Correction TP d'introduction à R : un langage de programmation

Attention : Ceci est un "TP-cours" où vous contruisez votre propre savoir (quelle chance !). Prenez des notes ! Posez des questions ! On ne devient pas R-guru en 3h sans mobilisation du cerveau...

1 Commencer avec R

R est un système d'analyse statistique et graphique créé par Ross Ihaka et Robert Gentleman. Il dérive du dialecte S créé par AT&T Bell Laboratories. R est distribué librement, sous licence GNU GPL.

Pour lancer le logiciel R depuis les machines du réseau, il suffit de taper "R" dans un terminal. Pour quitter la session, il suffit de taper la commande `quit()`.

R est un langage interprété, et non compilé, ce qui signifie que les commandes tapées au clavier sont directement exécutées sans avoir besoin d'écrire un programme complet comme dans les langages comme C, Fortran ou Java.

La syntaxe de R est relativement simple et intuitive. Les variables, données et fonctions du langages sont stockés dans la mémoire de l'ordinateur pendant tout le fonctionnement de la session, sous la forme d'*objets* portant chacun un *nom*. L'utilisateur agit sur ces objets avec des *opérateurs* (arithmétiques, logiques, de comparaison, ...) et des fonctions.

1.1 Un premier programme en R

Exercice 1 *Lancer R dans une console. Taper le code suivant :*

```
print("Bonjour!")
x <- 5
x
```

Tester la commande `quit()`. *Observer la possibilité de conserver les objets définis durant la session.*

La commande `quit()` permet de quitter l'interpréteur et de revenir à la console normale. Les trois options proposées sont de sauver l'espace de travail puis de quitter (`y`), de quitter sans sauver l'espace de travail (`n`) ou d'annuler l'opération et de revenir à l'interpréteur (`c`).

R étant un langage interprété, il peut être utile de conserver dans un script à part les commandes que vous voulez utiliser, pour ne pas avoir à retaper toute la séquence et utiliser des fonctions intéressantes.

Exercice 2 *Lancer votre éditeur de texte favori, et tapez dedans la séquence d'instructions précédente. Sauvegardez dans un fichier "TP1.R". Pour charger le script dans l'interpréteur R, on utilise la commande* `source()`. *Rechargez votre script. Que constatez-vous ?*

Le chemin vers le fichier TP1.R doit être spécifié lorsque l'on fait appel à `source`. Deux possibilités : soit on indique tout le chemin vers le fichier : `source("/home/login/hmsn118/TP1.R")`, soit on change le répertoire de l'interpréteur pour se trouver au même endroit que le fichier voulu : `setpwd("/home/login/hmsn118"); source("TP1.R")`.
Remarquez l'emploi des guillemets autour des chemins vers les fichiers. Quand on charge le fichier, on remarque que la valeur de `x` n'est pas affichée. Ce qui est chargé, ce sont les définitions d'objets (il y a donc bien la valeur attendue dans la variable `x`) et les fonctions avec effet de bord, comme la fonction `print`. Si on veut explicitement afficher la valeur de `x`, il suffit de remplacer `x` tout seul par `print(x)`.

Par la suite, vous sauvegarderez toutes vos définitions d'objets dans votre script.

1.2 Les variables

L'assignation d'une valeur à une variable se réalise grâce à la flèche : `<-`, dans un sens ou dans l'autre.

Exercice 3 *Observer ce que produisent les commandes suivantes tapées successivement :*

```
n <- 15
5 -> n
x <- 1
X <- 10
n
x
X
n <- 10 + 2
n
n <- 3 + rnorm(1)
(10 + 2)*5
5 < 7
name <- "Jules"; n1 <- 10; n2 <- 100; m<- 0.5
ls()
```

`n <- 15` met la valeur 15 dans la variable `n`. N'affiche rien.
`5 -> n` met la valeur 5 dans la variable `n`. N'affiche rien.
`x <- 1` met la valeur 1 dans la variable `x`. N'affiche rien.
`X <-10` met la valeur 10 dans la variable `X` (différente de `x`). N'affiche rien.
`n` affiche la valeur 5.
`x` affiche la valeur 1.
`X` affiche la valeur 10.
`n <- 10 + 2` met la valeur 12 dans la variable `n`. N'affiche rien.
`n` affiche la valeur 12.
`n <- 3 + rnorm(1)` met dans la variable `n` une valeur aléatoire valant 3 + une valeur aléatoire selon la noi normale centrée réduite. N'affiche rien.
`(10 + 2)*5` affiche le résultat du calcul, c'est-à-dire 60.
`5 < 7` affiche la valeur `TRUE`.
`name <- "Jules"; n1 <- 10; n2 <- 100; m<- 0.5` réalise quatre affectations, une chaîne de caractères "Jules" dans la variable `name`, la valeur 10 dans la variable `n1`, la valeur 100 dans la variable `n2` et la valeur 0.5 dans la variable `m`. On remarque que l'on peut enchaîner plusieurs commandes sur la même ligne à condition de les séparer par des point-virgules.
`ls()` donne la liste de tous les objets qui ont été affectés jusqu'à présent. Ce sonr les objets qui seront sauvés si on quitte avec l'option `y`.

