.2 Fonctions. Informations de base sur les objets

Nous venons de construire des vecteurs (“vector” pour R) contenant des données numériques. Nous avons, par ailleurs utilisé les fonctions `ls` et `rm`. Dans R, on peut appeler des fonctions pour interroger des objets et réaliser des actions à partir de ceux-ci. Les fonctions se présentent sous la forme suivante :

```
- nomdelafonction(arg1= a, arg2= b,...)
```

Arg1 et arg2 (...) correspondent au noms des arguments tandis que a et b correspondent aux objets que l'on souhaite passer à la fonction pour qu'elle effectue une tâche.

Dans l'exemple suivant on utilise la fonction `mean` pour calculer une moyenne puis une moyenne tronquée (on élimine 10% des valeurs faibles et 10% des valeurs fortes, soit 20%). Notez que les noms des l'arguments peuvent être abrégés voire omis. Dans ce dernier cas il faudra veiller à passer les arguments dans l'ordre (voir la section 3.4.1 pour connaître tous les arguments de la fonction `mean`)

Pour des informations sur la fonction `c` (pour `combine`) et l'opérateur "`:`", reportez vous à la section 4.1.1.

```
> y <- c(-50, 1:10, 700)
> y

 [1] -50  1  2  3  4  5  6  7  8  9 10 700

> is(y)

 [1] "numeric" "vector"

> length(y)

 [1] 12

> mean(y)

 [1] 58.75

> mean(y, trim = 0.2)

 [1] 5.5

> mean(y, tr = 0.2)

 [1] 5.5

> mean(y, 0.2)

 [1] 5.5
```

3.3 Les modes

Il existe quatre types de modes : "numérique", "caractère", "logique", "complexe". On peut stocker ces variables dans des vecteurs en utilisant la fonction `c` (combine) . **Attention, un vecteur n'accepte qu'un seul type de mode.**

```
> noms = c("celine", "alain", "robert")
> noms
```

```
[1] "celine" "alain" "robert"
```

```
> is(noms)
```

```
[1] "character" "vector"
```

```
> length(noms)
```

```
[1] 3
```

```
> logic = c(TRUE, FALSE, TRUE)
> logic
```

```
[1] TRUE FALSE TRUE
```

```
> is(logic)
```

```
[1] "logical" "vector"
```

Les variables logiques peuvent s'écrire sous la forme TRUE et FALSE ou sous la forme T et F.

```
> logic = c(T, F, T)
```

3.4 L'aide dans R

3.4.1 La fonction *help*

Elle peut aussi être appelée à l'aide du caractère "?". Elle permet d'appeler l'aide sur une fonction donnée. Parmi ses arguments, "html" permet de présenter le fichier d'aide dans le navigateur internet défini par défaut.

```
> help(mean)
```

```
> help(mean,html=TRUE)
```

Les différents champs de l'aide sont notamment :

- Description : renseigne sur le rôle de cette fonction.
- Usage : décrit la fonction et ses paramètres par défaut.
- Arguments : liste les arguments à passer à cette fonction et leurs types. On peut aussi accéder aux arguments à l'aide de la fonction `args`
- Value : le type de(s) l'objet(s) retourné(s) par la fonction.
- See Also : autres fonctions assez similaires ou complémentaires susceptible d'intéresser l'utilisateur.
- Exemples : quelques exemples d'utilisation de cette fonction. On peut accéder aux exemples via la fonction `example`.

3.4.2 La fonction `help.search`

Elle permet de chercher un terme donné dans les fichiers d'aide des fonctions.

```
> help.search("mean")
```

Vous remarquerez qu'ici l'argument est entre guillemets. L'argument est une chaîne de caractères (n'importe laquelle fera l'affaire) et non pas une fonction ou un objet de R comme avec la fonction `help`.

3.4.3 La fonction `apropos`

Une fonction terriblement pratique qui renvoie l'ensemble des fonctions dont le nom contient la chaîne de caractères recherchée. Souvent, les fonctions de base dans les langages de programmation ont des noms assez similaires, cela aide beaucoup. Attention cette fonction est sensible à la casse.

```
> apropos("file")
```

```
[1] "profile"          "download.file"    "file.edit"        "bzfile"
[5] "file"             "file.access"      "file.append"      "file.choose"
[9] "file.copy"        "file.create"      "file.exists"      "file.info"
[13] "file.path"        "file.remove"      "file.rename"      "file.show"
[17] "file.symlink"     "gzfile"           "list.files"       "memory.profile"
[21] "system.file"      "tempfile"         "zip.file.extract"
```

Avec une expression régulière très simple :

```
> apropos("^file")
```

```
[1] "file.edit"      "file"             "file.access"      "file.append"      "file.choose"
[6] "file.copy"      "file.create"      "file.exists"      "file.info"        "file.path"
[11] "file.remove"    "file.rename"      "file.show"        "file.symlink"
```

3.4.4 La fonction *help.start*

Elle donne accès à l'ensemble de l'aide sous forme de fichier HTML.

```
> help.start()
```

Après avoir lancé la fonction `help.start`, le navigateur internet charge des fichiers d'aide au format HTML. Ouvrez le lien vers "Packages". Ici sont répertoriés l'ensemble des librairies de fonctions R disponibles sur votre machine. Certaines librairies permettent l'analyse de microarrays comme : *affy*, *marray* et *sma*. Si vous ouvrez le lien vers la librairie *affy*, vous obtiendrez l'ensemble des fonctions contenues dans ces librairies et les rubriques d'aide associées.

4 Les objets dans R

Nous traiterons dans cette section des objets de type vector, factor matrix, list et data.frame. R est particulièrement doué dans la gestion des valeurs numériques qui se présenteront le plus fréquemment sous la forme de vecteurs (une dimension) ou matrice (deux dimensions) de données.

4.1 Les objets de type vecteur

4.1.1 Création de vecteurs

Les fonctions `c`, `rep` et `seq` Comme nous l'avons vu, R peut stocker des données dans des objets de type vecteur. On peut créer des vecteurs à l'aide des fonctions `c` (pour **combine**), `rep` (pour **repeat**), `seq` (pour **sequence**) ou de l'opérateur `:`. Comme nous l'avons souligné dans la section 3.3 le vecteur n'accepte qu'un type de mode.

```
> x <- 1:10
> x

[1] 1 2 3 4 5 6 7 8 9 10
```

```
> x <- c(2, 5, 7)
> x
```

```
[1] 2 5 7
```

```
> y <- rep(1, times = 5)
> y
```

```
[1] 1 1 1 1 1
```

```

> y <- rep(x, times = 3)
> y

[1] 2 5 7 2 5 7 2 5 7

> x <- seq(from = 1, to = 20, by = 2)
> x

[1] 1 3 5 7 9 11 13 15 17 19

> x <- seq(from = 1, to = 20, length = 10)
> x

[1] 1.000000 3.111111 5.222222 7.333333 9.444444 11.555556 13.666667
[8] 15.777778 17.888889 20.000000

```

Création de vecteurs contenant des variables pseudo-aléatoires Il est souvent très pratique en statistique de pouvoir générer des valeurs aléatoires. Les fonctions `rnorm`, `runif` et `rpois`, par exemple, permettent de générer des données aléatoires (`random`) selon une loi normale, uniforme ou de poisson, respectivement.

```

> x <- rnorm(10000, mean = 10, sd = 4)
> hist(x, col = "blue", breaks = 100)
> x <- runif(10000, min = 0, max = 10)
> hist(x, breaks = 20, col = "red")

```

Ici la fonction `hist` (*cf* ; la section sur les graphiques dans R, 9) est utilisée pour générer un histogramme de fréquences à partir de 10 000 valeurs aléatoires suivant une loi normale. Notez que l'argument `breaks` contrôle le nombre d'intervalles (axe des abscisses).

