Mise en œuvre d'algorithmes en R

T. Guyet

AGROCAMPUS-OUEST, Centre de Rennes

Remerciements : Merci à Simon Malinowski pour sa relecture attentive et ses remarques.
Version 0.3, Dernière modification : 5 mars 2014

Table des matières

1 Introduction

L'objectif de ce cours est la mise en pratique des enseignements d'algorithmique. Le langage R est utilisé comme support d'enseignement de l'algorithmique.
En particulier, l'objectif n'est pas d'introduire à la programmation en R et encore moins de présenter les bonnes pratiques de ce langage. Pour cela, il faudra se reporter sur des cours de programmation en R.
Dans le cadre de ce cours, je parlerai de "programme" pour se reporter à des fichiers dans lesquels sont écrites les instructions qu'on veut faire exécuter à la machine. Pour les programme R, on parle généralement de "script" pour désigner ce fichier. Par extension, quand je parle de "programme", cela peut également désigner l'ensemble des instructions qui réalisent la tâche souhaitée.
Dans la suite, les exemples sont présentés sous deux formes :

- les exemples exécutés directement dans la console R : toutes les lignes d'instructions commencent alors par le caractères ">", et les affichages de la console sont alors reproduits :

```
> a <-10
> print( a )
[1]  10
```

- les exemples de programmes écrits dans des fichiers ".R" : les lignes ne commencent pas par ">", et il n'y a pas d'affichage (ils sont effectués lors de l'exécution du fichier-programme)

```
a <- 10
print( a )
```

2 Environnement de programmation de R

2.1 Édition et exécution de programmes en R

Éditer un programme R consiste à éditer un fichier ".R" qui contiendra les instructions du programme. Ces programmes sont plus souvent appelés des **scripts**.
Pour créer un nouveau script depuis le logiciel R, vous pouvez aller dans le menu `Fichier > nouveau script`. Une nouvelle fenêtre s'ouvre dans laquelle vous pouvez écrire votre programme. Pour enregistrer votre programme, utilisez le menu `Fichier>Sauver` et donnez un nom de fichier explicite avec une extension ".R".
Pour éditer un fichier ".R", il faut :

1. Depuis R : dans le menu `Fichier>ouvrir un script ...` vous pouvez ouvrir l'éditeur de R
2. Depuis un logiciel extérieur (comme SciTE, NotePad++ ou Tinn-R), vous pouvez ouvrir le fichier depuis le logiciel choisi ou à partir de l'explorateur Windows. Les éditeurs extérieurs permettent généralement de mieux présenter le code (coloration syntaxique)

Pour exécuter un programme enregistré dans un fichier ".R", il faut utiliser la commande "source" depuis la console de R.
Pour interrompre l'exécution d'un programme, vous pouvez utiliser la combinaison de touches `Ctrl+C`.

2.2 RStudio

RStudio est un environnement de développement pour R. Des distributions de RStudio sont disponibles gratuitement pour Microsoft Windows, Mac OS X et Linux. La Figure 1 illustre

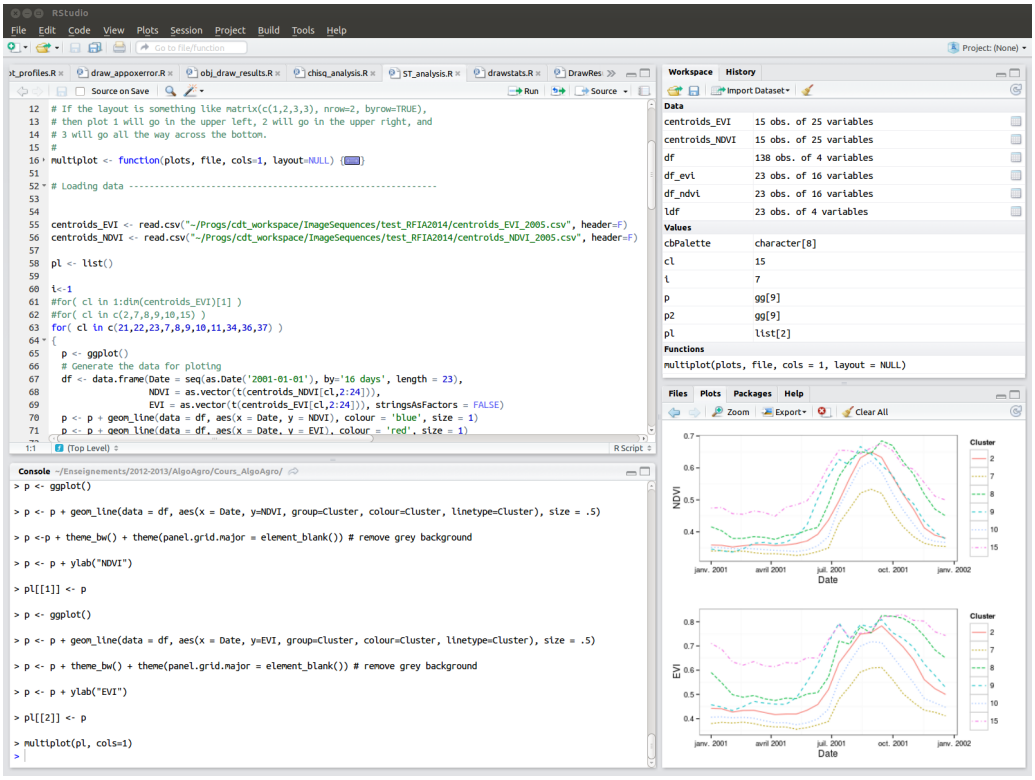


FIGURE 1 – Interface de RStudio

l'interface du logiciel.
C'est un outil moderne, professionnel et efficace pour aider au développement de programme avec R. Sa manipulation est relativement aisée et vous fera gagner du temps que vous ayez juste quelques lignes de R à écrire ou bien un très gros programme. Actuellement, c'est la seule réelle solution d'environnement de programmation pour R, ce logiciel est donc largement utilisé, dans les universités et dans les entreprises.
Les intérêts majeurs des outils tels que RStudio sont :

- avoir une coloration syntaxique adaptée au langage R : les chaînes de caractères sont en vert, les mots clés du langage sont en bleu, les commentaires en vert clair, etc. Ces mises en valeur facilitent la lecture des programmes,
- un outil d'indentation automatique : pour nettoyer votre code, il est possible de lui demander de l'indenter automatiquement !
- la completion : si vous taper le début d'une fonction et que vous utiliser la touche `Tab`, le logiciel vous propose toutes les fins possibles : ceci permet de retrouver rapidement le nom d'une fonction qu'on connait approximativement. Mieux encore, dans les parenthèses d'une fonction, la touche `Tab` donne accès à la liste des paramètres de la fonction. C'est particulièrement utile pour R.
- l'intégration de toutes les fonctionnalités sur un même écran : plus besoin de changer de fenêtre entre les graphiques, R et votre script, tout est visible en même temps !

2.2.1 Mettre en place son environnement de travail

RStudio est disponible en deux versions : RStudio *Desktop*, pour une exécution locale du logiciel comme tout autre application, et RStudio *Server* qui, lancé sur un serveur linux, permet d'accéder à RStudio par un navigateur web. Vous n'aurez besoin que de RStudio en version *Desktop*.
Pour installer RStudio *Desktop* (sous Windows/Linux ou Mac) :

1. il faut commencer par installer R en le téléchargeant par exemple ici `http://cran.r-project.org/`,
2. ensuite, il suffit de télécharger RStudio : `http://www.rstudio.com/ide/download/desktop` et l'installer,

Ensuite, RStudio devient le seul environnement de travail nécessaire pour développer des programmes en R. Il suffit donc de démarrer RStudio et vous êtes prêt à travailler !

2.2.2 Organisation de l'interface de RStudio

L'interface RStudio (Figure 1) est composée de quatre fenêtres :
- Fenêtre d'édition (en haut à gauche) : dans cette fenêtre apparaissent les fichiers contenant les scripts R que vous êtes en train de développer.
- Fenêtre de commande (en bas à gauche) : cette fenêtre contient une console dans laquelle les codes R sont saisis pour être exécutés. Vous pouvez également écrire directement dans cette console pour faire exécuter des commandes exactement comme vous le feriez dans R
- Fenêtre espace de travail/historique (en haut à droite) : contient les objets en mémoire, que l'on peut consulter en cliquant sur leur noms, ainsi que l'historique des commandes exécutées,
- Fenêtre explorateur / graphique / packages / aide (en bas à droite) : l'`explorateur` permet de se déplacer dans l'arborescence des répertoires (et de définir votre répertoire de travail), la fenêtre `graphique` contient les graphiques tracés via R (il est possible de les exporter), la fenêtre `packages` montre les packages installés et actuellement chargés et la fenêtre d'`aide` donne accès à la documentation de R sur les fonctions et packages.