1.3 Créér, lister et effacer des données en mémoire

La fonction `ls()` permet de connaître, à un instant donné, les objets qui sont présents en mémoire.

La fonction `ls.str()` fournit des informations complémentaires sur les objets, et la fonction `ls(pat = "m")` va donner uniquement les objets dont les noms comportent un "m".

Exercice 4

1. *Testez les fonctions* `ls` *et* `ls.str` *avec différents patterns.*
2. *Que fait la sequence d'instructions suivante :*

   ```
   M <- data.frame(n1, n2, m)
   ls.str(pat = "M")
   ```

3. *Effacez tous les objets en mémoire à l'aide de la fonction* `rm`

1.

```
> ls(pat="name")
[1] "name"
> ls.str()
m : num 0.5
n : num 2.19
```

```
n1 : num 10
n2 : num 100
name : chr "Jules"
x : num 1
X : num 10
> ls.str(pat="n")
n : num 2.19
n1 : num 10
n2 : num 100
name : chr "Jules"
> ls.str(pat="m")
m : num 0.5
name : chr "Jules"
2.
> M <- data.frame(n1, n2, m)
> ls.str(pat = "M")
M : 'data.frame': 1 obs. of 3 variables:
$ n1: num 10
$ n2: num 100
$ m : num 0.5
> ls.str(pat = "M" , max.level = -1)
# produit un message d'erreur...
3.
> ls()
[1] "m" "M" "n" "n1" "n2" "name" "x" "X"
> rm("m","M","n","n1","n2","name","x","X")
> ls()
character(0)
```

1.4 L'aide en ligne

La fonction `help()` permet d'obtenir de l'aide sur la façon d'obtenir de l'aide... Pour résumer, il y a deux façons d'avoir une aide sur une fonction donnée, en tapant la commande précédée d'un point d'interrogation, ou en donnant la commande comme paramètre de l'aide. On peut aussi passer du mode aide en ligne au mode aide console en jouant sur le commutateur `htmlhelp={T,F}`.

Exercice 5 *Recherchez de l'aide sur la fonction* `rm`. *Comment détruit-on en une seule commande tous les objets de la session ? Comment détruit-on seulement les objets qui ont un "m" dans leur nom ? Testez l'aide en ligne* `help.start()`.

```
> ls()
[1] "m" "M" "n" "n1" "n2" "name" "x" "X"
> rm(list=ls(pat="m"))
> ls()
[1] "M" "n" "n1" "n2" "x" "X"
> rm(list=ls())
> ls()
character(0)
```

Dans ce qui suit, quand on spécifiera une fonction à utiliser pour résoudre un exercice, prenez l'habitude d'invoquer l'aide sur cette fonction pour comprendre son utilisation.

2 Variables : types, assignation, comparaison

En R, le type de base est le **vecteur**, dont nous reparlerons d'ici quelques sections. Les données dans R sont *typées*, c'est-à-dire que l'on ne sera pas autorisé à mélanger des booléens avec des données de type numériques, ou des chaînes de caractères. Voici les types de base :

- `numeric` qui peut être `integer` (entier) ou `double` (double). Les opérateurs applicables sont : `+`, `-`, `*`, `/`, `^` , `%%` (modulo), `%/%` (division entière)
- **logical** correspond au type booléen, il prend deux valeurs possibles `TRUE` et `FALSE` (en majuscule) que l'on peut abréger avec `T` et `F`. Les opérateurs sont `!` (négation), `&&` (ET logique), `||` (OU logique), `xor()` (OU exclusif).
- `character` désigne les chaînes de caractères. Une constante chaîne de caractère doit être délimitée par des guillemets. Fonctions de manipulation : `paste(..)`, `grep(..)`, etc.

Remarque : pour connaître la classe d'un objet i.e. le type associé à un objet, on utilise la fonction `class(nom_objet)`.

Les opérateurs de comparaison classiques `==`, `!=`, `>`, `>=`, etc., sont disponibles en R et renvoient une valeur de type `logical`.

Exercice 6 *Choisir trois valeurs numériques pour des variables a, b, et c. Calculer le discriminant delta correspondant à l'équation* $ax^2 + bx + c = 0$ *(rappel ? !* $\Delta = b^2 - 4ac$*. Afficher la valeur de delta, puis de delta* > 0*, delta* $= 0$*, delta* < 0*.*

```
> a = 3
> b = 1
> c = -3
> delta = b*b -4*a*c
> delta
[1] 37
> delta < 0
[1] FALSE
> delta == 0
[1] FALSE
> delta > 0
[1] TRUE
```

3 Structures de contrôle et boucles en tout genre

L'exercice précédent vous a sans doute donné envie de proposer une réponse un peu plus élaborée à la question cruciale, "est-ce que l'équation $ax^2 + bx + c = 0$ admet des solutions dans $\mathbb{R}$?". Pour cela, il faut commencer à établir des conditions, ce qui se fait de façon fort classique avec l'instruction `if ... else ...`.