Création de vecteurs contenant des caractères alphabétiques R contient des variables pré-définies pour faciliter la vie de l'utilisateur. On peut les utiliser pour créer des vecteurs contenant des caractères alphabétiques :

```

> x <- letters[1:10]
> x

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

> x <- LETTERS[1:10]
> x

[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

```

4.1.2 Opérations de tri et de randomisation sur les vecteurs

On pourra faire des opérations simples de tri ou de randomisation sur les vecteurs. Notez que, dans l'exemple suivant, quand l'argument `replace` de la fonction `sample` est positionné sur `TRUE`, le tirage aléatoire est effectué avec remise.

```
> x <- 1:10
> sort(x, decreasing = T)

[1] 10  9  8  7  6  5  4  3  2  1

> sample(x)

[1]  8  2  9  3  4  5  6  7 10  1

> sample(x, replace = T)

[1]  5  2  5  4  8  8  4  6 10  5
```

4.1.3 Indexation des vecteurs

Pour interroger les éléments d'un vecteur on utilisera (i) un vecteurs d'indices (*ie*; de positions), ou un vecteur logique ou les noms des éléments du vecteur.

vecteur d'indices Il s'agit, dans ce cas, d'indiquer par une ou plusieurs valeur(s) numérique(s) les positions que l'on souhaite extraire dans le vecteur. On peut utiliser la fonction `which` qui renvoie les positions pour lesquels une condition est vérifiée.

```
> x <- c(-2, 4, -5, 1, 7)
> x[3]

[1] -5

> x[3:5]

[1] -5  1  7

> x[c(3, 5)]

[1] -5  7

> ind <- which(x > 0)
> ind
```

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

```
> x[ind]
```

```
[1] 4 1 7
```

On pourra aussi utiliser l'indexation de positionnement pour déléter les éléments d'un vecteur.

```
> x[-ind]
```

```
[1] -2 -5
```

Indexation par un vecteur logique Le vecteur logique est une suite de booléens (TRUE ou FALSE). On peut le créer très facilement en utilisant des opérateurs de comparaisons (`==`, `!=`, `>`, `<`, `<=`, `>=`) qui renvoie TRUE ou FALSE. On peut combiner ces comparaisons avec des opérateurs logiques (`&`, `|`).

```
> x > 0
```

```
[1] FALSE TRUE FALSE TRUE TRUE
```

```
> x[x > 0]
```

```
[1] 4 1 7
```

```
> x > 0 & x < 5
```

```
[1] FALSE TRUE FALSE TRUE FALSE
```

```
> x[x > 0 & x < 5]
```

```
[1] 4 1
```

Ici, seules les valeurs de `x` pour lesquels la condition est vraie sont renvoyés. Il s'agit en fait d'une boucle implicite. Cette syntaxe est extrêmement économique en terme de code et, par conséquent, elle est très employée dans le langage R.

```
> y <- 1:5
```

```
> cbind(x, y)
```

```

      x y
[1,] -2 1
[2,]  4 2
[3,] -5 3
[4,]  1 4
[5,]  7 5

> x[x > 0 & y > 3]

[1] 1 7

> x[x > 1 | y > 3]

[1] 4 1 7

```

Indexation par le nom des éléments du vecteur On peut associer un nom aux différents éléments d'un vecteur pour les identifier. Ces noms peuvent servir à l'indexation.

```

> names(x)

NULL

> names(x) <- letters[1:5]
> x

  a  b  c  d  e
-2  4 -5  1  7

> x["c"]

  c
-5

```

4.1.4 Operation mathématique sur les vecteurs

On peut effectuer des opérations mathématiques sur les vecteurs à l'aide d'opérateurs classiques : $+$, $-$, $*$. Pour élever à la puissance on utilise l'accent circonflexe. Ces opérations ne nécessitent pas d'utiliser des boucles "for" comme c'est le cas dans de nombreux langages et permettent donc d'effectuer en quelques lignes de code des opérations très complexes. C'est l'une des caractéristiques principales de R : la vectorisation. La structure vectorielle du langage rend les boucles implicites dans les expressions. Pour additionner deux vecteurs on écrira plutôt :

```

> x <- 2:5
> x

[1] 2 3 4 5

> y <- seq(1, 10, length = 4)
> y

[1] 1 4 7 10

> x + y

[1] 3 7 11 15

> x - y

[1] 1 -1 -3 -5

> x * y

[1] 2 12 28 50

> x^2

[1] 4 9 16 25

> x^0.5

[1] 1.414214 1.732051 2.000000 2.236068

```

4.2 Les facteurs

La fonction `factor` permet de créer des variables catégoriques à partir de vecteurs de modes caractères ou numériques. On pourra en effet, choisir d'analyser les individus ou les variables selon leur appartenance à une catégorie donnée. La fonction `levels` permet de connaître le nom des catégories que l'on pourra modifier comme dans l'exemple suivant.

```

> u <- sample(letters[1:3], size = 10, replace = T)
> u

[1] "b" "a" "c" "b" "b" "c" "a" "c" "a" "c"

> is(u)

```

```

[1] "character" "vector"

> u <- factor(u)
> levels(u)

[1] "a" "b" "c"

> levels(u) <- c("chat", "chien", "lapin")
> u

[1] chien chat  lapin chien chien lapin chat  lapin chat  lapin
Levels: chat chien lapin

```

NB : La fonction `gl` est aussi fréquemment utilisée pour générer des objets de type `factor`.

les facteurs constituent aussi un outil très efficace pour transformer des variables de types caractères en variables numériques via la fonction `as.numeric`.

```

as.numeric(u)

[1] 2 1 3 2 2 3 1 3 1 3 # Un vecteur (!= facteur)

```

4.3 Les matrices.

C'est simplement un tableau à 2 dimensions. C'est un cas particulier de la structure `array` qui accepte `k` dimensions.

4.3.1 Création des matrices

On produira une matrice avec la fonction `matrix`. **Tout comme le vecteur, une matrice n'accepte qu'un seul type de mode.** L'argument `data` de la fonction `matrix` est un vecteur qui indique ce que l'on souhaite mettre dans la matrice, l'argument `byrow` indique si on range les données en lignes ou en colonnes et les arguments `nrow` et `ncol` permettent d'indiquer le nombre de lignes et de colonnes respectivement.

Comme vous pouvez le constater dans l'exemple qui suit, cet objet contient des noms pour les lignes et des noms pour les colonnes.

```

> mat <- matrix(1:20, nc = 5, nr = 4, byrow = T)
> dim(mat)

[1] 4 5

```

```

> colnames(mat) <- LETTERS[1:5]
> row.names(mat) <- letters[1:4]
> mat

  A B C D E
a  1 2 3 4 5
b  6 7 8 9 10
c 11 12 13 14 15
d 16 17 18 19 20

```

Par ailleurs, on pourra agglomérer (*bind*) des lignes (**rows**) ou des colonnes avec les fonction `cbind` et `rbind` pour générer des matrices.

```

> x <- cbind(1:10, 21:30)
> is(x)

[1] "matrix"      "structure" "array"      "vector"     "vector"

> x

      [,1] [,2]
[1,]    1  21
[2,]    2  22
[3,]    3  23
[4,]    4  24
[5,]    5  25
[6,]    6  26
[7,]    7  27
[8,]    8  28
[9,]    9  29
[10,]  10  30

```

```

> x <- rbind(1:10, 21:30)
> is(x)

[1] "matrix"      "structure" "array"      "vector"     "vector"

> x

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]   21   22   23   24   25   26   27   28   29   30

```

4.3.2 Indexation des matrices

Pour l'indexation, on utilisera, comme dans le cas du vecteur, des vecteurs contenant des indices de positionnement ou des vecteurs logiques. Cependant, il faudra considérer une dimension supplémentaire. Le premier vecteur correspondra aux lignes, le deuxième aux colonnes.

```
> mat[1, ]
```

```
A B C D E
1 2 3 4 5
```

```
> mat[1:3, ]
```

```
  A B C D E
a  1 2 3 4 5
b  6 7 8 9 10
c 11 12 13 14 15
```

```
> mat[c(1, 3), ]
```

```
  A B C D E
a  1 2 3 4 5
c 11 12 13 14 15
```

```
> mat[-c(1, 3), ]
```

```
  A B C D E
b  6 7 8 9 10
d 16 17 18 19 20
```

```
> mat[, 1:3]
```

```
  A B C
a  1 2 3
b  6 7 8
c 11 12 13
d 16 17 18
```

```
> mat[1:3, 1:3]
```

```
  A B C
a  1 2 3
b  6 7 8
c 11 12 13
```

```
> ind <- mat[, 1] > 10
> ind
```

```
      a      b      c      d
FALSE FALSE TRUE  TRUE
```

```
> mat[ind, ]
```

```
      A B C D E
c 11 12 13 14 15
d 16 17 18 19 20
```