Remarque 2 - Masquage de partie du code

Dans la fenêtre d'édition, vous noterez que les lignes sont numérotées. Juste à droite des numéros, certaines lignes contiennent une petite flèche. Si vous cliquez sur la flèche, vous pouvez masquer certaines lignes du code pour le rendre globalement plus lisible lorsqu'il est long. Ces flèches sont disponibles sur les débuts de blocs d'instruction (structures de contrôle, ou fonction). Il suffit de cliquer de nouveau sur la flèche vers la droite pour faire réapparaître le code.

2.2.3 Travailler avec RStudio

Pour travailler avec RStudio, vous devez savoir créer un programme, le sauvegarder et l'exécuter. Toutes les autres fonctionnalités sont, dans le cadre de ce cours, superflux.
- Création d'un nouveau fichier : utiliser l'icône `+` (en vert) en haut à gauche de l'interface (dans la barre d'icônes) et choisir la création d'un `R Script`.
- Enregistrer votre fichier : tant que le nom du fichier est en rouge (accompagné d'une étoile), c'est que le fichier n'a pas été enregistré récemment. Pour enregistrer votre fichier, utiliser la combinaison de touche `Ctrl+S` ou le menu `Fichier>Enregistrer`.
- Exécuter votre script : comme d'habitude, il y a plusieurs solutions : avec des combinaisons de touches ou avec des icônes (au dessus de la fenêtre de votre code). Deux actions sont proposées (et utiles) :
  - exécution de la sélection de code courante (avec des combinaisons de touche `Ctrl+Return` ou avec l'icône `Run`) : seules les lignes sélectionnées sont exécutées.
  - exécution de l'intégralité du script (avec des combinaisons de touche `Ctrl+Shift+Return` ou avec l'icône `Source`) : tout le script est exécuté depuis le début.

  Personnellement, j'incite fortement à utiliser uniquement la seconde solution !

2.3 Décryptage d'un premier programme

```
#
# ceci est mon premier programme
#

#declaration et affectation des variables a et b
a<-10
b<-10

#declaration d'une chaine de caracteres
str <- ''Le resultat est :''

# operation arithmetique
res <- a + b
res <- res + b

# affichages de variables dans la console
print( str )
print( res )

#declaration et affectation d'un vecteur (tableau de dimension 1)
vec <- c(34, 45, 56)

#acces aux elements du tableau
vec[2] <- vec[1] + vec[3]

#definition d'une fonction
MaFonction <- function( v )
{
  v <- v + 1
  return( v )
}

#appel de fonction
vec2 <- MaFonction( vec )
print(str)
print(vec2)
```

Remarques :
- les lignes commençant par "#" sont des lignes de commentaire, elles ne sont pas exécutées par R
- la fonction `print` permet de demander l'affichage du contenu d'une variable

2.4 Session R

Les variables définies dans une session R sont utilisables à tout moment dans un programme. La commande `ls()` liste toutes les variables qui ont été déclarées dans une session. Pour supprimer toutes les variables d'une session, il faut utilisant la commande `rm(list =ls()`

2.5 Accéder à l'aide

Pour accéder à l'aide, tapez l'une des instructions suivantes dans la console R (remplacez (...) par la fonction dont vous recherchez l'aide : un seul mot) :

```
> ?(...)
```

ou

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

La description de la fonction trouvée s'affiche directement dans la console R. Les aides en ligne sont en anglais. Vous pouvez faire défiler le texte à l'aide des flèches haut et bas. Pour quitter l'aide, tapez sur 'q'.
Pour les recherches de mots clés de R, utiliser des guillemets (simples ou doubles). Par exemple :

```
> ?'while'
```

Si on ne connaît pas le nom de la fonction, on peut utiliser la commande "help.search".
Cette commande produit une liste de toutes les fonctions de R dont la description contient l'expression ou le terme indiqué. On peut taper par exemple :

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

Accès à l'aide des fonctions depuis le site internet de R :
– les documentations sur le langage R : http ://cran.r-project.org/doc/manuals/R-lang.html
– recherche dans la documentation à l'aide de la recherche dans le panneau à gauche du site http ://cran.r-project.org

2.6 Utilisation de packages

La simple commande ci-dessous permettra de télécharger le package de fonctions du package RODBC (*cf.* section 6.4)

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

La procédure d'installation commence par le choix d'un site de téléchargement puis le téléchargement est automatiquement lancé. Finalement, pour les packages développés dans un autre langage que R (en C ou en Fortran), il y a une étape de compilation du programme. Pour vous, peut importe les étapes, dans le principe tout ce passe automatiquement !

Pour charger le package en mémoire dans une session R, on fera la commande ci-dessous. Celle-ci aura pour effet de définir toutes les fonctions du package.

```
> library(RODBC)
```

Une fois qu'un package a été téléchargé, il ne sera plus nécessaire de refaire la commande install.packages en revanche, il sera nécessaire de faire appel à la commande library à chaque nouvelle session R.

Lorsqu'on écrira un script qui nécessite un package particulier, vous pourrez commencer votre script par les commandes indiquant les packages nécessaires :

```
require("RODBC")
```

Lors de l'exécution de cette commande, R tentera de charger le package en mémoire et si celui-ci n'est pas installé il affichera une erreur à l'utilisateur.

Remarque 3 - Installation des *packages* avec RStudio

RStudio simplifie également l'installation et l'activation des *packages* R puisque tout peut se faire simplement à l'aide de l'interface graphique (fenêtre en bas à droite, onglet `Packages`).

---

3 Les variables dans R

En programmation, les données sont représentées en mémoire par des variables. La programmation est ensuite uniquement de la manipulation de variables. La façon dont sont organisées les données dans les variables influence également les algorithmes. Dans le cas du R, vous aller découvrir plein de structures de données élémentaires qui vont vous faciliter la vie. Ces structures de données ne sont pas disponibles dans des langages élémentaires comme le C, il est alors nécessaire de les reconstruire où de procéder différemment pour programmer une même tâche. Les structures de données sont même à l'origine de paradigmes de programmation en entier. La programmation orienté objet (par ex. le Java) utilise une forme particulière de représentation des données. Dans le cas du R, langage de statistique, l'application spécifique du langage a aussi conduit à y développer des structures de données particulières (par exemple les **data.frame**).
Pour ces différentes raisons, il est donc essentiel de bien comprendre comment sont organiser les variables en mémoire et ce qu'on peut en faire.
Dans cette section, on commence par présenter les bases des manipulations des variables en présentant le fonctionnement des variables scalaires, c'est-à-dire des nombres, par opposition aux matrices. Dans un second temps, on présentera l'utilisation des tableaux : à 1 dimension (les vecteurs), à deux dimensions (les matrices) ... les tableaux à n dimensions. Ensuite, on s'intéressera à des structures de données spécifiques à R : **data.frame**, **list** , **factor**.

3.1 Déclaration et affectation de scalaires

L'affectation d'une variable utilisation une notation fléchée ("<−" qui s'écrit avec le signe inférieur et le signe moins).
En R, il n'y a pas de déclaration explicite de variable. Les variables sont implicitement déclarées à leur initialisation. Il faut noter que le langage R distingue les minuscules des majuscules [1] : la variable **a** est donc différente de la variable **A**.
Si la déclaration n'est pas explicite, il faut néanmoins qu'une variable ait été initialisée (et donc implicitement déclarée) avant de pouvoir l'utiliser.

Exemple 1 - Déclaration d'une variable

La troisième ligne ne fonctionne pas car **d** *n'a pas été définie avant.*

```
> a <- 10 # declare la variable a et l'affecte avec la valeur 10
> b <- 3*a # declare la variable b et l'affecte avec le resultat de l'operation 3*a
> b <- 3*d # provoque une erreur car d n'existe pas
Erreur : objet 'd' introuvable
```

! Attention ! - Mon programme marche en TP, mais plus chez moi ? ? ?

Lorsque vous travaillez sur un script, vous serez amenés à exécuter celui-ci et y faire des modifications. Il est très probable que certaines variables utilisées dans une version antérieures de votre script existent dans R sans pour autant que la variable ait bien été initialisée (donc déclarée) dans votre script. Il est alors tout aussi probable que la prochaine fois que vous travaillerez sur votre script (avec un environnement R qui n'a pas le même historique) votre programme se comporte différemment.

1. Lorsqu'on distingue les minuscules des majuscules, on parle de "sensibilité à la casse".

Pour éviter cela, je conseille d'écrire des scripts qui vont supprimer toutes les variables de la mémoire de R avant de commencer. Pour cela, vous pouvez utiliser la commande `rm(list = ls())`.