Exercice 7 *Lancer la commande* `help("if")` *et remarquez que l'aide sur les conditionnelles présente aussi la syntaxe des boucles. Notez celle-ci pour plus tard.*

La syntaxe des boucles est `for (variable in sequence)` ... Ce n'est donc pas une forme classique pour les boucles. Il va falloir définir des séquences de valeurs que pourra prendre la variable.

Exercice 8 *Reprendre l'exercice sur la résolution de l'équation du second degré, en intégrant une conditionnelle pour tenir compte des trois cas, en affichant "Deux solutions réelles distinctes" quand delta* > 0*, "une solution réelle double" si delta* $= 0$ *et "pas de solution réelle, mais deux solutions complexes" sinon. Testez en changeant les valeurs de a, b, c pour couvrir tous les cas.*

```
if (delta == 0) {print("une solution réelle double")}
else if (delta > 0) {print ("Deux solutions réelles distinctes")}
else {print("pas de solution réelle, mais deux solutions complexes")}
```

Exercice 9 *Calculez la somme des 100 premiers nombres entiers à l'aide d'une boucle* `for`*, puis d'une boucle* `while`*. Nous verrons un peu plus tard comment faire cela en une seule commande !*

```
# Boucle for
> sum = 0
> for (i in 0:99) {sum = sum + i}
> sum
[1] 4950
# Boucle while
> sum = 0
> i = 0
> while (i < 100) {sum = sum + i; i = i + 1}
> sum
[1] 4950
```

4 Fonctions

Vous avez certainement éprouvé de la frustration et un sentiment d'inutilité lors des tests de l'exercice 8. Changer de valeurs nécessite de recopier les lignes de commande contenant la conditionnelle. Pour outrepasser cette difficulté, on peut tout simplement définir un bloc ou une fonction. Un bloc permet de définir une succession d'instructions, que l'on peut nommer. Par exemple :

```
bloc <- {
if(delta == 0){print("Une racine double")}
else if (delta > 0) {print("Deux racines réelles")}
else {print("Deux racines complexes")}
}
```

Il aurait alors été suffisant de relancer uniquement l'affectation des variables a, b, c, le calcul de *delta* et l'évaluation de `bloc` lors des tests de l'exercice 8.

Mais pour aller plus loin, on peut créer une fonction, qui prendra en paramètre trois variables numériques, et affichera le message adéquat. Une fonction se déclare avec le mot clé `function`. Une fonction est un bloc d'expressions, forcément nommée, visible comme un objet en mémoire avec `ls()`, qui peut être supprimée avec `rm()`, comme tout autre objet. Elle prend des paramètres en entrée (non typés, le typage se fera dynamiquement lors de l'interprétation), et renvoie une valeur en sortie (même si on n'écrit pas spécifiquement d'instruction pour cela).

Exercice 10 *1. Déclarez la fonction suivante :*

```
petit <- function (a, b){
d <- ifelse(a < b, a, 0)
return(d)
}
```

2. *Observez la syntaxe du* `ifelse` *qui est encore une autre façon d'exprimer une conditionnelle (R est plein de surprises !)*
3. *Observez à l'aide de* `ls.str(pat="petit")` *quel est le type de cet objet.*
4. *Lancez cette fonction sur plusieurs paramètres, éventuellement de types saugrenus.*
5. *Ajoutez une instruction après le* `return` *et relancez la fonction. Que constatez-vous ?*
6. *Enfin, enlevez la ligne contenant le* `return`*, et relancez la fonction. Que constatez-vous ? Essayez maintenant* `class(petit(4,5))`*.*

La syntaxe du `ifelse` se présente sous la forme d'une fonction à trois arguments, le premier est de type booléen, c'est la condition ; le deuxième est une expression qui est interprétée quand la condition vaut vrai, et la troisième est une expression qui est interprétée quand la condition vaut faux.
D'après `ls.str(pat="petit")`, l'objet petit est une fonction à deux paramètres a et b.
Quand on rajoute une expression après le return, elle n'est pas exécutée, mais si on enlève le return, la commande `petit(2,3)` n'affiche rien, mais si on récupère la valeur de `petit(2,3)` dans une variable, et qu'on interprète la variable, on récupère la valeur de `d`.
`class(petit(4,5)` donne "numeric", donc est bien de type numérique comme attendu.