On pourra, si on le souhaite, indexer par une matrice logique et tester une condition très facilement sur l'ensemble des cellules de la matrice. Cette syntaxe est particulièrement rapide pour appliquer un seuil ou un plafond sur les valeurs d'une matrice.

```
> mat < 6
```

```
      A      B      C      D      E
a TRUE TRUE TRUE TRUE TRUE
b FALSE FALSE FALSE FALSE FALSE
c FALSE FALSE FALSE FALSE FALSE
d FALSE FALSE FALSE FALSE FALSE
```

```
> mat[mat < 6] <- 6
```

```
> mat
```

```
      A B C D E
a 6 6 6 6 6
b 6 7 8 9 10
c 11 12 13 14 15
d 16 17 18 19 20
```

4.3.3 Trier les colonnes d'une matrice

On utilisera la fonction `order` qui renvoie les indices de départ d'un vecteur trié.

```
> mat <- matrix(sample(1:20),nc=4)
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]   19    3    4   17    5
[2,]   15   10   13    7   16
[3,]    8    9    2   18   12
[4,]    6    1   14   11   20
```

```
mat[order(mat[,2]), ]

      [,1] [,2] [,3] [,4] [,5]
[1,]    6    1   14   11   20
[2,]   19    3    4   17    5
[3,]    8    9    2   18   12
[4,]   15   10   13    7   16
```

4.4 Les listes

L'objet `list`, est l'objet le plus flexible des objets basiques de R. Au contraire de la matrice, il ne possède pas de contraintes de tailles (dans la matrice toutes les lignes et toutes les colonnes ont la même taille par définition). De même, il ne contient pas de contraintes de modes et on pourra même y stocker des objets très divers. Etant donné cette flexibilité, les fonctions renvoient souvent les résultats sous cette forme (exemple : une liste contenant des vecteurs, des matrices et autres objets divers et variés issus du calcul). Chaque élément de la liste possède un nom. Ces noms sont accessibles via la fonction `names`.

4.4.1 Création des listes

```
> L1 <- list(A = 1:5, B = matrix(8:13, nr = 2), C = LETTERS[10:15])
> L1

$A
[1] 1 2 3 4 5

$B
      [,1] [,2] [,3]
[1,]    8   10   12
[2,]    9   11   13

$C
[1] "J" "K" "L" "M" "N" "O"

> names(L1)

[1] "A" "B" "C"
```

4.4.2 Indexation des listes

Attention, dans le cas des listes, la fonction d'indexation ("`[`") renvoie une liste, alors qu'on souhaiterait généralement avoir accès au contenu (dans l'exemple, un vecteur numérique).

```
> L1[1]
```

```
$A
```

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

```
> is(L1[1])
```

```
[1] "list" "vector"
```

Pour avoir accès au vecteur contenu dans la liste, il faut utiliser la fonction d'indexation spécifique des liste ("`[]`") et lui passer un numérique ou le nom d'un élément de la liste comme argument.

```
> L1[[1]]
```

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

```
> is(L1[[1]])
```

```
[1] "integer" "vector" "numeric"
```

```
> names(L1)
```

```
[1] "A" "B" "C"
```

```
> L1[["A"]]
```

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

On pourra, de plus utiliser l'opérateur `$` pour indexer la liste.

```
> L1$A
```

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

NB : On ne peut indexer qu'un seul élément de la liste à la fois. C'est là une limite de cet objet.

4.5 L'objet *data.frame*

Le `data.frame` est un tableau de données à deux dimensions composé d'un ou plusieurs vecteurs ayant tous la même longueur mais pouvant être de modes différents. C'est un intermédiaire entre la liste et la matrice. Comme la matrice il peut posséder des noms de lignes et des noms de colonnes accessibles par les fonctions `row.names` et `colnames`. Ses dimensions peuvent être interrogées par la fonction `dim` (comme dans le cas des objets `matrix`).

4.5.1 Création des objets *data.frame*

On pourra créer des objets `data.frame` en utilisant la fonction `data.frame`. Cette fonction prend comme arguments différents vecteurs qui constitueront les colonnes du `data.frame`. Attention, les vecteurs de caractères sont transformés par défaut en objet de type `factor`. On peut éviter cet effet en traitant les vecteurs de caractères à l'aide de la fonction `I` (voyez l'aide sur cette fonction si vous souhaitez en savoir plus).

```
> df <- data.frame(x = 10:1, y = 1:10, z = letters[10:19])
> df
```

```
   x y z
1 10 1 j
2  9 2 k
3  8 3 l
4  7 4 m
5  6 5 n
6  5 6 o
7  4 7 p
8  3 8 q
9  2 9 r
10 1 10 s
```

```
> is(df[, 3])
```

```
[1] "factor" "oldClass"
```

```
> dim(df)
```

```
[1] 10  3
```

4.5.2 Indexation des objets *data.frame*

A nouveau, on se situe entre la liste et la matrice. Pour l'indexation, on peut procéder (*i*) comme pour les matrices, (*ii*) se référer aux noms des colonnes en utilisant le caractère "\$" (*iii*) ou encore utiliser la fonction d'indexation ("[[") (*cf* : la section 4.4.2 sur l'indexation des listes).

```
> df[, 1]
```

```
[1] 10 9 8 7 6 5 4 3 2 1
```

```
> df$x
```

```
[1] 10 9 8 7 6 5 4 3 2 1
> df[["x"]]
[1] 10 9 8 7 6 5 4 3 2 1
```

5 Boucles for, vectorisation et fonctions de la famille apply

Les structures de répétition classiques sont implémentées dans R (`for`, `while`, `repeat`...). Nous traiterons ici des boucles `for` que l'utilisateur peut être amené rapidement à utiliser. L'une des limites des boucles `for` est que celles-ci nécessitent d'écrire du code sur plusieurs lignes ce qui va à l'encontre de la philosophie du langage R dans lequel l'utilisateur souhaite, en règle générale, avoir un résultat via une seule et unique instruction. Par ailleurs, étant donné la structure intrinsèque de R elles se révèlent souvent assez lentes. On ne les utilisera que quand aucune autre solution n'est disponible (assez rarement) ou lorsque leur utilisation permettra au code de gagner en lisibilité (si vous souhaitez partager vos scripts ou librairies).

5.1 Les boucles for

La syntaxe des boucles `for` est la suivante :

```
for( i in vec){ ... }
```

Dans cette structure, la variable `i` prendra successivement les valeurs contenues dans le vecteur `vec`. Considérons l'addition de deux vecteurs : Cette addition pourrait être écrite avec une boucle `for` comme cela se fait dans la plupart des langages. Notez que l'addition nécessite la création préalable d'un troisième vecteur pour stocker le résultat.

```
> x <- 1:10
> y <- -x
> z <- vector()
> for (i in 1:length(x)) {
+   z[i] <- x[i] + y[i]
+ }
> z

[1] 0 0 0 0 0 0 0 0 0 0
```

5.2 La vectorisation

Dans R, les boucles peuvent de manière générale être évitées, grâce à une caractéristique du langage R : la vectorisation. La structure vectorielle rend les boucles implicites dans les expressions. Pour additionner deux vecteurs on utilisera l'écriture suivante :

```
> z <- x + y
> z

[1] 0 0 0 0 0 0 0 0 0 0
```

NB : Cette solution est plus concise mais aussi plus rapide !

5.3 Fonctions de la famille *apply*

Toujours dans un but de concision, il existe des fonctions permettant d'effectuer des opérations sur l'ensemble des éléments d'un vecteur, d'une liste ou encore sur toutes les lignes ou colonnes d'une matrice ou d'un data.frame. Certaines des fonctions de la famille *apply* sont présentées dans cette section.