3.1.1 Typages des variables

Bien que les variables ne soient pas déclarées, elles sont bien typées. Le type de la variable dépend de ce qui a été mis dedans (!). En R, on parle plutôt de "mode" et de "mode de stockage".
Les modes standards de R :

- **numeric** : ce type désigne un nombre sans plus de précision sur leur nature exacte : nombre entier, à virgule, ...
- **logical** : les booléens (dont les valeurs sont **TRUE** ou **T** et **FALSE** ou **F**).
- **character** : les textes ou caractères, en mémoire les textes sont représentés comme des suites de caractères, chaque caractère étant encodé par une valeur numérique entière.

Pour les nombres, il y a classiquement une différence entre les représentations des nombres entiers (représentation exactes) et les représentations des nombres à virgules (représentations approchées des nombres). Une variable de mode **numeric** peut être représentée en mémoire de deux façons :

- **double** ou **real** : les nombres réels (c'est le type par défaut de tous les nombres). Les nombres qui sont des **double** ou **real** sont aussi des **numeric**.
- **integer** : les entiers relatifs (positifs et négatifs). Les nombres qui sont des **integer** sont aussi des **numeric**.

Pour connaître le mode de stockage d'une variable numérique, vous pouvez utiliser la fonction **storage.mode**.

Remarque 4

Vous serez rapidement amenés à découvrir des variables qui ne sont dans aucune de ces catégories. Il est en effet possible de définir ses propres types de variables pour étendre les possibilités algorithmiques. En R, la définition d'un nouveau type de données consiste à définir une classe. L'utilisation des classes n'est pas l'objet de ce cours. Elles ne seront pas abordées.

Pour connaître le type d'une variable (pour les modes de standard, les modes de stockages et les classes personnelles), il est possible de tester son type avec les fonctions génériques "**is**" (**is.numeric**, **is.logical** ...)

Exemple 2 - Types de données

```
> a <- 3.7
> b <- 3
> mode(a) # Affichage du mode de la variable : dans ce cas, c'est un nombre
[1] "numeric"
> storage.mode(a) # Affichage du mode de stockage de ce nombre
[1] "double"
> storage.mode(b) # par defaut, le mode de stockage est "double", meme si on y met un entier.
[1] "double"
> is.double(a) # TRUE car par defaut les nombres sont des double
[1] TRUE
```

```
> is.integer(a) # FALSE car a n'est pas en entier !
[1] FALSE
> is.logical(b) # TRUE car c est bien un booleen (resultat de la fonction is.numeric)
[1] TRUE
> is.integer(b) # FALSE car les nombres sont des double, meme si 3 semble etre entier, il est pris
    pour un double
[1] FALSE
> is.numeric(b) # TRUE car les nombres sont tous des numeric
[1] TRUE
```

3.1.2 Conversion des nombres / coercion

Pour *forcer* un type, on peut utiliser les fonctions génériques "as" (as.numeric(...), as.logical (...)). Ces fonctions "as" permettent de faire des conversions [2] de types (on parle de *coercion* pour R). Lorsque des erreurs de conversion sont faites, un message averti de l'erreur et la valeur est remplacée par **NA** (*cf.* Section 3.2.5 sur les valeurs manquantes).

Exemple 3 - Conversion des types

```
> a <- as.integer(3.7) # ATTENTION, ici a va contenir la valeur entiere 3
> print(a)
[1] 3
> b <- as.integer(3)
> is.double(a)
[1] TRUE
> is.double(a)
[1] FALSE
> is.integer(a)
[1] TRUE
> is.integer(b)
[1] TRUE
> is.double(b)
[1] FALSE
> is.numeric(a) #a est bien un nombre
[1] TRUE
> is.numeric(b) #b est bien un nombre
[1] TRUE
```

```
> vecBoo <- TRUE
> vecStr <- as.character(vecBoo) #conversion de la valeur booleenne en texte
> print(vecBoo)
[1] TRUE
> print(vecStr) #l'affichage entre guillemet indique qu'il s'agit d'un texte
[1] "TRUE"
> mode(vecStr)
[1] "character"
```

3.1.3 Mutabilité des variables

Le type des variables est "mutable" : une variable initialisée avec un entier peut se voir attribuer une valeur non-entière (y compris du texte). Ceci est une propriété inhérente au fait que le type

2. En terme technique informatique on parle de *transtypage* ...

de la variable dépend de ce qu'elle contient.
Le type de la variable est alors définie par le type de **la dernière valeur** avec laquelle a été affectée la variable.

Exemple 4 - Mutabilité

L'exemple ci-dessous illustre comment une variable initialisée par une valeur numérique (et donc par défaut de type **double***) change de type lorsqu'on lui met une autre valeur, explicitement entière à l'intérieur. Pis, lorsqu'on y met du texte, cette même variable n'est même plus un nombre.*

```
> a <- 4.3
> is.integer(a) # ICI a n'est pas un entier
[1] FALSE
> a <- as.integer(4) # On met une autre valeur dans a
> is.integer(a) # ICI a est un entier
[1] TRUE
> a <- "coucou"
> is.numeric(a) # ICI a n'est plus un nombre !
[1] FALSE
```

Le caractère mutable des variables est assez particulier à R. C'est un caractère qui se rencontre dans quelques langages de programmations scientifiques (matlab, scilab, ...) mais assez peu dans les langages généraliste (c'est tout de même le cas de python).
C'est à la fois un défaut et une qualité du langage :

- L'absence de typage strict des variables est une propriété qui peut être vue comme pratique pour les programmeurs novices, mais pour le programmeur averti qui veut réaliser une programme propre, c'est une difficulté supplémentaire. L'absence de typage permet d'utiliser des variables dont les types sont implicites pour le programmeur. Cette permission de l'implicite induit des difficultés de programmation : pour faire une fonction générique, c'est au programmeur de prévoir tous les cas possibles.
- De plus, l'absence de typage rend les programmes difficiles à lire puisqu'on se demande en permanence ce que contient une variable.
- Finalement, un problème de pose à la lecture des paramètres formels d'une fonction. N'étant pas typés, de nouveau, c'est au programmeur de prévoir toutes les éventualités d'utilisation de sa fonction.

3.2 Structures de données homogènes

3.2.1 Les variables sont des vecteurs

Les variables de bases en R sont des vecteurs (tableaux à une dimension).
La constitution manuelle d'un vecteur se fait par la fonction **c**(**el1**, **el2**, ...) . Chaque élément du vecteur doit être séparé d'une virgule (les points servent de séparateurs des décimales d'un nombre).
L'accès à un élément du tableau se fait en donnant l'indice du tableau entre crochets. Ceci permet d'accéder à la valeur de l'élément désigné ou d'affecter cet élément avec une valeur particulière.
La longueur d'un tableau est donnée par la fonction **length**()

Exemple 5 - Vecteurs

```
> v <- c(3.4, 5, 6.78, 2)
> print( v[3] ) # Affiche de troisieme element du tableau v
[1] 6,78
> length(v)
[1] 4
> v[2] <- 43.5
> print( v ) # Affiche tout le vecteur v
[1] 3.40 43.50 6.78 2.00
> v <- 1:10
> print( v[-5] ) # Attention : les indices negatifs servent a supprimer un element du vecteur !
[1] 1 2 3 4 6 7 8 9 10
```

Il est possible d'utiliser un vecteur d'indices pour sélectionner un sous-ensemble de valeurs d'un vecteur. Un sous-ensemble de valeur peut également se faire affecter par un vecteur à condition qu'il soit exactement de même taille (sinon erreur).

! Attention !

Il n'est pas possible d'utiliser plusieurs indices séparés par des virgules : la signification d'une telle écriture est différente (cf. matrices) ! Il faut utiliser un vecteur d'indices pour sélectionner plusieurs éléments d'un vecteur.