Exercice 11 *Écrire une bonne fois pour toute une belle fonction qui résoud les équations du second degré !*

```
resol = function (a,b,c)delta = b*b - 4*a*c; ifelse(delta == 0, print("double"),
ifelse(delta > 0, print("réelles"), print("complexes")))
```

On peut également spécifier des paramètres par défaut, qui seront donc optionnels lors de l'appel. Attention, cependant, cela nécessite de faire des appels en explicitant les variables dont on précise la valeur. Par exemple, un tel appel pour la fonction `petit` serait : petit(a=2,b=3).

Exercice 12 *Écrire une fonction* `ma_somme` *qui prend en entrée un paramètre* `deb` *qui prendra une valeur par défaut de 1 et un paramètre* `fin` *qui prend comme valeur par défaut 100, et qui calcule la somme des entiers entre* `deb` *et* `fin`*. Testez différents types d'appels, en spécifiant ou non les variables. . .*

Remarquez la souplesse de la syntaxe des appels...

```
> masomme = function (deb=1,fin=100) {sum = 0; for (i in deb:fin) {sum = sum + i};
return(sum)}
> masomme()
[1] 5050
> masomme(0,99)
[1] 4950
> masomme(,23)
[1] 276
> masomme(5,100)
[1] 5040
> masomme(5,)
[1] 5040
> masomme(deb=8)
[1] 5022
```

On peut jouer de façon très fine avec les fonctions en R. Par exemple, tout objet peut être paramètre d'une fonction, y compris une autre fonction. Par exemple :

```
#met au carré une variable
CARRE <- function(x){
return(x^2)
}
#met au cube
CUBE <- function(x){
return(x^3)
}
#somme de 1 à n
#avec une FONCTION générique
somme <- function(FONCTION,n){
s <- 0
for (i in 1:n){
s <- s + FONCTION(i)
}
return(s)
}
#appel avec carré
print(somme(CARRE,10)) # résultat = 385
#appel avec cube
print(somme(CUBE,10)) # résultat = 3025
```

Exercice 13 *Définir une fonction "encore plus générique" en vous inspirant de la précédente. Il s'agira de remplacer l'opérateur* `+` *intervenant dans la boucle par une autre fonction (binaire, c'est-à-dire qu'on supposera qu'elle prend deux paramètres) passée en paramètre. De même, il faudra pouvoir passer en paramètre la valeur initiale de* `s`*. Testez avec le produit, pour une valeur initiale 1 et la fonction* `CARRE`*.*

```
> generique = function(FONCTION,OPER,n,init)
    \{ s = init; for (i in 1:n) \{s = OPER(s,FONCTION(i))\}; return(s)\}
> plus = function (x,y) \{return(x+y)\}
> fois = function (x,y) \{return(x*y)\}
> generique(CARRE,plus,20,0)
[1] 2870
> generique(CARRE,fois,20,1)
[1] 5.919012e+36
```

5 Structures de données vecteurs, matrices et dataframe

Nous allons maintenant nous intéresser (longuement, mais c'est important !) à la façon dont R organise les données. Au début du TP, il est précisé que le type de base est le vecteur. Pour bien s'en convaincre, il suffit d'interpréter une variable de type quelconque :

```
> x = 5
> x
[1] 5
```

Le `[1]` devant la valeur de x signifie en réalité qu'on a affaire à une ligne de vecteur, qui commence à l'indice 1.

5.1 Les vecteurs sous toutes les coutures

La function qui permet de créer des vecteurs est la fonction `c()`. Par exemple :

```
monvec = c(1,5,8,23,4,90,12)
```

Exercice 14 *Observez la séquence d'instructions suivantes, en notant pour chacune d'entre elle ce qu'elle produit. N'oubliez pas que vous pouvez vous servir de l'aide de R !*

```
monvec = c(1,5,8,23,4,90,12)
monvec
class(monvec)
length(monvec)
monvec[4]
masequence = 1:10
masequence
monvec + masequence
masequence = 1:length(monvec)
monvec + masequence
monvec < masequence
sum(monvec)
prod(monvec)
monvec * monvec
grosvec = 1:200
grosvec
rep(6,3)
```

```
> monvec = c(1,5,8,23,4,90,12)
> monvec
[1]  1  5  8 23  4 90 12
> class(monvec)
[1] "numeric"
> length(monvec)
[1] 7
> monvec[4]
[1] 23
```

On a déclaré une variable `monvec` qui contient un vecteur avec les valeurs indiquées (dont la fonction `c` permet de créer des vecteurs). Le type de données dans ce vecteur est numérique. Sa longueur est 7. On peut accéder directement aux éléments du vecteur en indiquant leur indice entre crochets.

```
> masequence = 1:10
> masequence
 [1]  1  2  3  4  5  6  7  8  9 10
> monvec + masequence
 [1]  2  7 11 27  9 96 19  9 14 18
Warning message:
In monvec + masequence :
  longer object length is not a multiple of shorter object length
> masequence = 1:length(monvec)
> monvec + masequence
[1]  2  7 11 27  9 96 19
```