5.4 La fonction *apply*

La fonction `apply()` permet de réaliser des opérations sur les lignes ou les colonnes d'une matrice ou d'un data.frame. Elle permet de contourner l'utilisation de la boucle "for" qui comme dans l'exemple suivant (calcul des valeurs moyennes des colonnes d'une matrice) nécessite une syntaxe assez lourde.

```
> mat <- matrix(sample(20), ncol = 5)
> mat

      [,1] [,2] [,3] [,4] [,5]
[1,]    8    1   11   14   20
[2,]   13    2    7    5   16
[3,]   17   10   15    3   18
[4,]   12    6   19    4    9

> z <- vector()
> for (i in 1:ncol(mat)) {
+   z[i] <- mean(mat[, i])
+ }
> z

[1] 12.50  4.75 13.00  6.50 15.75
```

5.4.1 Syntaxe de la fonction `apply`

La syntaxe et les arguments de la fonction `apply` sont les suivants

```
apply(X, MARGIN, FUN, ...)
```

- X est une matrice ou un `data.frame`
- MARGIN indique si la fonction doit être appliquée sur les lignes (MARGIN=1) ou les colonnes (MARGIN=2)
- FUN est la fonction à appliquer
- ... des arguments supplémentaires pour FUN

la syntaxe pour le calcul des valeurs médianes de chacune des colonnes devient :

```
> apply(mat, 2, mean)
```

```
[1] 12.50  4.75 13.00  6.50 15.75
```

5.4.2 Cas où la fonction à appliquer comporte plusieurs arguments

Dans ce cas, le premier argument correspondra toujours à la nième colonne ou ligne de X (en fonction de MARGIN). Les autres arguments de la fonctions seront passés à la fin de la commande `apply` (argument "..."). Exemple pour calculer une moyenne tronquée.

```
> apply(mat, 2, mean, trim = 0.5)
```

```
[1] 12.5  4.0 13.0  4.5 17.0
```

5.5 La fonction `tapply`

Cette fonction permet d'effectuer des calculs sur les éléments d'un vecteur connaissant leur appartenance à des catégories.

```
> fac <- as.factor(sample(LETTERS[1:5], size = 20, replace = T))
```

```
> fac
```

```
[1] A E B D D B A E B E A C E E B B C E D A
```

```
Levels: A B C D E
```

```
> x <- runif(20, 0, 10)
```

```
> tapply(x, fac, sort)
```

```

$A
[1] 0.1422657 1.9218015 3.3894531 5.3228755

$B
[1] 0.5105884 3.2262803 4.7701483 5.2399521 6.6570511

$C
[1] 4.973641 9.694104

$D
[1] 4.526778 7.040531 7.704796

$E
[1] 0.4712470 0.7388998 1.4465574 4.1397842 7.9786786 9.3307780

```

5.6 La fonction *lapply*

Cette fonction est réservée aux objets `list` ou `data.frame`. Au contraire de `apply`, `lapply` elle ne possède pas l'argument `MARGIN`. Sur un `data.frame` le calcul se fera sur les colonnes.

5.6.1 Syntaxe de la fonction *lapply*

La syntaxe de cette fonction est la suivante :

```
lapply(X, FUN, ...)
```

- `X` est une liste ou un `data.frame`
- `FUN` est la fonction à appliquer
- ... des arguments supplémentaires pour `FUN`

5.6.2 Exemples d'utilisation de *lapply*

Un exemple avec la fonction `is`. L'application de cette fonction à chaque élément de la liste permet de connaître son type (sa "classe").

```

> L1 <- list(A = rnorm(10), B = c(T, F), C = mat)
> lapply(L1, is)

```

```

$A
[1] "numeric" "vector"

```

```

$B

```

```
[1] "logical" "vector"
```

```
$C
```

```
[1] "matrix"      "structure" "array"      "vector"     "vector"
```

5.7 Autres fonctions de la famille *apply*

Il existe d'autres fonctions sur le modèle de la fonction `apply` qui ne seront pas présentées dans ce tutoriel.

```
> apropos("apply")
```

```
[1] "dendrapply" "kernapply"  "apply"      "eapply"     "lapply"
[6] "mapply"     "sapply"     "tapply"
```

6 Gestion des valeurs manquantes

Dans R, les valeurs manquantes prennent pour valeur `NA` (Not Available).

```
> mat <- matrix(1:20, nc = 5)
> mat[mat < 3] <- NA
> mat
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]  NA   5   9  13  17
[2,]  NA   6  10  14  18
[3,]   3   7  11  15  19
[4,]   4   8  12  16  20
```

```
> apply(mat, 2, mean)
```

```
[1]  NA  6.5 10.5 14.5 18.5
```

Le résultat obtenu n'est pas celui attendu. Par défaut, la fonction `mean` renvoie `NA` si elle rencontre une valeur manquante dans un vecteur. Il faudra utiliser la syntaxe suivante :

```
> apply(mat, 2, mean, na.rm = T)
```

```
[1]  3.5  6.5 10.5 14.5 18.5
```

Dans cette dernière instruction l'argument `na.rm` (fréquemment rencontré) est passé à la fonction `mean` pour lui indiquer de ne pas considérer les valeurs manquantes (`na.remove`). A noter l'utilisation très pratique de `na.omit`. Cette fonction permet d'éliminer les éléments d'un vecteur ou les lignes d'une matrice qui contiennent des valeurs manquantes.

```
> na.omit(mat)

      [,1] [,2] [,3] [,4] [,5]
[1,]    3    7   11   15   19
[2,]    4    8   12   16   20
attr(,"na.action")
[1] 1 2
attr(,"class")
[1] "omit"
```

7 Structures conditionnelles

On retrouvera dans R les structures conditionnelles classiques. On peut citer, ici, la structure `if/else`.

```
> x <- TRUE
> if (x) {
+   cat("C'est vrai")
+ } else {
+   cat("C'est faux")
+ }
```

C'est vrai

le vecteur `x` est évalué par la structure de contrôle. S'il contient la valeur `TRUE` la première instruction est réalisée sinon c'est la deuxième.

8 Fonctions de conversion

Dans certain cas, on pourra convertir un objet en un autre. Les fonctions permettant cette action possèdent en générale le préfixe "`as.`". Comme vous pouvez le constater avec l'instruction suivante, elle sont très nombreuses.

```
> apropos("^as\\.")
```

On peut citer les fonctions :

- `as.factor` pour convertir en facteur.
- `as.data.frame` pour convertir en `data.frame`.
- `as.list` pour convertir en liste.
- `as.numeric` pour convertir en vecteur numérique.
- `as.character` pour convertir en vecteur de caractères.
- ...

Suivant le type de l'objet passé en argument, ces fonctions se comportent différemment et seule l'usage permet de les maîtriser. Citons pour finir la fonction `unlist` qui permet de transformer une liste en un vecteur.

9 Les graphiques avec R

R dispose de bibliothèques graphiques puissantes. Pour vous en convaincre, regardez les démonstrations associées aux bibliothèques *graphics*, *persp* ou encore *rgl*. Sous *Linux* Vous pouvez passer d'une fenêtre à une autre à l'aide du raccourci "Alt" + "tab".

```
> demo(graphics)
> demo(persp)
> library(rgl)
> demo(rgl)
```

La gestion des graphiques dans R sera rapidement abordée dans ce tutoriel. Vous pourrez vous reporter à la section 4 du tutoriel de E. Paradis ("Les graphiques avec R") pour de plus amples informations.

9.1 Gestion des fenêtres graphiques

La fonction `x11` vous permettra de créer une nouvelle fenêtre graphique. La fonction `graphics.off` permet de la détruire.