Exemple 6 - Sélection par un vecteur d'indices

```
> v <-c("toto", "tata", "foo", "bar")
> v[ c(1, 4) ]
[1] "toto" "bar"
> v[ c(4, 2) ]
[1] "bar" "tata"
> v[ c(4, 2) ] <- c("ubu", "ran")
> print(v)
[1] "toto" "ran" "foo" "ubu"
```

Tous les éléments d'un vecteur sont de même type : integer, character, double, ... Si tous les éléments passés à la fonction c (...) ne sont pas de même type, ils sont convertis dans le type le plus "fort" présent dans la liste des éléments, l'ordre de force de type étant le suivant :

character>double>integer>logical

Exemple 7 - Typage d'un vecteur par ses éléments

Souvenez vous que R est un langage qui se croit intelligent ... et il essaye de se croire plus malin que vous en tentant de vous aider sans vous le dire !
Dans l'exemple ci-dessous, il est important de noter la différence d'affichage de **v** *lors du premier* print *et du second : lors du premier affichage,* ***tout*** *le vecteur est de type* character *et*

les nombres sont affichés entre guillemets. Ceci indique que "3.4" est pris comme une chaîne de caractère et non comme un nombre. Lors du second affichage, le vecteur **v** *est un vecteur d'entiers. Le chiffre trois est alors affiché sans guillemets et les valeurs binaires* **TRUE**/**FALSE** *sont traduites par des nombres 1 et 0.*

```
> v <- c(3.4, 5, 6.78, as.integer(2))
> is.integer(v)
[1] FALSE
> v <-c("coucou", 3.4, 5, 6.78)
> is.character( v )
[1] TRUE
> print(v)
[1] "coucou" "3.4" "5" "6.78"
> v <- c(TRUE, FALSE, as.integer(3) )
> print( v )
[1] 1 0 3
> v <- c(TRUE, FALSE, FALSE)
> print( v )
[1] TRUE FALSE FALSE
```

Dans ce second exemple, le simple fait d'insérer un élément qui n'est pas du même type que les autres à pour conséquence de transtyper tout le vecteur !

```
1 > vecNum <- c(1, 2, 3) # construction d'un vecteur de 3 elements
2 > print(vecNum)
3 [1] 1 2 3
4 > vecNum[2] <- "two" #insertion d'un texte dans le second element du tableau
5 > print(vecNum) # le vecteur ne contient plus que du texte
6 [1] "1" "two" "2"
7 > mode(vecNum) # confirmation
8 [1] "character"
```

Il est souvent utile de définir un vecteur qui contient une séquence de nombres. Il est également possible de définir de tels vecteurs soit avec la fonction **seq**, soit avec la notation du type 1:232.

Exemple 8 - Séquences

```
> v <- 4:32 #v est un vecteur contenant 29 cases dont les valeurs varient de 4 a 32.
> v
[1] 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
[26] 29 30 31 32
> v <- seq(4,32) # idem
> v <- seq(4, 32, 2) # v est un vecteur contenant 15 cases dont les valeurs varient de 4 a 32, mais de
    2 en 2
> v
[1] 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
```

3.2.2 Accès aux données d'un vecteur

La notation crochetée (avec "crochets") permet d'accéder aux données d'un vecteur. Dans les crochets, il est possible d'utiliser :
– un nombre entier : l'indice d'un élément du vecteur,
– un vecteur de nombres entier : le résultat sera alors le sous-vecteur pour les indices donnés,

– un nombre négatif ou un vecteur de nombre négatif : retranche des éléments du vecteur.

Exemple 9

```
> vecNum <- 1:10
> print(vecNum)
 [1]  1  2  3  4  5  6  7  8  9 10
> vecNum[3]
[1] 3
> vecNum[5:8]
[1] 5 6 7 8
> idx <- c(3, 7, 9)
> vecNum[idx]
[1] 3 7 9
> vecNum[-4:-6]
 [1]  1  2  3  7  8  9 10
```

3.2.3 Les matrices

Une matrice est un tableau de dimension 2.
Comme les vecteurs, les matrices contiennent des objets de type **character**, **double**, **integer** ou **logical**, et tous les objets doivent être de même type. La syntaxe pour créer une matrice est la suivante :

```
> mat <- matrix(vec, ncol=p, nrow=n)
```

– **vec** : contient les valeurs avec lesquelles initialiser la matrice (si la taille du vecteur ne correspond pas à la taille de la matrice annoncée, un **Warning**[3]est affiché)
– **n** : définit le nombre de ligne de la matrice
– **p** : définit le nombre de colonne de la matrice

Il est possible de ne pas préciser **vec**, dans ce cas, la matrice générée est vide (contient des **NA**, *cf.* valeurs manquantes).
Pour connaître la taille d'une matrice, il faut utiliser les fonctions **dim**, **nrow** et **ncol**. La fonction **length** donne le nombre total d'éléments de la matrice (**nrow**×**ncol**).
Pour accéder à un élément du tableau, il faut indique les indices pour le numéro de colonne et le numéro de ligne entre crochet, séparé par une virgule :

```
> mat [col, lig ]
```

Exemple 10 - Matrices

```
> m <- matrix( 1:12, ncol=4, nrow=3)
> m
[,1] [,2] [,3] [,4]
[1,]  1  4  7  10
[2,]  2  5  8  11
[3,]  3  6  9  12
> m[1, 3]
[1]  7
```

3. Un **Warning** est un message affiché dans la console pour avertir que quelque chose d'anormal s'est passé. Le **Warning** n'est pas une erreur, mais il peut être le symptôme d'une erreur.

```
> m[3,1]
[1]  3
> dim(m)
[1]  3  4
> dim(m)[1] #permet de recuperer le nombre de colonne
[1]  3
```

Pour construire une matrice contenant uniformément une valeur, utiliser le paramètre **data**.

```
> m <- matrix( data=3, ncol=4, nrow=2)
> m
     [,1] [,2] [,3] [,4]
[1,]    3    3    3    3
[2,]    3    3    3    3
```

Sans **data**, *la matrice ne contient que des vides (*NA*)*

```
> m <- matrix( ncol=4, nrow=2)
> m
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
```

De même que pour les vecteurs, il est possible de sélectionner une sous-matrice à partir d'un vecteur de numéros de lignes et d'un vecteur de numéros de colonnes :

Exemple 11 - Sous-matrices

(suite de l'exemple précédent)

```
> sub_m <- m[ c(1,3), c(3,1) ]
> print( sub_m )
     [,1] [,2]
[1,]    7    1
[2,]    9    3
> m[ c(1,3), c(3,1) ] <- matrix(1:4, ncol=2, nrow=2)
> m
     [,1] [,2] [,3] [,4]
[1,]    3    4    1   10
[2,]    2    5    8   11
[3,]    4    6    2   12
```

3.2.4 Les tableaux / array

Les tableaux ou **array** sont des généralisations des vecteurs (1 dimension) et des matrices (2 dimensions), à des données des dimensions supérieures à 2.
Pour définir un tableau, il faut indiquer le nombre de valeurs pour chacune des dimensions du tableau au travers du paramètre **dim** de la fonction **array**. De même que pour les matrices, il est possible d'initialisé un tableau en donnant soit un vecteur de données, soit une valeur uniforme par défaut.
De même que pour les matrices :
- **dim** donne un vecteur qui contient les tailles des différentes dimensions du tableau
- **length** donne le nombre total d'élément du tableau (produit des dimensions)

Exemple 12

Dans l'exemple ci-dessous, on illustre la construction d'un tableau à 3 dimensions contenant 24 valeurs.

```
> arrNum <- array( data=3, dim = c(2,3,4) )
> print( arrNum )
, , 1
     [,1] [,2] [,3]
[1,]    3    3    3
[2,]    3    3    3

, , 2
     [,1] [,2] [,3]
[1,]    3    3    3
[2,]    3    3    3

, , 3
     [,1] [,2] [,3]
[1,]    3    3    3
[2,]    3    3    3

, , 4
     [,1] [,2] [,3]
[1,]    3    3    3
[2,]    3    3    3
```

3.2.5 Vecteurs vides, valeurs manquantes et non-nombres

Vecteurs vides À la question "qu'est ce qu'un vecteur vide ?", on peut trouver deux réponses. Soit il s'agit d'un vecteur qui ne contient aucun élément, soit d'un vecteur qui contient des éléments vides. Et bien tout cela est possible en R.

Remarque 5 - Vecteur de taille 0

L'instruction `c()` permet de déclarer un vecteur de taille 0, par exemple :

```
> a <- c()
> length( a )
[1] 0
```

Éléments vides Une variable, un vecteur ou une matrice peuvent ne pas contenir de valeur (casier vide). Dans ce cas, la valeur qui est affichée est **NA**. **NA** est une valeur particulière qui signifie qu'il n'y a pas de valeur (en anglais "*not a number*").
Lorsqu'une variable a été déclarée comme vecteur, il est possible de lui ajouter des éléments en initialisant l'élément désigné par un indice hors des limites. Si l'indice n'est pas contiguë aux éléments du vecteur, alors les éléments non remplis du vecteur seront initialisés avec la valeur **NA**.

```
> v <- 1:10
> v[15] <- 3
> v
[1]  1  2  3  4  5  6  7  8  9 10 NA NA NA NA  3
```

Dans cet exemple, vous noterez que la taille du vecteur s'est automatiquement adaptée au besoin d'indexation.
Il est possible de savoir si une variable ne contient pas de valeur (*i.e.* contient la valeur **NA**) en utilisant la fonction **is.na**().