On définit un deuxième vecteur `masequence` de longueur 10, qui contient les entiers entre 1 et 10. Remarquez la syntaxe pour définir une séquence d'entiers debut :fin. On ajoute ensuite, élément par élément, les deux vecteurs `monvec` et `masequence`, mais ce n'est pas autorisé car ils ont des longueurs incompatibles. Pour que cela fonctionne, il faut que la longueur de l'un soit multiple de la longueur de l'autre. On redimensionne donc `masequence` pour lui donner la longueur de l'autre vecteur. Et là, l'addition terme à terme est possible.

```
> monvec < masequence
[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
> sum(monvec)
[1] 143
> prod(monvec)
[1] 3974400
> monvec * monvec
[1]    1   25   64  529   16 8100  144
```

On peut faire de nombreuses opérations sur les vecteurs. Les comparer terme à terme avec un opérateur booléen comme dans la première instruction (on obtient alors un vecteur logique qui indique les comparaisons qui sont vraies et celles qui sont fausses), faire la somme ou le produit de tous les éléments du vecteur, ou encore multiplier terme à terme deux vecteurs (de longueur compatible).

```
> grosvec = 1:200
> grosvec
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
[109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
[127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
[145] 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
[163] 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
[181] 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198
[199] 199 200
```

On définit un gros vecteur. On remarque ici qu'à l'affichage, les nombres entre crochets indiquent l'indice du premier élément de la ligne.

```
> rep(6,3)
[1] 6 6 6
```

La fonction `rep` permet de définir des vecteurs de nombres répétés, on indique en premier argument le nombre à répéter, et en deuxième le nombre de répétitions.

Comme vous pouvez le constater, les vecteurs sont très souples d'utilisation ! C'est pourquoi il est bien important de documenter à l'aide de commentaires l'utilisation que l'on en fait.

Exercice 15 *Écrire une seule instruction permettant de résoudre l'exercice 9.*

```
> sum(0:99)
[1] 4950
```

On peut sélectionner certains élements dans les vecteurs, en indiquant entre crochets soit l'ensemble des indices sélectionnés, soit une condition qui doit être vérifiée par les élements sélectionnés. Dans ce dernier cas, on parle de "filtre".

Exercice 16 *Observer le comportement des instructions suivantes, puis écrire une fonction qui prend en entrée un vecteur numérique et renvoie en sortie les éléments pairs de ce vecteur.*

```
vect = c(27,32,56,78,11,45)
vect[c(2,4)]
vect[1:3]
vect[vect>40]
```

```
> vect = c(27,32,56,78,11,45)
> vect[c(2,4)]
[1] 32 78
> vect[1:3]
[1] 27 32 56
> vect[vect>40]
[1] 56 78 45
```

On définit le vecteur `vect` avec les valeurs indiquées. La deuxième instruction permet de sélectionner les éléments d'indices 2 et 4. La troisième instruction sélectionne les éléments d'indices 1 à 3, et la dernière les éléments strictement supérieurs à 40.

Exercice 17 *Testez les fonctions* `min()`, `max()`, `sort()`, `rank()`, `order()`. *Si x est un vecteur numérique, que fait* `x[order(x)]` *?*

```
> min(vect)
[1] 11
> max(vect)
[1] 78
> sort(vect)
[1] 11 27 32 45 56 78
```

Les fonctions `min` et `max` donnent respectivement la valeur minimale et maximale dans le vecteur. La fonction `sort` permet de trier le vecteur par order croissant.

```
> rank(vect)
[1] 2 3 5 6 1 4
> order(vect)
[1] 5 1 2 6 3 4
> vect[order(vect)]
[1] 11 27 32 45 56 78
```

La fonction `rank` indique, élément par élément, son indice dans le vecteur s'il était trié. Ainsi, le premier élément de `vect`, 27, se retrouvera en deuxième place dans le tableau trié.

La fonction `order` indique, place par place dans le tableau trié, l'élément de `vect` à aller chercher pour le mettre à cette place. Ainsi, le premier élément du tableau trié est à aller chercher à l'indice 5 dans `vect` : c'est l'élément 11.

Enfin, la dernière commande permet de trier le vecteur (comme `sort`) car il définit le vecteur prenant les valeurs de `vect` mais dans l'ordre précisé par `order(vect)`.

Pour faciliter l'utilisation des vecteurs en donnant une sémantique à leurs éléments, on peut nommer les éléments d'un vecteur à l'aide de la fonction `names`.