9.2 Quelques exemples de fonctions graphiques

Il existe, dans R, des fonctions graphiques primaires pour produire un graphique (`plot`, `hist`, `boxplot`, `image`, `matplot`...) et des fonctions graphiques secondaires permettant de modifier ou d'améliorer ce graphique (`title`, `legend`, `arrows`, `points`...). Quelques une de ces fonctions sont présentées en figure 1 et 2. De nombreuses autres fonctions sont disponibles dans des bibliothèques spécifiques (par exemple pour les microarrays)

<code>plot(x)</code>	graphe des valeurs de x (sur l'axe des y) ordonnées sur l'axe des x
<code>plot(x, y)</code>	graphe bivarié de x (sur l'axe des x) et y (sur l'axe des y)
<code>sunflowerplot(x, y)</code>	idem que <code>plot()</code> mais les points superposés sont dessinés en forme de fleurs dont le nombre de pétales représente le nombre de points
<code>pie(x)</code>	graphe en camembert
<code>boxplot(x)</code>	graphe boîtes et moustaches
<code>stripchart(x)</code>	graphe des valeurs de x sur une ligne (une alternative à <code>boxplot()</code> pour des petits échantillons)
<code>coplot(x~y z)</code>	graphe bivarié de x et y pour chaque valeur (ou intervalle de valeurs) de z
<code>interaction.plot(f1, f2, y)</code>	si $f1$ et $f2$ sont des facteurs, graphe des moyennes de y (sur l'axe des y) en fonction des valeurs de $f1$ (sur l'axe des x) et de $f2$ (différentes courbes); l'option <code>fun</code> permet de choisir la statistique résumée de y (par défaut <code>fun=mean</code>)
<code>matplot(x,y)</code>	graphe bivarié de la 1 ^{ère} colonne de x contre la 1 ^{ère} de y , la 2 ^{ème} de x contre la 2 ^{ème} de y , etc.
<code>dotchart(x)</code>	si x est un tableau de données, dessine un graphe de Cleveland (graphes superposés ligne par ligne et colonne par colonne)
<code>fourfoldplot(x)</code>	visualise, avec des quarts de cercles, l'association entre deux variables dichotomiques pour différentes populations (x doit être un tableau avec <code>dim=c(2, 2, k)</code> ou une matrice avec <code>dim=c(2, 2)</code> si $k = 1$)
<code>assocplot(x)</code>	graphe de Cohen–Friendly indiquant les déviations de l'hypothèse d'indépendance des lignes et des colonnes dans un tableau de contingence à deux dimensions
<code>mosaicplot(x)</code>	graphe en 'mosaïque' des résidus d'une régression log-linéaire sur une table de contingence
<code>pairs(x)</code>	si x est une matrice ou un tableau de données, dessine tous les graphes bivariés entre les colonnes de x
<code>plot.ts(x)</code>	si x est un objet de classe "ts", graphe de x en fonction du temps, x peut être multivarié mais les séries doivent avoir les mêmes fréquence et dates

FIG. 1 – Quelques fonctions graphiques primaires dans R. Pour une liste plus complète veuillez consulter le manuel de E. Paradis http://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf.

<code>points(x, y)</code>	ajoute des points (l'option <code>type=</code> peut être utilisée)
<code>lines(x, y)</code>	idem mais avec des lignes
<code>text(x, y, labels, ...)</code>	ajoute le texte spécifié par <code>labels</code> au coordonnées <code>(x,y)</code> ; un usage typique sera : <code>plot(x, y, type="n"); text(x, y, names)</code>
<code>mtext(text, side=3, line=0, ...)</code>	ajoute le texte spécifié par <code>text</code> dans la marge spécifiée par <code>side</code> (cf. <code>axis()</code> plus bas); <code>line</code> spécifie la ligne à partir du cadre de traçage
<code>segments(x0, y0, x1, y1)</code>	trace des lignes des points <code>(x0,y0)</code> aux points <code>(x1,y1)</code>
<code>arrows(x0, y0, x1, y1, angle=30, code=2)</code>	idem avec des flèches aux points <code>(x0,y0)</code> si <code>code=2</code> , aux points <code>(x1,y1)</code> si <code>code=1</code> , ou aux deux si <code>code=3</code> ; <code>angle</code> contrôle l'angle de la pointe par rapport à l'axe
<code>abline(a,b)</code>	trace une ligne de pente <code>b</code> et ordonnée à l'origine <code>a</code>
<code>abline(h=y)</code>	trace une ligne horizontale sur l'ordonnée <code>y</code>
<code>abline(v=x)</code>	trace une ligne verticale sur l'abscisse <code>x</code>
<code>abline(lm.obj)</code>	trace la droite de régression donnée par <code>lm.obj</code> (cf. section 5)
<code>rect(x1, y1, x2, y2)</code>	trace un rectangle délimité à gauche par <code>x1</code> , à droite par <code>x2</code> , en bas par <code>y1</code> et en haut par <code>y2</code>
<code>polygon(x, y)</code>	trace un polygone reliant les points dont les coordonnées sont données par <code>x</code> et <code>y</code>
<code>legend(x, y, legend)</code>	ajoute la légende au point de coordonnées <code>(x,y)</code> avec les symboles donnés par <code>legend</code>
<code>title()</code>	ajoute un titre et optionnellement un sous-titre
<code>axis(side, vect)</code>	ajoute un axe en bas (<code>side=1</code>), à gauche (2), en haut (3) ou à droite (4); <code>vect</code> (optionnel) indique les abscisses (ou ordonnées) où les graduations seront tracées
<code>box()</code>	ajoute un cadre autour du graphe
<code>rug(x)</code>	dessine les données <code>x</code> sur l'axe des <code>x</code> sous forme de petits traits verticaux
<code>locator(n, type="n", ...)</code>	retourne les coordonnées <code>(x, y)</code> après que l'utilisateur ait cliqué <code>n</code> fois sur le graphe avec la souris; également trace des symboles (<code>type="p"</code>) ou des lignes (<code>type="l"</code>) en fonction de paramètres graphiques optionnels (...); par défaut ne trace rien (<code>type="n"</code>)

FIG. 2 – Quelques fonctions graphiques secondaires dans R. Pour une liste plus complète veuillez consulter le manuel de E. Paradis http://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf.

9.3 Paramètres graphiques

9.3.1 Quelques mots sur la fonction *par*

En plus des fonctions graphiques secondaires, la présentation des graphiques peut être améliorée grâce aux paramètres graphiques gérés par la fonction `par` qui possède environ 80 arguments (couleur de fond, tailles et couleurs des polices, types et couleurs des axes...). Cependant, en règle générale, les fonctions primaires possèdent des arguments qui sont passés à la fonction `par`, ce qui évite à l'utilisateur d'appeler cette fonction. On utilisera plutôt la fonction `par` pour un paramétrage fin du graphique.

Il est possible de partitionner une fenêtre graphique (pour positionner plusieurs graphiques) à l'aide des fonctions `split.screen`, `layout` ou `par`. Dans l'exemple suivant, la fonction `par` est utilisée pour scinder la fenêtre graphique en 4 (2 lignes et 2 colonnes). La fonction `hist` (avec les paramètres `col='blue'` et `breaks=100`) est ensuite appliquée aux trois éléments d'une liste `x` (le résultat est présenté en figure 3).

```
> x <- list(rnorm(10000),runif(10000),rgamma(10000,2))
> par(mfrow=c(2,2))
> lapply(x, hist, col='blue',breaks=100)
> par(mfrow=c(1,1))
```

9.3.2 Gestion des couleurs

L'utilisateur dispose, par défaut, d'un certain nombre et d'un certain type de couleurs dans sa "palette". Les couleurs disponibles peuvent être appelées avec la fonction `palette` (8 couleurs par défaut). Dans un graphique, les couleurs, souvent contrôlées par l'argument `col`, peuvent être définies sous forme d'un vecteur d'entiers (ex : `1 :10`) ou d'un vecteur de chaînes de caractères.