Exemple 13 - NA

```
> v <- 1:3
> v[2]=NA # NA est une valeur qui peut etre  utilisee
> print( v )
[1]  1 NA 3
> is.na(v) # La fonction is.na teste si les elements sont NA (le resultat est un vecteur de logical )
[1] FALSE TRUE FALSE
> m <- matrix(ncol=2, nrow=2) # declaration d'un tableau sans initialisation de son contenu : il ne
    contient que des valeurs NA
> print( m )
     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA
> m[1,2] <- 3
> m
     [,1] [,2]
[1,]   NA    3
[2,]   NA   NA
> is.na(m) # application de is.na sur un tableau : donne le tableau des valeurs NA
      [,1]  [,2]
[1,] TRUE FALSE
[2,] TRUE  TRUE
```

Pour savoir si un vecteur, une matrice ou un **array** *n'a aucun élément* **NA**, *il est possible de faire l'opération suivante :*

```
> v <- array( data=3, dim = c(2,3,4) )
> sum( is.na(v) ) # retourne 0 car il n'y a aucun NA
[1] 0
> v[1, 2, 3:4] <- NA
> sum( is.na(v) ) # retourne 1 car il y a 2 NA
[1] 2
```

Une opération élémentaire qui cherchera à traiter une variable dont la valeur est **NA** aura pour résultat **NA**.

3.3 Structures de données avancées : les classes

En plus des données matricielles, R propose quelques structures de données un peu plus complexe pour manipuler des données également plus complexes. Ces structures de données sont souvent puissantes, et donc difficile à maîtriser.
Par opposition aux structures de données précédentes, qui sont des structures de données homogènes (c'est-à-dire que tous les éléments d'un vecteur ou d'une matrice sont du même type), certaines des structures de données avancées proposées par R sont hétérogènes, c'est-à-dire qui permet de contenir des variables de types différents. C'est le cas des **list** et des **data.frame**.
On commence par présenter les structures de liste, qui sont les structures de données les plus générales en R. Ensuite, nous présentons les structures de données de **factor**, plutôt particulière mais qui peut être rencontrée. Et finalement, on présente les **data.frame** qui sont les structures

de données très spécifiques aux données statistiques que traite le langage de programmation statistique qu'est R.
En fin de section, on revient sur la notion de classe. En plus de son mode, et de son mode de stockage, une variable pourra également avoir une classe qui décrit à quel type de structure de données complexe on a affaire.

3.3.1 Les listes (list)

Les listes sont les structures de données les plus génériques de R. Une liste est composée d'un ou plusieurs éléments. Chaque élément d'une liste peut avoir son propre type (classe, mode et mode de stockage) : ce peut être un scalaire, un tableau/vecteur/matrice ou une autre liste. Les différentes éléments de classe tableau peuvent être de taille différentes les unes des autres. Bref ! Une liste peut contenir n'importe quel autre type de données et les mélanger à souhait ! Cette propriété est notamment utilisée par certaines fonctions pour renvoyer des résultats complexes sous forme d'une seule variable.
En particulier, une liste peut contenir d'autres listes. On a ainsi des listes de listes ... et si on veut de liste. Rapidement, ce peut être difficile de s'y retrouvé puisque, les variables n'étant pas déclarées explicitement, il est toujours difficile d'être sûr de ce qu'il y a dedans (seule une utilisation rigoureuse des variables le permet).

L'accès aux éléments d'une liste se fait soit par l'indice dans la liste, en utilisant l'index entre double crochets [[...]] , soit en utilisant des noms pour chacun des éléments d'une liste. Pour accéder aux éléments d'une liste par le nom de cet élément, il faut utiliser le symbole $.
Pour connaître les étiquettes des éléments d'une liste, il suffit de faire appel à la fonction names.

Exemple 14 - Utilisation des listes

L'exemple ci-dessous illustre la définition et l'accès aux éléments de listes. Vous noterez que le print d'une liste affiche chaque élément en précisant d'abord son nom ou son numéro entre double crochets, puis l'élément lui même.
NB : L'exemple ci-dessous est un script (les commandes ne sont pas précédées du prompt. Les affichages sont illustrés après, comme le ferait l'exécution du script.

```
l<-list(1:4, 2:3, TRUE, "toto") # definition d'une liste a 4 elements
print( l [[1]] )

l2<-l[2:3]              # l2 est une sous-liste de l
print( l2 [[1]] )

# construction d'une liste contenant des listes
l4 <- list( l, l2)
l4 [[1]][[2]] <- 6      # modification du second element de la premiere sous-liste
print( l4 [[2]][[2]] )  # affichage du second element de la seconde sous-liste
```

affiche

```
[1] 1 2 3 4
[1] 2 3
[1] 3.4
```

Pour savoir si une variable est une liste (et non un vecteur), vous pouvez utiliser les fonctions is . list , et is . vector.

Il est également possible de donner des noms aux différents aux éléments d'une liste. Dans ce cas, la fonction **list** s'utilise de la sorte : **list** (**nom1**=**el1**,**nom2**=**el2**,...). Les noms permettent d'accéder à chaque élément de la liste à l'aide de son nom précédé du signe $.

Exemple 15 - Utilisation des noms d'éléments

```
#creation d'une liste avec noms
li<-list(num=1:5,y="couleur",a=TRUE)

print( li$num )    #Acces aux element par leur nom
print( li [[1]] )  #Acces aux element par leur indice

#ajout d'un element a une liste
li$new.elem <- as.integer(3)

print( li )        # visualisation de la liste
```

affiche :

```
[1] 1 2 3 4 5
[1] 1 2 3 4 5
$num
[1] 1 2 3 4 5
$y
[1] "couleur"
$a
[1] TRUE
$new.elem
[1] 3
```

3.3.2 Les facteurs (factor et ordered)

Les facteurs sont des vecteurs ayant 2 attributs supplémentaires permettant la manipulation de données qualitatives : les **labels** et les **levels**. Les **levels** décrivent les différentes valeurs que peuvent prendre les éléments du vecteur. La structure de données retient également leur nombre. Les facteurs forment une classe d'objets et bénéficieront donc de traitements particuliers pour certaines fonctions par exemple **plot**(). Il y a deux types différents de facteurs :

– les facteurs non ordonnés (mâle, femelle) appelés **factor**
– les facteurs ordonnés (riche, aisé, pauvre) appelés **ordered**

L'utilisation des facteurs peut conduire à des comportements étranges si on ne les souhaite pas explicitement utiliser leurs propriétés. Il est donc recommandé de les utiliser avec prudence.

Exemple 16 - Facteurs

L'intérêt de l'exemple ci-dessous est d'illustrer la construction explicite d'un facteur et la notion des niveaux. On constate que le vecteurs a 4 éléments (dont des répétitions) mais seulement deux niveaux, tous distincts.

```
> f<-factor( c(10,10,13,13) ) # facteur construit a partir d'un vecteur
> print(f) # affichage du facteur
 [1] 10 10 13 13
Levels: 10 13
```

3.3.3 Les données tabulaires (data.frame)

Il est utile de dire quelques mots des data.frame puisque c'est ce type de données qui est récupéré par la fonction read.table (permettant la lecture de fichiers de données).
Les data.frame sont des structures de données en tableau à 2 dimensions dont les colonnes peuvent être de types hétérogènes. On pourra noter que ces données sont plus structurées que les listes puisqu'il n'est pas possible de tout mélanger.
Il est possible de donner des noms aux colonnes et aux lignes de ce tableau (comme les listes). L'accès aux éléments peut se faire alors comme pour un tableau (par index) ou en utilisant également les noms des colonnes (uniquement les colonnes [4]). De même que pour les listes, l'accès aux colonnes par noms est possible en utilisant la notation $.