Exercice 18 *Explorez l'aide de la fonction* `names` *et créez un vecteur contenant les valeurs 1 à 7, avec les noms respectifs des jours de la semaine. Il est alors possible d'accéder directement au numéro du jour de la semaine en sélectionnant le nom entre crochet dans le vecteur. Expérimentez.*

```
> semaine = 1:7
> names(semaine) = c("Lundi","Mardi","Mercredi","Jeudi","Vendredi","Samedi","Dimanche")
> semaine
   Lundi    Mardi Mercredi    Jeudi Vendredi   Samedi Dimanche
       1        2        3        4        5        6        7
> semaine[3]
Mercredi
       3
> semaine["Jeudi"]
Jeudi
    4
```

Les fonctions usuelles permettant de calculer des observations statistiques sur les vecteurs sont très utiles en R. Par exemple, on dispose des fonctions `mean` et `sd` (pour standard deviation) qui calculent respectivement la moyenne et l'écart-type des vecteurs qu'elles prennent en paramètre.

Exercice 19 *Calculez la moyenne et l'écart-type des valeurs paires du vecteur* `vect` *de l'exercice 16.*

```
> mean(vect)
[1] 41.5
> sd(vect)
[1] 23.60297
```

5.2 Les matrices, vecteurs à deux dimensions

Pour construire une matrice, on utilise la fonction `matrix`, qui permet de créer la matrice à partir d'un vecteur. Par exemple :

```
v <- c(1.2,2.3,4.1,2.5,1.4,2.7)
m <- matrix(v,nrow=2,ncol=3)
attributes(m)
print(m)
```

Par défaut, les matrices sont organisées par colonne, mais on peut les organiser par ligne, en utilisant l'argument `byrow=TRUE`.

Exercice 20 *Créez une matrice m contenant les entiers de 1 à 9, organisée en lignes de 3 colonnes. L'argument* `nrow` *sera alors optionnel.*

```
> m = matrix(1:9,byrow=T,ncol =3)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Exercice 21 *Encore un peu d'observation... Que font les instructions suivantes :*

```
m = matrix(c(1,5,78,2,3,8,34,67,10,3,45,6),byrow=T,ncol=3)
m
nrow(m)
ncol(m)
2*m
m[1,3]
m[4]
m[1,]
m[,1]
m[,2:3]
cbind(m, matrix(1:12,byrow=TRUE,ncol=3))
rbind(m, matrix(1:6,byrow=TRUE,ncol=3))
```

```
> m = matrix(c(1,5,78,2,3,8,34,67,10,3,45,6),byrow=T,ncol=3)
> m
     [,1] [,2] [,3]
[1,]    1    5   78
[2,]    2    3    8
[3,]   34   67   10
[4,]    3   45    6
> nrow(m)
[1] 4
> ncol(m)
[1] 3
```

On déclare dans la variable `m` une matrice de 12 valeurs, organisées en 4 lignes de 3 colonnes.

```
> 2*m
     [,1] [,2] [,3]
[1,]    2   10  156
[2,]    4    6   16
[3,]   68  134   20
[4,]    6   90   12
```

On peut, comme pour les vecteurs, appliquer une opération à chaque élément de la matrice.

```
> m[1,3]
[1] 78
> m[4]
[1] 3
> m[1,]
[1]  1  5 78
> m[,1]
[1]  1  2 34  3
> m[,2:3]
     [,1] [,2]
[1,]    5   78
[2,]    3    8
[3,]   67   10
[4,]   45    6
```

On peut, encore une fois, comme pour les vecteurs, accéder aux éléments directement en indiquant le numéro de ligne puis de colonne, ou l'indice dans le vecteur initial. Si on ne spécifie qu'un numéro de ligne, on récupère le vecteur correspondant à cette ligne. De même pour les colonnes. De même pour les ensembles de numéros de lignes ou de colonnes.

```
> cbind(m, matrix(1:12,byrow=TRUE,ncol=3))
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5   78    1    2    3
[2,]    2    3    8    4    5    6
[3,]   34   67   10    7    8    9
[4,]    3   45    6   10   11   12
> rbind(m, matrix(1:6,byrow=TRUE,ncol=3))
     [,1] [,2] [,3]
[1,]    1    5   78
[2,]    2    3    8
[3,]   34   67   10
[4,]    3   45    6
[5,]    1    2    3
[6,]    4    5    6
```

Les commandes `cbind` et `rbind` permettent respectivement de rajouter des colonnes ou des lignes à une matrice, ou de coller deux matrices de dimensions compatibles par colonnes ou par lignes.

On peut attribuer des noms aux lignes et aux colonnes avec `rownames()` et `colnames()`. Cela peut-être très intéressant quand on récupère des données depuis un fichier csv.

La fonction `apply()` permet d'appliquer une fonction à chaque ligne ou chaque colonne d'une matrice. Ses arguments sont :

1. le nom de la matrice
2. un nombre pour dire si la fonction doit s'appliquer aux lignes (1), aux colonnes (2) ou aux deux (c(1,2))
3. le nom de la fonction à appliquer

Par exemple, pour calculer la somme de chaque ligne ou de chaque colonne d'un tableau :