Dans le cas d'un vecteur d'entiers, ses valeurs permettront d'indexer le vecteur renvoyé par la fonction `palette`. Ainsi, la valeur 1 correspondra à la première couleur de la palette et la valeur `n` à la `n`^{ième} valeur de la palette (voir l'exemple qui suit). Dans le cas d'une chaîne de caractère on mettra des noms de couleurs (l'ensemble des noms possibles est renvoyé par la fonction `color`) ou des valeurs hexadécimales (du type `#CC00CC`). Dans le cas de valeurs hexadécimales, les deux premiers caractères correspondent au rouge, les deux suivants au vert et les deux derniers au bleu (ex : noir=`'000000'`, blanc=`'FFFFFF'`, rouge=`'FF0000'`). Des jeux de couleurs au format hexadécimale peuvent être renvoyés par des fonctions : `rainbow`, `heat.colors`, `topo.colors`, `cm.colors`, `terrain.colors`...

```
> palette()

[1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"
[8] "gray"
```

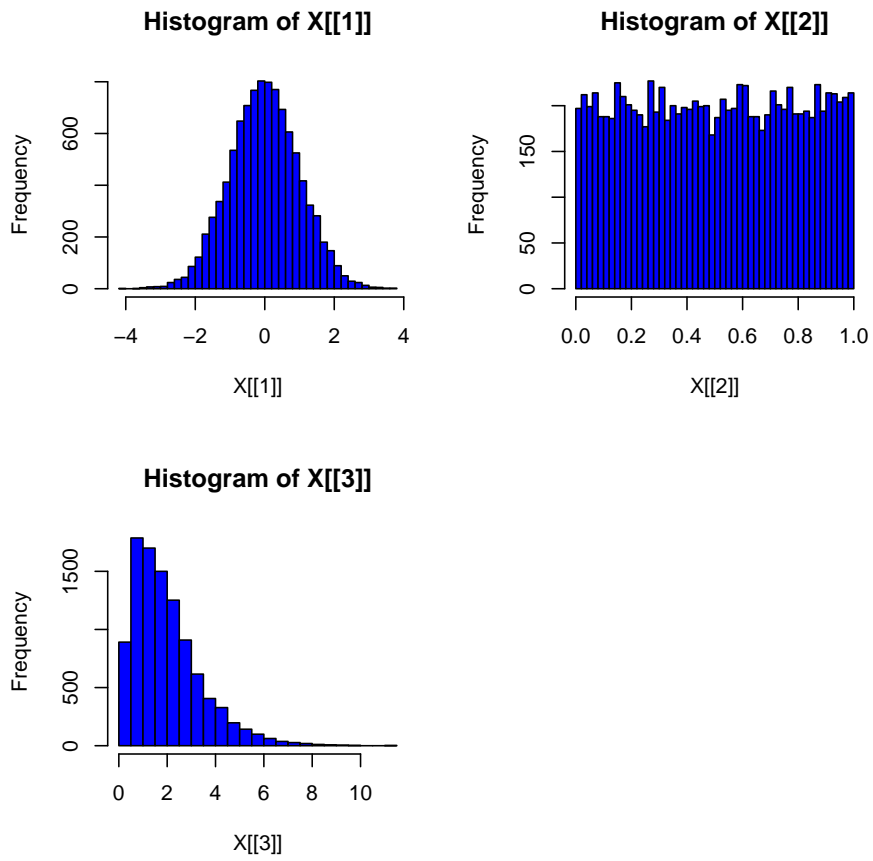


FIG. 3 – Histogrammes de fréquences générés avec des données issues d’une loi normale, uniforme et gamma. La fenêtre graphique a été préalablement partitionnée à l’aide de la fonction `par`.

```

> plot(21:28, 21:28, col = 1:8, pch = 16, cex = 10)
> rainbow(8)

[1] "#FF0000" "#FFBF00" "#80FF00" "#00FF40" "#00FFFF" "#0040FF" "#8000FF"
[8] "#FF00BF"

> plot(21:28, 21:28, col = rainbow(8), pch = 16, cex = 10)
> coul = colors()
> length(coul)

[1] 657

> coul[1:10]

[1] "white"          "aliceblue"      "antiquewhite"  "antiquewhite1"
[5] "antiquewhite2" "antiquewhite3" "antiquewhite4" "aquamarine"
[9] "aquamarine1"   "aquamarine2"

> plot(21:28, col = coul[1:8], pch = 16, cex = 10)

```

9.4 Un exemple de session graphique

Pour illustrer les fonctions graphiques, on peut utiliser un des nombreux jeux de données disponibles dans R via la fonction `data`. Ici on utilisera les données `airquality` qui sont issues de l'analyse de la qualité de l'air à New York entre mai et septembre 1973, en fonction de paramètres météorologiques. Un thème d'actualité... La fonction `pairs` permet de représenter les variables 2 à 2 (figure 4).

```

> data(airquality)
> is(airquality)
> air <- airquality
> names(air)
> help(air)
> air <- na.omit(air)
> pairs(air, panel = panel.smooth,
+ main = 'airquality', col='blue', pch='*')

```

On pourrait représenter quatre variables sur un même graphique. On utilisera l'axe des abscisses et des ordonnées pour représenter les variables Ozone et Wind, la taille des points pour représenter la variable Temperature et la couleur pour la variable Month (figure 5). La taille des points d'un graphique est généralement comprise entre 0.1 et 2. Pour ce mettre dans cette gamme et mieux visualiser l'effet des températures on effectue l'opération suivante.

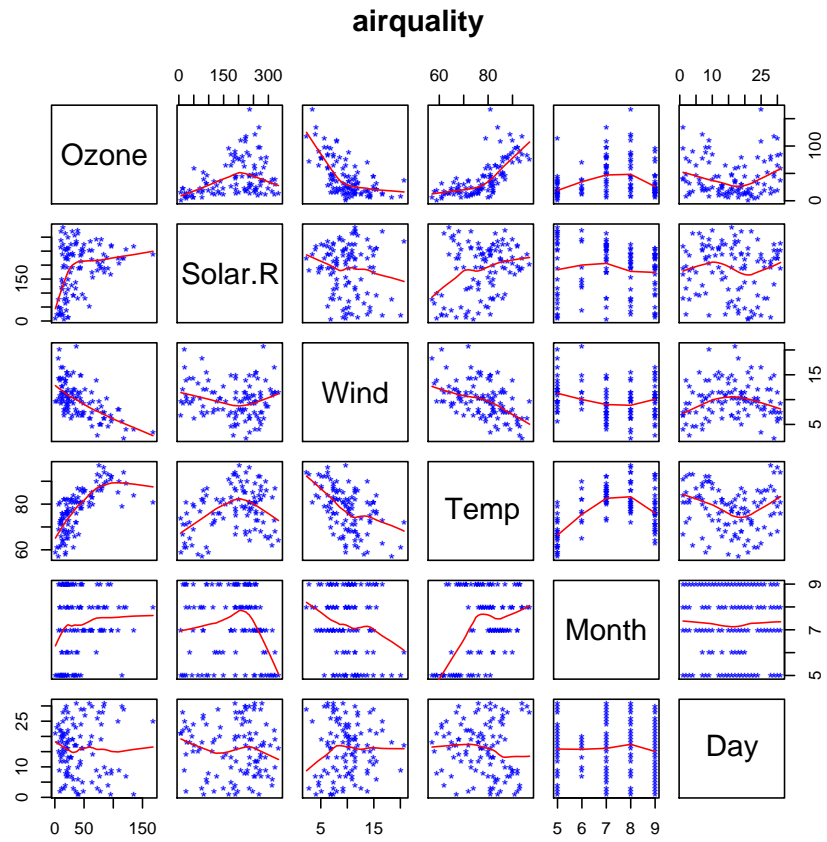


FIG. 4 – Représentation 2 à 2 des variables du jeu de données `airquality`. Une regression de type “lowess” est réalisée pour chaque couple.

```
> air$T <- (air$T - min(air$T) + 1)/10
```

On peut alors lancer le graphique après avoir changé la palette des couleurs.

```
> palette()
> palette(rainbow(5))
> palette()
> plot(air$O,air$W,cex=air$T,col=air$M-4, pch=16)
> text(air$O,air$W,lab=air$M,cex=0.5)
> title("Paramètres atmosphériques (New-York, 1973)")
> legend(130,20,legend=c('Mai', 'Juin', 'Juillet', 'Aout', 'Sept. '),fill=palette())
```

10 Gestion des fichiers, lecture et écriture

10.1 Fonction getwd

La fonction `getwd` permet d'afficher le répertoire de travail (**get working directory**).

```
> getwd()
```

10.2 Fonction dir

On pourra lister les fichiers dans un répertoire à l'aide de la fonction `dir()` (**directory**).

```
> dir()
```

La valeur renvoyée par `dir` est un vecteur de chaînes de caractères (vérifiez le) on peut donc utiliser l'indexation. Ce type de raccourci sera très fréquemment utilisé par la suite pour d'autres fonctions :

```
> dir()[1]
```

10.3 Fonction setwd

La fonction `setwd` permet de se déplacer dans l'arborescence.

```
> setwd("/home/login")# c'est un exemple
```