Exemple 17 - les data.frame

```
v1=sample(1:4,10,rep=T) #vecteur contenant 10 elements (valeurs prises dans 1:4, avec repetition
    possible )
v2=sample(LETTERS,10) #vecteur contenant 10 elements (valeurs prises dans les lettres majuscules)
v3=runif(10)          #vecteur contenant 10 elements ( distrib uniforme)
v4=rnorm(10)          #vecteur contenant 10 elements ( distrib normale)

xx=data.frame(v1,v2,v3,v4) # creation du data.frame
print(xx)                  # affichage

if( is.data.frame( xx ) ) {
  print( xx$v1 )           # acces a une colonne par nom
  print( xx[1] )           # acces a une colonne par index
  print( xx[9,] )          # acces a une ligne
  print( xx[ c(2, 4, 6), ] )# acces a plusieurs lignes
  print( xx[1,3] )         # acces a un element par index
  print( xx$v1[3] )        # acces a un element par nom et index
}

#ajout de noms aux lignes (attention a la taille du vecteur)
rownames( xx ) <- c("L1","L2","L3","L4","L5","L6","L7","L8","L9","L10")

# recuperation d'une matrice (sans noms de colonnes)
m<-unname(as.matrix(xx))
```

affiche :

```
   v1 v2         v3          v4
1   3  X 0.97540708 -0.03517206
2   2  W 0.36233840  2.18431102
3   4  U 0.07116431  0.33586812
4   2  T 0.53673750  0.44167623
5   3  R 0.98077959 -0.71284531
6   2  K 0.32014701 -1.61995982
7   1  B 0.24713918 -0.16623567
8   3  F 0.41825691 -1.59483593
9   4  M 0.91991287  1.37746367
10  1  V 0.59848708 -0.65977459
 [1] 3 2 4 2 3 2 1 3 4 1
```

4. En tout cas, je n'ai pas réussi avec des noms de lignes

```
   v1
1   3
2   2
3   4
4   2
5   3
6   2
7   1
8   3
9   4
10  1
  v1 v2        v3       v4
9  4  M 0.9199129 1.377464
  v1 v2        v3         v4
2  2  W 0.3623384  2.1843110
4  2  T 0.5367375  0.4416762
6  2  K 0.3201470 -1.6199598
[1] 0.975407
[1] 4
```

3.3.4 Classes et "coercions"

La classe d'une variable est une information complémentaire à celle de son mode. Pour connaître la classe d'une variable, il faut faire appel à la fonction **class** :
- pour les vecteurs : **class** et **mode** retournent la même chose (**numeric**, **character**, **logical**, etc.)
- pour les matrices et les tableaux : **class** retourne **matrix** ou **array** et **mode** retourne **numeric**, **character**, **logical**, etc.
- pour les structures de données complexes : **class** et **mode** retournent **data.frame**, **factor** ou **list**

De même qu'il est possible de modifier le mode d'une variable (en en modifiant le contenu informationnel !), il est également possible de transformer sa classe. Pour cela, il est possible d'utiliser les fonctions génériques **as.vector**, **as.factor**, **as.list**, mais attention à leurs résultats. **Globalement, la coercion des classes d'une variable est peu recommandé !**

Exemple 18 - conversion en numérique des facteurs

Les tentatives ci-dessous pour la conversion d'un facteur en vecteur de nombre montre que ce n'est pas trivial :

```
> x<-factor(c(10,10,13,13)) #definition du facteur
> as.numeric(x) # l'utilisation de as.numeric traduit le x comme une suite de references internes aux
    levels
[1] 1 2 3
> as.numeric(unclass(x)) # le unclass ne resoud pas le probleme
[1] 1 2 3
> as.vector(x) # l'utilisation de as.vector traduit n'importe quel variable sous la forme de texte !
[1] "10" "10" "13" "13"
> as.numeric(as.vector(x)) #enfin la bonne solution : on transtype deux fois !
[1] 10 11 13
```

Si pour une raison ou une autre vous avez besoin de transformer une liste en vecteur, vous pouvez le faire en utilisant la fonction **unlist** (...) . La fonction **unname**(...) supprime les noms associés aux éléments d'une liste (ou d'un vecteur). L'effet de la transformation d'une liste en un vecteur est d'aplatir entièrement les données dans un vecteur. En particulier, les listes contenant des vecteurs ou d'autres listes vont donner une structure de données linéaire : chaque élément sera ajouté au même vecteur. De plus, la conversion se fait en suivant la règle du type le plus "fort".

Exemple 19 - Suppression de la structure de liste

suite de l'exemple 3.3.1 :

```
# transformation d'une liste en vecteur
vec<-unlist(li)
print( vec )

#suppression des noms des elements
vec <- unname( vec )
print( vec )
```

affiche :

```
     num1      num2      num3      num4      num5         y         a
      "1"       "2"       "3"       "4"       "5" "couleur"    "TRUE"

[1] "1"       "2"       "3"       "4"       "5"       "couleur" "TRUE"
```

3.4 Les opérateurs élémentaires sur les variables

On complète cette présentation des variables en donnant succinctement les opérateurs usuels entre les variables des différents types.

3.4.1 Opérateurs sur les scalaires

Les opérateurs arithmétiques usuels sont les suivants :
- + : addition
- – : soustraction
- * : multiplication
- / : division réelle, le résultat est toujours un **numeric**, même si vous diviser des **integer**
- %/% : division entière, %% modulo (reste de la division entière)
- ^ : puissance, le résultat est toujours un **numeric**. il est possible d'élever un nombre avec un puissance non-entière

Opérateurs de comparaison :
- > : supérieur à
- < : inférieur à
- >= : supérieur ou égal à
- <= : inférieur ou égal à
- == : égal à
- != : différent de

Opérateurs logique :
- & : "et" logique

– | : “ou” logique
– ! : “non” logique
Il est bien évidemment possible de combiner plusieurs opérateurs arithmétiques sur une même ligne en utilisant également des parenthèses avec les ordres de priorité usuels.

```
a <- 10
b <- 10
t <- (a*30 +4) / (b-56)
```

! Attention ! - Utilisation du “.”

Le “.” ne sert pas d’opérateur. Il est généralement utilisé pour séparer des mots dans des noms de variables ou de fonctions (e.g. data.frame(...))

3.4.2 Opérations sur les vecteurs

On peut distinguer deux types d’opérations sur les vecteurs pour lesquels les opérateurs (arithmétiques et logiques) sur les scalaires peuvent être étendus :
– l’opération d’un scalaire sur un vecteur : les opérations s’appliquent individuellement à chaque élément de ce vecteur.
– une opération entre deux vecteurs : l’opération est appliquée dimension par dimension.
Le résultat d’une opération arithmétique ou logique entre deux vecteurs de taille n sera également un vecteur de taille n. Il est indispensable dans les deux vecteurs est la même taille pour que l’opération soit valide. Dans le cas contraire, une erreur se produira.

Exemple 20 - Opération sur les vecteurs

Dans ce premier exemple, on illustre des opérations d’un scalaire sur un vecteur.

```
> v <- c(45, 23, 3)
> v <- 1+v
> print( v )
[1]  46  24  4
> print( v * 4 )
[1]  184  96  16
```

Dans ce second exemple, on illustre des opérations entre vecteurs

```
> a <- c(45, 23, 3)
> b < c(1, 2, 3)
> v <- a+b
> print( v )
[1]  46  25  6
> d <- c(2, 3) # vecteur de taille 2 seulement !
> v <- a - d #Une erreur se produit :
Message d'avis :
In a - d :
  la taille d'un objet plus long n'est pas multiple de la taille d'un objet plus court
> v1 <- c(45, 23)
> v2 <- c(45, 3)
> v1 == v2 # comparaison des valeurs 2 a 2 : le resultat est une matrice de booleens
[1]  TRUE FALSE
```

On peut noter en particulier que l'opération de multiplication de deux vecteurs ne correspond pas à l'opération usuelle dans l'algèbre vectorielle, *c.-à-d.* le produit scalaire ou plus généralement un produit de matrices.
Pour faire le produit de matrice, il faut utiliser l'opérateur **%∗%**. Ainsi, pour faire le produit scalaire de deux vecteurs, il faut écrire les instructions suivantes :

```
> v <- c(3, 4, 5)
> w <- c(4, 5, 6)
> v%*%w
     [,1]
[1,]   62
```

3.4.3 Opérations sur les matrices

Les opérations arithmétiques et logiques (entre matrices ou entre un scalaire et une matrice) fonctionnent de la même manière que pour les vecteurs.
En particulier, l'opérateur ∗ n'est pas un opérateur de multiplication de matrice. La multiplication de matrice est obtenue par l'opérateur \%∗\% (Attention aux tailles des matrices).
La fonction **t**() permet de transposer une matrice.

Exemple 21 - Opérations sur les matrices

```
> m1 <- matrix(1:12, ncol=4, nrow=3)
> m2 <- matrix(1:12 + 3, ncol=4, nrow=3)
> m1 %*% t(m2)
     [,1] [,2] [,3]
[1,]  232  254  276
[2,]  266  292  318
[3,]  300  330  360
```

3.4.4 Opérations sur les chaînes de caractères

On ne présente ici que deux opérations de traitement de chaînes de caractères (extraction d'une sous-chaîne, et concaténation de deux chaînes), l'opération de comparaison de chaînes de caractères, et la fonction qui permet de récupérer la longueur d'une chaîne de caractères. D'autres fonctions avancées permettent de traiter les chaînes de caractères même si R n'est pas l'outil adéquat pour ce genre de traitement automatique.