```
# On crée d'abord une matrice avec 2 lignes et 3 colonnes
data<-matrix(c(1,2,3,4,5,6), nrow=2)
# On donne un nom aux lignes et aux colonnes
```

```
colnames(data)=c("C1","C2","C3")
rownames(data)=c("L1","L2")
# On utilise la fonction apply() pour faire la somme de chaque ligne
apply(data, 1, sum)
# Pour faire la somme de chaque colonne, on remplace 1 par 2
apply(data, 2, sum)
```

Cette fonction est très puissante !

Exercice 22 *Générez un vecteur de taille 100 au hasard avec la fonction* `rnorm` *(l'aide est votre amie !). Puis créez une matrice de 10 lignes et 10 colonnes à partir de ce vecteur. Enfin, calculez la moyenne par ligne, et par colonne, de cette matrice.*

```
> m = matrix(rnorm(100),ncol=10)
> m
             [,1]       [,2]       [,3]       [,4]        [,5]        [,6]
 [1,]  1.73279064  0.6633089  0.4425840 -1.0924915 -1.23986414  1.45949505
 [2,] -1.28620750 -1.4603098  0.6035881  0.9553478 -0.01037296 -1.85983813
 [3,]  0.86572143 -1.5881007 -0.8101011 -1.1722695  0.04887436 -0.02371838
 [4,] -0.82538929 -0.2755217  1.1890631  0.3430258  1.48970046  1.44439572
 [5,] -0.07427399 -0.1476672  1.3704012  0.6688816 -1.04508923 -0.59446880
 [6,]  0.66266390 -0.5203988  1.6869009  0.4968854  0.49843593 -1.06905512
 [7,] -1.22820190 -0.2304252  1.0609172 -1.6067677  0.56843268  0.10125443
 [8,]  1.08478106 -0.7836512 -2.3047876 -0.6364173  0.68439462 -1.02410296
 [9,] -1.13764493 -1.8789855  0.9890553 -1.2910277 -0.26058221 -0.22503630
[10,] -0.52802617  0.0133158  0.8282429  0.3079933  0.68023558  0.44221062
             [,7]       [,8]       [,9]       [,10]
 [1,]  1.1349124  0.7768832 -1.2470123 -0.35327655
 [2,] -1.2298063 -0.1460580  1.9364599  0.01895913
 [3,]  0.4903787  0.3400002 -1.3334350 -1.07793994
 [4,] -2.5260494 -0.2283895  0.5525725  1.44852100
 [5,] -0.3200862  1.5681339 -0.5698767  1.79628318
 [6,] -0.1381601  0.8272287 -1.4069676 -0.16962604
 [7,]  1.0761452 -0.3010551  0.5261457 -0.07505887
 [8,] -0.3917588 -1.1178345 -1.9867243 -0.06466824
 [9,] -0.6101261  0.2371679 -0.9500775 -0.26941687
[10,]  1.1849369  0.9973665  1.3980344 -0.89532208
> apply(m,1,mean)
 [1]  0.22773298 -0.24782378 -0.42605899  0.26119287  0.26522378  0.08679071
 [7] -0.01086136 -0.65407692 -0.53966738  0.44289879
> apply(m,2,mean)
 [1] -0.07337867 -0.62084355  0.50558640 -0.30268397  0.14141651 -0.13488639
 [7] -0.13296136  0.29534433 -0.30808808  0.03584547
```

5.3 Les dataframes, structure de stockage par excellence

Les vecteurs et les matrices ont vocation à conserver des données homogènes, c'est-à-dire de même type. Mais les vraies données utilisées en statistiques sont hétérogènes. Il va donc falloir en disposer sous forme de listes de vecteurs, que l'on va appeler des dataframes. Les dataframes vont se manipuler de façon très semblable aux matrices, avec la possibilité de faire un accès direct aux colonnes à l'aide de l'opérateur `$`.

Nous allons jouer avec un dataframe qui est chargé de base dans l'interpréteur R. Il s'agit du dataframe `mtcars`.

Exercice 23 *Tapez* `mtcars` *dans l'interpréteur. Que constatez-vous ? Testez les commandes suivantes :*

```
ls(mtcars)
mtcars$mpg
colnames(mtcars)
rownames(mtcars)
head(mtcars,5)
subset(mtcars,mtcars$vs==0)
summary(mtcars)
```

La commande `mtcars` affiche un gros tableau de données.

```
> ls(mtcars)
 [1] "am"   "carb" "cyl"  "disp" "drat" "gear" "hp"   "mpg"  "qsec" "vs"
[11] "wt"
```

Cette commande donne les noms des vecteurs colonnes stockés dans le dataframe, par ordre alphabétique.

```
> mtcars$mpg
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

L'accès à une colonne nommée se fait en utilisant le signe $ (comme en programmation objet, on accède aux propriétés de l'objet dataframe via ce symbole).

```
> colnames(mtcars)
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
> rownames(mtcars)
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"
> head(mtcars,5)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

Ces commandes permettent respectivement d'afficher les noms des colonnes, les noms des lignes et les cinq premières lignes du dataframe.