NB : `setwd` est très pratique au sein d'une fonction ou encore sous unix. Cependant, sous windows (dans lequel la complétion n'est pas réalisée) il est souvent plus pratique d'utiliser le menu :

File > Change dir>Browse>mon dossier

Paramètres atmosphériques (New-York, 1973)

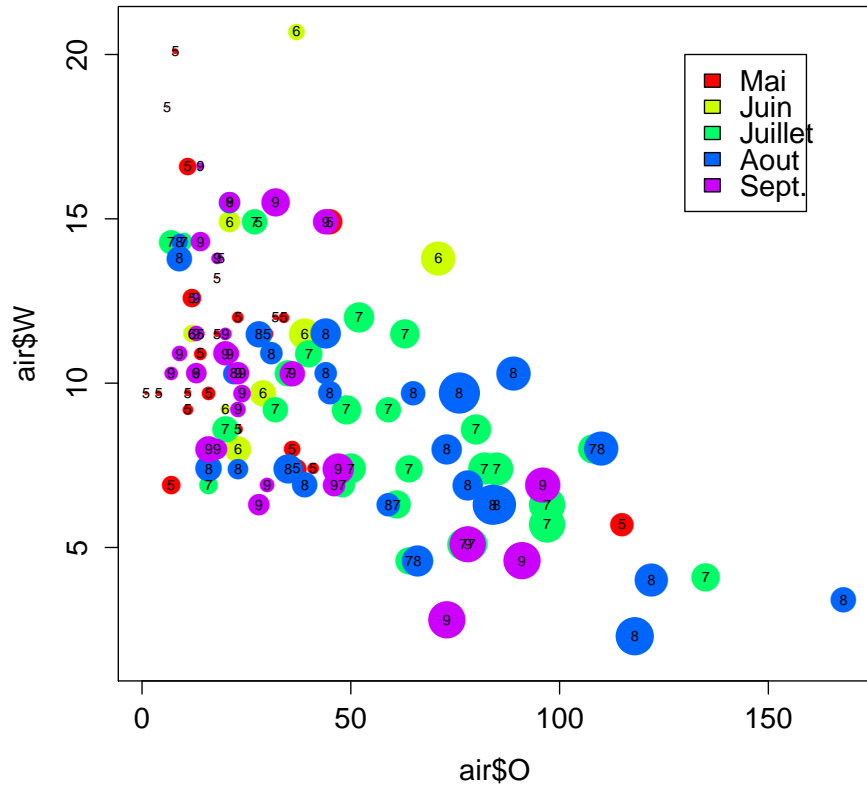


FIG. 5 – Paramètres atmosphériques et météorologiques dans la ville de New-York en 1973. La taille des points est proportionnelle à la température. Les couleurs et chiffres correspondent aux mois.

10.4 Fonction `dir.create`

La fonction `dir.create` permet de créer des répertoires. Exemple : Création d'un nouveau répertoire nommé "myDir". On s'y déplace ensuite avec `setwd`.

```
> dir.create("myDir")
> setwd("myDir")
```

10.5 Fonction `file.create`

R dispose de nombreuses fonction pour effectuer des tâches sur des fichiers :

- `file.create` : création de fichiers
- `file.show` : pour visualiser le contenu d'un fichier
- `cat` : écriture dans un fichier ou sur la sortie standard si l'argument ("file") n'est pas renseigné
- `file.remove` : détruire un fichier

```
> file.create("f1.txt")
> cat("bonjour", rnorm(10), file="f1.txt", sep="\n")
> file.show("f1.txt")
> file.remove("f1.txt")
```

10.6 Fonction `scan`

Elle permet le chargement de fichiers mais permettra aussi la lecture de données depuis la console lorsque l'argument "file" est omis. On pourra, par exemple, copier-coller une colonne depuis un tableur (ex : Excel ou OpenOffice.calc) sans avoir à passer par un fichier. Attention, ce mode de lecture est relativement lent.

```
> x <- scan(what="character")
1: alice
2: julien
> x <- scan(what="numeric")
1: 1
2: 30
```

10.7 Fonction `read.table`

Cette commande est incontournable puisqu'elle permettra la lecture de tableaux de données (Table 1). Cette fonction renvoie un objet de type `data.frame` et contient de nombreux arguments qui permettent de paramétrer la lecture des données. Tapez "?read.table" pour avoir accès à l'aide sur cette fonction.

<code>file</code>	le nom du fichier (entre <code>"</code>), éventuellement avec son chemin d'accès (le symbole <code>\</code> est interdit et doit être remplacé par <code>/</code> , même sous Windows)
<code>header</code>	une valeur logique (FALSE ou TRUE) indiquant si le fichier contient les noms des variables sur la 1 ^{ère} ligne
<code>sep</code>	le séparateur de champ dans le fichier, par exemple <code>sep="\t"</code> si c'est une tabulation
<code>quote</code>	les caractères utilisés pour citer les variables de mode caractère
<code>dec</code>	le caractère utilisé pour les décimales
<code>row.names</code>	un vecteur contenant les noms des lignes qui peut être un vecteur de mode caractère, ou le numéro (ou le nom) d'une variable du fichier (par défaut : 1, 2, 3, ...)
<code>col.names</code>	un vecteur contenant les noms des variables (par défaut : V1, V2, V3, ...)
<code>as.is</code>	contrôle la conversion des variables caractères en factor (si FALSE) ou les conserve en caractères (TRUE)
<code>na.strings</code>	indique la valeur des données manquantes (sera converti en NA)
<code>skip</code>	le nombre de lignes à sauter avant de commencer la lecture des données
<code>check.names</code>	si TRUE, vérifie que les noms des variables sont valides pour R
<code>strip.white</code>	(conditionnel à <code>sep</code>) si TRUE, <code>scan</code> efface les espaces (= blancs) avant et après les variables de mode caractère

TAB. 1 – Les arguments de la fonction `read.table`. D'après E. Paradis, R pour les débutants).

10.8 Fonctions `write.table`

Elle permet d'écrire sur le disque les données contenues dans une matrice ou un `data.frame` :

```
> m <- matrix(1:20,nc=10,nr=2)
> write.table(m,"UnFichier.txt",sep="\t", row.names=F,col.names=F)
> dir()
[1] "UnFichier.txt"
> d <- read.table("UnFichier.txt",sep="\t")
> all(d==m)
> [1] TRUE
```

11 Remarques à propos des objets dans R

11.1 Présentation

Nous avons parlé jusqu'ici de fonctions mais il serait plus juste d'un point de vue sémantique de parler de "méthodes". En effet, au même titre que dans des langages purement orientés vers l'objet, les fonctions, dans R sont capables de s'adapter aux objets qui leur sont passés en argument ("polymorphisme"). Ainsi, lorsque vous appelez la fonction `summary` (qui permet d'avoir un résumé des données contenues dans l'objet), celle-ci fonctionnera si vous lui passez des objets de types `vector`, `matrix`, `data.frame`, `factor`...

```

> x <- sample(rep(1:4, 5), replace = T)
> x
[1] 2 1 4 4 3 2 3 1 3 2 1 4 3 1 1 3 3 2 4 2

> summary(x)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 1.00  1.75   2.50   2.45   3.00   4.00

> summary(as.factor(x))
 1 2 3 4
 5 5 6 4

> summary(matrix(x, nc = 5))
      V1      V2      V3      V4      V5
Min.  :1.00  Min.  :1.00  Min.  :1.00  Min.   :1  Min.   :2.00
1st Qu.:1.75  1st Qu.:1.75  1st Qu.:1.75  1st Qu.:1  1st Qu.:2.00
Median :3.00  Median :2.50  Median :2.50  Median :2  Median :2.50
Mean   :2.75  Mean   :2.25  Mean   :2.50  Mean   :2  Mean   :2.75
3rd Qu.:4.00  3rd Qu.:3.00  3rd Qu.:3.25  3rd Qu.:3  3rd Qu.:3.25
Max.   :4.00  Max.   :3.00  Max.   :4.00  Max.   :3  Max.   :4.00

> apropos("^summary")
 [1] "summary"                "Summary"
 [3] "summary.aov"            "summary.aovlist"
 [5] "summary.connection"    "summary.data.frame"
 [7] "Summary.data.frame"    "summary.Date"
 [9] "Summary.Date"          "summary.default"
[11] "Summary.difftime"      "summary.factor"
[13] "Summary.factor"        "summary.glm"
[15] "summary.infl"          "summary.lm"
[17] "summary.manova"        "summary.matrix"

```

En fait il n'existe pas une fonction `summary`, mais de nombreuses fonctions (ou plutôt méthodes). Quand un objet particulier est rencontré, la fonction qui lui correspond est appelée. Si aucune méthode n'est définie pour cet objet, c'est la fonction `summary.default` qui est appelée. Si l'objet est différent de ce qu'attend la fonction `summary.default`, il y a une erreur.