- L'extraction d'une sous-chaîne de caractères à partir d'une chaîne de caractères s'effectue à l'aide de la fonction **substr**(), dont la syntaxe est la suivante : **sCC <−substr(CC, id.debut, id.fin)**
- La concaténation de deux chaînes de caractères s'effectue à l'aide de la fonction **paste**, dont la syntaxe est la suivante : **cCC <−paste(CC1, CC2, sep="...")**. Le paramètre optionnel **sep** définie le caractère de séparation des deux chaînes concaténées (par défaut, l'espace).
- La comparaison (exacte) de deux chaînes de caractères s'effectue à l'aide de l'opérateur de comparaison =='
- La fonction **nchar**() donne le nombre de caractères d'une chaîne de caractères, c'est-à-dire sa longueur.

Exemple 22 - Chaînes de caractères

```
> mot1 <- "foo" #comparaison de chaines de caracteres
> mot2 <- "bar"
> mot3 <- "bar"
> mot1==mot2
[1] FALSE
> mot1==mot3
[1] FALSE
> mot2==mot3
[1] TRUE
> mot <- paste(mot1, mot2) #concatenation de chaines de caracteres
> print(mot)
[1] "foo bar"
> mot <- paste(mot1, mot2, sep="") #concatenation avec separateur vide
> print(mot)
[1] "foobar"
> substr(mot, 3, 5) #extraction d'une sous-chaine de caracteres
[1] "oba"
> nchar(mot) # longueur d'une chaine de caracteres
[1]  6
```

4 Les structures de contrôle dans R

4.1 Les blocs d'instructions

Un bloc d'instruction est un ensemble d'instructions délimitées par des accolades :
– une accolade ouvrante pour indiquer le début bloc
– une accolade fermante pour indiquer la fin le bloc
L'indentation du contenu d'un bloc est optionnel, mais **indispensable** pour une meilleur lisibilité du programme. Il faut donc écrire :

```
{
  a <- 10
  b <- 20
  {
    c <- a*b
  }
}
```

et non pas (illisible)

```
{
a <- 10
b <- 20
{
c <- a*b
}
}
```

4.2 Structure conditionnelle : if / else

La syntaxe formelle d'un if / else est la suivante :

```
if ( condition ) {
  # bloc d' instructions  1
```

```
}else{
  # bloc d' instructions  2
}
```

La condition est notée entre parenthèses. Elle doit contenir une variable scalaire de type `logical` ou le résultat d'une opération qui retourne un `logical`. Si la variable n'est pas de type `logical`, elle est d'abord *implicitement* transformée en type `logical` (voir `as.logical`). En particulier, il est donc possible de passer un nombre dans la condition, alors `0` veut dire **FALSE** et tout autre nombre que `0` veut dire **TRUE**.
Il est fortement conseillé d'utiliser le même positionnement des crochets que ce qui est illustré :
– le crochet ouvrant se trouve à côté de la parenthèse fermante de la condition du `if`, ou juste à côté du `else`,
– si il n'y a pas de `else`, le crochet fermant est seul sur sa ligne (exemple ci-dessous)
– si il y a un `else`, les accolades fermantes pour le bloc 1 et le bloc 2 sont verticalement alignées avec le "~**i**" du `if`.
– le `else` doit **impérativement** être à côté de l'accolade fermante du `if` (sinon, une erreur se produit).
La clause `else` est facultative. L'écriture ci-dessous est tout à fait valide :

```
if ( x<0 ) {
  x <- -x
}
print(x)
```

Les `if/else` peuvent être imbriquées comme suit :

```
if (cond1) {
  # bloc d' instructions  1
} else if (cond2) {
  # bloc d' instructions  2
} else if (cond3) {
  # bloc d' instructions  3
} else {
  # bloc d' instructions  4
}
```

Exemple 23 - Structure if/else

En fonction de la valeur qui sera saisis par l'utilisateur (cf. instruction `scan()`) le programme affichera des messages différents :

```
1 a <- scan()
2
3 if ( a==10 ) {
4   print("coucou")
5 } else if ( a < 20 ) {
6   print("toto")
7 } else {
8   print("bar")
9 }
```

4.3 Structure répétitive : `while`

On pourra noter qu'il existe uniquement le `while` et qu'il n'existe pas de structure du type `REPETER ... JUSQU'À`.

4.3.1 Utilisation d'un while

La syntaxe formelle du while est la suivante :

```
while( condition ) {
  # instructions
}
```

L'utilisation de la condition est similaire au if :
- la condition est notée entre parenthèses,
- la condition attendue est de type logical,
- le crochet ouvrant est à côté de la condition, et le fermant est seul sur sa ligne, aligné verticalement avec le "w" du while

Exemple 24 - Exemple d'utilisation du while

Ce programme affiche la séquence des entiers de 1 à 9 :

```
1 x<-1
2
3 while( x<10 ) {
4   print(x)
5   x <- x+1
6 }
```

Exemple 25 - Parcours d'un tableau

Cet exemple illustre le parcours d'un tableau en R en en réalisant l'affichage. Cet exemple est bien évidemment à connaître parfaitement...

```
1 v <- c(2, 34, 6.78, 9.2)
2 i <- 1
3 while( i<=length(v) ) {
4   print( v[i] )
5   i <- i+1
6 }
```

Exemple 26 - Parcours d'une matrice

Pour un parcours d'une matrice, on illustre l'imbrication de deux boucles. De nouveau, il s'agit d'un exemple très classique à connaître. Dans ce cas, le programme somme tous les éléments du tableau.

```
1 m <- matrix(1:12, ncol=3, nrow=4)
2 i <- 1 #index de la ligne
3 sum <- 0
4 while( i<=nrow(m) ) {
5   j <-1 #index de la colonne
6   while( j<=ncol(m) ) {
7     sum <- sum + m[i,j]
8     j <- j+1
9   }
```

```
  i <- i+1
}
print( sum )
```

J'attire l'attention de nouveau sur l'indentation du code qui facilite la lecture.

4.3.2 Arrêts (break) et sauts (next) de boucle

Le déroulement normal d'une boucle peut être interrompu à l'aide des instructions suivantes :
– **break** : sort immédiatement de la boucle,
– **next** : arrête le traitement de l'itération courante pour revenir directement en haut du **while**.

Exemple 27 - Exemple de parcours d'un tableau avec break et next

```
x <- c(37, -6, 41, 0, 10)
i <- 1
while( i < length(x) ) {
  if ( x[i] == 0 ) {
    break
  }
  if ( x[i] < 0 ) {
    i <- i+1 #Attention : ne pas oublier d'incrementer i avant le next !!
    next
  }
  print( x[i] )

  i <- i+1
}
```

Ce programme affiche :

```
[1] 37
[1] 41
```

4.4 Structure répétitive : for

La structure **for** est une structure qui permet de boucler en faisant évoluer une variable, appelée itérateur, sur un ensemble de valeurs données. La syntaxe d'un **for** est la suivante :

```
for( i in v ) {
  #instructions pour traiter i
}
```

où :
– **v** est un vecteur définissant les valeurs à parcourir (valeurs successivement prises par la variable **i**,
– **i** est l'itérateur : il prend successivement, à chaque tour de boucle, les valeurs du vecteur **v**

Exemple 28

```
for( i in c(3.4, 5.6, 7) ) {
  print(i)
}

for( str in c("ceci", "est", "un", "for")
{
  print(str)
}
```

Cet exemple affiche :

```
[1]  3.4
[1]  5.6
[1]  7
[1] "ceci"
[1] "est"
[1] "un"
[1] "for"
```

L'exemple ci-dessous peut être utilisé pour traiter plusieurs fichiers :

```
#construction d'une liste de fichiers a traiter
FICHIER <- c("Aedes.dat", "Culex.dat", "Anopheles.dat")
#traitement 1 a 1 des fichiers
for (FILE in FICHIER) {
  DF <- read.table(FILE, header = TRUE) # ouverture du fichier FILE
  mod <- lm(prevalence ~ ... , data = DF)
  print(summary(mod))
}
```

Le cas le plus classique d'utilisation est un itérateur qui parcourt une séquence d'entiers.

Exemple 29 - Parcours d'une séquence d'entiers

```
for( it in 1:100 ) {
  print(it)
}
```

Ce programme a une équivalence directe avec un **while** *:*

```
it <- 1
while( it <=100 ){
  print(it)
  it <- it+1
}
```

En programmation, ceci correspond à un besoin très courant : les parcours de vecteurs et de matrices. Tous les parcours de tableaux ou de matrice peuvent être simplement écrits avec des **for**.