```
> subset(mtcars,mtcars$vs==0)
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
```

La fonction `subset` permet de ne sélectionner qu'une sous-partie du dataframe, celle qui satisfait la condition passée en second argument (le premier argument étant le dataframe lui-même).

```
> summary(mtcars)
      mpg             cyl             disp             hp
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
 Median :19.20   Median :6.000   Median :196.3   Median :123.0
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
      drat             wt             qsec             vs
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
       am              gear            carb
 Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :0.0000   Median :4.000   Median :2.000
 Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

La fonction `summary` est indicative des données qui sont stockées dans le dataframe. On peut voir grâce à cela si elles sont quantitatives ou qualitatives (ici elles sont toutes quantitatives) et si elles sont quantitatives, on obtient différentes mesures statistiques d'intérêt comme les extrêmes, les quartiles et la moyenne. Si elles sont quantitatives, on obtient les effectifs dans chaque classe détectée.

Exercice 24 *Calculez la moyenne des colonnes* `mpg`, `disp`, *et* `hp`, *pour les modèles qui ont 8 cylindres.*

```
> apply(subset(mtcars,mtcars$cyl==8),2,mean)[c(1,3,4)]
     mpg     disp       hp
 15.1000 353.1000 209.2143
```

Pour conclure ce TP en beauté, nous allons apprendre comment communiquer avec le monde extérieur depuis l'interpréteur R. En effet, les données sont rarement stockées directement dans l'interpréteur, mais le plus souvent dans des fichiers tabulés (format csv, ods, xls, etc.).

Pour charger un fichier tabulé, on utilise la fonction `read.table()`, et pour exporter un dataframe dans un fichier tabulé, la fonction `write.table()`.

Exercice 25 *Récupérez le fichier tabulé "mais.txt" sur l'espace Moodle du cours. Chargez ce fichier avec la commande :*

```
mais = read.table("mais.txt", sep="\t",header=TRUE)
```

Quelles sont les variables représentées dans ce tableau de données ? Lesquelles sont quantitatives, et lesquelles sont qualitatives (indice, la fonction `summary` *va vous aider) ?*

```
> summary(mais)
    Individu         Hauteur          Masse        Nb.grains
 Min.   :  1.00   Min.   :155.0   Min.   :1104   Min.   : 73.0
 1st Qu.: 25.75   1st Qu.:228.0   1st Qu.:1525   1st Qu.:203.0
 Median : 50.50   Median :263.0   Median :1830   Median :298.0
 Mean   : 50.50   Mean   :259.4   Mean   :1812   Mean   :292.6
 3rd Qu.: 75.25   3rd Qu.:291.0   3rd Qu.:2022   3rd Qu.:369.0
 Max.   :100.00   Max.   :359.0   Max.   :2752   Max.   :509.0
```

```
                NA's   :3       NA's   :3      NA's   :3
 Masse.grains          Couleur   Germination.epi    Enracinement  Verse
Min.   : 21.9   Jaune      :48   Non :90         Faible   :19    Non :57
1st Qu.: 60.9   Jaune.rouge:22   Oui : 9         Fort     :26    Oui :42
Median : 89.4   Rouge      :29   NA's: 1         Moyen    :28    NA's: 1
Mean   : 88.0   NA's       : 1                   Tres.fort:27
3rd Qu.:110.7
Max.   :152.7
NA's   :3
Attaque   Parcelle    Hauteur.J7    Verse.Traitement Nb.jours.attaque
Non:54   Est  :33   Min.   :163.0   Non :44          Min.   : 12.00
Oui:46   Nord :17   1st Qu.:224.2   Oui :55          1st Qu.: 47.50
         Ouest:36   Median :265.0   NA's: 1          Median : 79.00
         Sud  :14   Mean   :257.4                    Mean   : 83.82
                    3rd Qu.:291.0                    3rd Qu.:133.00
                    Max.   :347.0                    Max.   :133.00
                                                     NA's   :33
Censure.droite
Min.   :0.0000
1st Qu.:0.0000
Median :1.0000
Mean   :0.5735
3rd Qu.:1.0000
Max.   :1.0000
NA's   :32
```

Il y a quinze variables représentées dans ce dataframe, dont huit sont quantitatives (Individu, Hauteur, Masse, Nb.grains, Masse.grains, Hauteur, Nb.jours.attaque et Censure.droite) et sept sont quantitatives (Couleur, Germination.epi, Enracinement, Verse, Attaque, Parcelle et Verse.Traitement). Remarquez qu'il y a des valeurs "NA" indiquées pour certaines variables. Ce sont des données manquantes (Not Available).

Nous continuerons à jouer avec ce jeu de données dans un prochain TP... Quittez la session en sauvegardant votre espace de travail, vous gagnerez du temps !