Cette approche permet de minimiser le code puisqu'avec une seule fonction on traite des objets très différents sans se soucier de leurs types. Ce mode de fonctionnement est rencontré pour les objets de type "S3". Si on souhaite modifier la sortie on peut convertir l'objet avec une fonction de conversion ("as.", Cf section 8).

11.2 implantation d'une classe spécifique avec le formalisme "S4"

Le projet Bioconductor a permis de populariser la création de classes selon le formalisme "S4". Voilà un exemple permettant la création d'une classe "grpMicro" (*ie*, un groupe de microarrays) qui pourrait regrouper au sein d'une même instance des informations spécifiques d'une expérience de biopuces (deux couleurs).

```
> # grpMicro= un groupe de microarrays
> # L'argument "représentation" correspond à l'ensemble des
> # attributs de l'objet
> # L'argument "prototype" correspond aux valeurs des attributs par défaut

> setClass("grpMic",
  representation(
    rouge="matrix",
    vert="matrix",
    phenotype="matrix",
    genes="character",
    description="character"),
  prototype=list(
    rouge=matrix(nr=0,nc=0),
    vert=matrix(nr=0,nc=0),
    phenotype=matrix(nr=0,nc=0),
    description=new("character")
  )
)

> # on crée un nouvel objet de la classe grpMic ("une instance")
> monMA=new("grpMic")
> # on insert des données dans les champs "rouge" et "vert" de
> # l'objet
> monMA@vert <- matrix(rnorm(50),nc=5)
> monMA@rouge <- matrix(rnorm(50),nc=5)

# on appelle la méthode show
> monMA # show(monMA) donne le même résultat
> monMA
An object of class ■grpMic■
Slot "rouge":
      [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.2051713 -1.27056207 -1.1045189 0.3240014 -0.06463309
[2,] 0.6657903 0.29377438 -0.3388988 0.4325695 -1.19240689
```

```
[3,] 0.5738321 -0.99397729 0.7030940 -1.0716308 -0.06861840
[4,] -3.0179180 -0.12774912 -0.1188203 1.1507929 1.10492388
[5,] -0.7249525 0.65412414 -0.6214066 1.7374973 0.02248188
[6,] -0.7376729 -0.27760596 -0.1915933 -0.7967234 -0.02957149
[7,] -0.9649573 0.03892689 -0.8581441 0.2965948 0.91353441
[8,] 1.2965189 1.73985360 -0.3833713 0.6668347 0.14525068
[9,] 1.8408906 -0.48977407 -0.6991744 0.6230924 0.02083169
[10,] 0.8842606 1.67756627 0.4462402 -1.7597524 -0.57513841
```

Slot "vert":

```
      [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 1.7153920 -0.07490453 -2.2567381 -0.7233113 0.6853212
[2,] -0.2823390 -1.09078372 -0.5596569 0.2943607 -1.2814961
[3,] -0.4847466 0.75635278 -1.0578807 2.7280499 -1.1328181
[4,] -1.3752882 -0.60870596 -1.1568834 1.1593624 -0.3774755
[5,] 1.0904968 0.52434888 -1.1066795 -0.3900857 -1.3835309
[6,] -1.0861703 1.06045757 1.0821633 -0.2353755 0.6999585
[7,] 1.1772676 0.61244104 -0.6360943 1.9375349 0.2473462
[8,] 0.6399646 0.27892227 1.0645251 0.8460971 1.3907317
[9,] -1.1192042 0.97362055 0.5475345 1.0395862 -1.3767873
[10,] -1.7108041 -0.78232675 0.6504359 -0.1487441 1.4123549
```

Slot "phenotype":

<0 x 0 matrix>

Slot "genes":

character(0)

Slot "description":

character(0)

```
> # On re-défini la méthode "show" pour un objet de type "grpMic"
```

```
> setMethod("show", signature("grpMic"),
  function(object){
    cat("Une instance de la classe grpMic\n")
    cat("Nb échantillons=", ncol(object@rouge),"\n")
    cat("Nb de genes=", nrow(object@rouge),"\n")
  })
```

```
> # on appelle la méthode show
```

```

> monMA # show(monMA) donne le même résultat
Une instance de la classe grpMic
Nb échantillons= 5
Nb de genes= 10

```

NB : Notez que l'on pourra redéfinir la fonction d'indexation "[]".

12 Installation d'une librairie dans R

Les bibliothèques contiennent un ensemble de fonctions développées pour réaliser des tâches spécifiques. Pour connaître les bibliothèques disponibles sur votre machine utilisez la fonction `library`.

```

> library()
Packages dans la bibliothèque '/usr/lib/R/library' :

adapt          adapt -- multidimensional numerical integration
ade4           Analysis of Ecological Data : Exploratory and
              Euclidean methods in Environmental sciences
affy           Methods for Affymetrix Oligonucleotide Arrays
affydata       Affymetrix Data for Demonstration Purpose
affyio         Tools for parsing Affymetrix data files
affyPLM        Methods for fitting probe-level models
affyQCReport   QC Report Generation for affyBatch objects
AgiData        Data for the AgiND package
...

```

Ces bibliothèques sont installées dans le répertoire "library" créé lors de l'installation de R. La bibliothèque principale est "base".

```

> system.file()
[1] "/usr/lib/R/library/base"

```

Lors du lancement de R, seules les fonctions de certaines bibliothèques (essentielles) sont chargées dans la mémoire vive de votre ordinateur ("base", "methods", "stats", "graphics", "grDevices", "utils", "datasets"...) :

```

> search()
[1] ".GlobalEnv"          "package:stats"       "package:graphics"
[4] "package:grDevices"   "package:utils"       "package:datasets"
[7] "Autoloads"           "package:methods"     "Autoloads"
[10] "package:base"

```

L'environnement mémoire dans lequel vous travaillez s'appelle ".GlobalEnv". Les objets et fonctions des bibliothèques chargées en mémoire sont placés dans d'autres environnements mémoire pour éviter tout conflit (sinon vous pourriez écraser certains objets sans le savoir).

```
> ls() #ls(".GlobalEnv")
[1] "R.version"      "R.version.string" "version"
> ls("package:base")
 [1] "^"           "~"
 [3] "<"          "<<-"
 [5] "<="         "<-"
...
```

Pour installer une bibliothèque depuis le site du CRAN (<http://cran.r-project.org>), utilisez la fonction `install.packages` (sous windows, utilisez plutôt le menu).

```
> install.packages("R2HTML")
```

Vous pouvez visualiser les bibliothèques disponibles aux adresses suivantes :

- Site du CRAN : <http://cran.r-project.org/>
- Site de Bioconductor : <http://www.bioconductor.org/>
- Site du projet OMEGA : <http://www.omegahat.org/>

13 Le fichier Rprofile

Le contenu de ce fichier est analysé lors du démarrage de R. On pourra y placer des fonctions fréquemment utilisées ou du code spécifiant une action à réaliser (ex : `library(R2HTML)` si vous souhaitez que cette bibliothèque soit toujours chargée lors du démarrage). Sous windows, il faut éditer le fichier présent dans : "RepertoireD'installationDeR/library/base/R/Rprofile" Sous linux, il faut éditer le fichier présent dans le répertoire suivant. On placera ce fichier dans son "home" sous la forme d'un fichier caché ".Rprofile".

```
> baseR <- system.file("R",package = "base")
> baseR
[1] "/usr/lib/R/library/base/R"
> dir(baseR)
> file.show(file.path(baseR,"Rprofile"))
```