Exemple 30 - parcours d'un tableau

```
v <- c(2, 34, 6.78, 9.2)
for( i in 1:length(v) ) {
  print( v[i] )
}
```

Exemple 31 - parcours d'une matrice

```
m <- matrix(1:12, ncol=3, nrow=4)
for( i in 1:nrow(m) ) {
  for( j in 1:ncol(m) ) {
    print( m[i,j] )
  }
}
```

Exemple 32 - compte le nombre de voyelles d'un mot

```
mot <- "coucou"
nb <- 0
for( i in 1:nchar(mot) ) {
  char <- substr(mot, i, i)
  if( char == "a" || char == "e" || char == "i" || char == "o" || char == "u") {
    nb <- nb + 1
  }
}
print( nb )
```

Remarque 6 - break et next dans les for

Les instructions **break** et **next** peuvent également être utilisées dans une boucle **for**.

5 Les fonctions en R

La plupart des instructions de R sont des appels de fonction. Lors des appels, les paramètres de ces fonctions sont passés entre parenthèses.

5.1 Définition d'une fonction

Pour définir la fonction, il faut utiliser la syntaxe suivante pour le mot clé "**function**" :

```
MaFonction <- function (p1, p2, p3) {
   #definition de la fonction
  return(4.5)
}
```

– **function** est un mot clé pour définir une fonction,
– le nom de la fonction est donné à gauche de la flèche, la fonction est "comme" une variable définie par l'affectation de la fonction,

– les paramètres formels sont donnés entre parenthèses après le mot clé "function" : les paramètres sont indiqués uniquement en donnant leur nom formel. En particulier, leur type n'est pas précisé. Ceci signifie que dans la fonction ne vous ne pouvez faire aucune présupposition sur le type d'une variable.
– le corps de la fonction est décrit par le programmeur entre les accolades qui définissent un bloc.

Contrairement aux autres blocs, toutes les variables définies dans un bloc de fonction sont **locales** à la fonction. C'est-à-dire que les variables définies à l'intérieur de la fonction ne peuvent être directement utilisées à l'extérieur de cette même fonction.

Exemple 33 - Variable locale d'une fonction

```
1  rm( list =ls() )
2
3  a <- c(4, 5, 6)
4
5  f <- function() {
6    a <- 3
7    print( a )
8  }
9
10 f();
11 print( a );
```

L'affichage de ce programme est le suivant :

```
[1] 3
[1] 4 5 6
```

Dans cet exemple, on commence par définir une variable **a** *et une fonction* **f** *dans la session R. La fonction* **f** *modifie la valeur d'une variable locale* **a** *et l'affiche. Cette variable n'est pas la même que celle qui a été définie dans la session R de sorte que si on appelle la fonction puis qu'on affiche* **a** *(dans la session R) alors l'affichage obtenu est l'affichage de la variable* **a** *d'origine. En résumé, la modification de* **a** *dans la fonction est sans effet pour* **a** *dans la session.*

L'instruction **return** indique la fin immédiate de la fonction, et elle prend en paramètre une valeur qui sera retournée par la fonction. Le type de l'objet retourné par votre fonction dépend de la valeur qui est donnée au **return**. Ce type n'est pas fixé.

Exemple 34 - Passage de paramètres et instruction return

```
1  #declaration de la fonction
2  MaFonction <- function(p1)
3  {
4    a <- 3*p1
5    return(a)
6  }
7
8  #appel de fonction
9  v <- Mafonction(34)
10 print(v)
```

On notera au passage qu'il n'y a pas de différence entre une fonction et une procédure en R. Une procédure sera simplement une fonction qui ne retourne rien (par exemple la fonction **print**).

5.2 Appels de fonction

Il y a deux façons de faire le lien entre les paramètres passés à une fonction et ses paramètres formels :
– par la position des paramètres,
– en nommant directement le paramètre formel qui doit être assigné.
Vous n'êtes pas obligé de donner une valeur pour tous les paramètres d'une fonction. S'il manque des paramètres importants, alors une erreur sera générée, et si c'est un paramètre optionnel, alors la fonction aura prévu une valeur par défaut.
La récupération de la valeur retournée par une fonction peut se faire en utilisant l'affectation. Il ne peut être fait aucune présupposition sur le type de la valeur retournée par une fonction.

Exemple 35 - appel de la fonction de l'exemple précédent

```
> a <- MaFonction ( 21 )
> print( a )
[1]  63
> var <- 10
> a <- MaFonction( var )
> print( a )
[1]  30
> a <- MaFonction( 1:5 )
> print(a)
[1]  3  6  9  12  15
> a <- MaFonction( "coucou" )
Erreur dans 3 * p1 : argument non numerique pour un operateur binaire
```

Remarque 7

Une fonction doit avoir été définie "avant" son utilisation (c'est-à-dire plus haut dans le fichier du programme).

6 Les entrées-sorties

6.1 Affichage à l'écran, fonctions print et cat

La fonction **print** affiche le contenu d'une variable sous la forme d'une matrice, tandis que la fonction **cat** affiche une chaîne de caractères ou d'une variable sans décorum.

Exemple 36 - print et cat

```
> m <- matrix(1:12, ncol=4, nrow=3)
> cat(m)
1 2 3 4 5 6 7 8 9 10 11 12>
> print(m)

  [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> str <- "coucou"
> print(str)
[1] "coucou"
> cat(str)
coucou>
```

Dans cet exemple, vous pouvez noter qu'à la fin de l'affichage de **cat** *il y a un ">". Ceci s'applique par le fait que* **cat** *ne fait de retour à la ligne à la fin de son affichage. Pour faire un retour à la ligne, il faut ajouter* **cat**`("\n")`

Exemple 37 - print et cat

```
> cat("coucou\n");
coucou
> cat("coucou")
coucou>
> cat(m);cat("\n") # le ";" permet de separer deux instructions ecrites sur une meme ligne
1 2 3 4 5 6 7 8 9 10 11 12
>
```

En fonction de vos besoins, vous utiliserez soit la fonction **print** ou la fonction **cat**. La fonction **cat** est plus adaptée pour écrire des messages à destination de l'utilisateur du programme car les chaînes de caractères ne sont pas mises entre guillemets et il n'y a pas l'index du vecteur en début de ligne.

6.2 Lecture clavier : fonction scan

La fonction **scan** demande une saisie au clavier à l'utilisateur. La syntaxe de cette instruction est la suivante :

```
A <- scan()
```

Cette instruction permet de demander à l'ordinateur d'attendre une saisie clavier de l'utilisateur, et de mettre le résultat de la saisie dans la variable **A**.
La fonction **scan** ne permet pas de récupérer des chaînes de caractères, mais uniquement des nombres.
Comme toutes les variables sont des vecteurs, la fonction **scan** attend la saisie de plusieurs valeurs par l'utilisateur. Lors de la saisie, l'ordinateur attend la saisie d'une premier nombre, l'utilisateur effectue la saisie et la valide par la touche `Entrée`. Ensuite, l'ordinateur attend une

autre saisie jusqu'à ce que l'utilisateur valide sans rien saisir (c'est-à-dire qu'il fait deux fois `Entrée`).

Pour faire une saisie clavier d'un scalaire, il est préférable de préciser qu'on ne souhaite récupérer qu'une unique valeur :

```
A <- scan(n=1)
```

6.3 Lecture/écriture d'un fichier

Dans de nombreux cas, les données que l'on souhaite analyser proviennent de sources externes sous forme de fichiers. Les fichiers ASCII servent, le plus souvent, pour échanger des données brutes. Un fichier ASCII est un fichier qui peut être ouvert par un éditeur de texte simple (*p.ex.* notepad).
Les données sous forme matricielle sont généralement organisées dans un fichier ASCII en ligne et pour chaque ligne, les valeurs de chaque colonne sont séparées par un caractère spécial (une tabulation, une virgule, ...). Ce format de fichier est appelé CSV (en anglais *Comma Separated Values*). Les fichiers CSV peuvent être ouverts et créés à partir d'une feuille de calcul Excel ou OOCalc.
La fonction `read.table()` de R permet d'ouvrir un fichier CSV, et la commande `write.table()` permet d'enregistrer des données tabulaires sous la forme d'un fichier CSV. Le type des données qui sont utilisés par ces fonctions est `data.frame` (*cf.* "section autres structures de données").
Pour plus d'informations sur ces fonctions, consultez l'aide.

6.4 Connexion à une base de données avec RODBC

Dans le cadre du cours d'Agrocampus-Ouest, on illustre également l'utilisation de bases de données comme possibilité d'entrée/sortie pour les données. Les requêtes SQL sur une base de données permettent d'importer des données dans des variables, et il est également possible d'enregistrer des résultats dans la base de données.

6.4.1 Installation et chargement du package RODBC

Plusieurs packages sont disponible dans R pour accéder à des bases de données. Dans notre cours, nous utiliserons la package RODBC disponible facilement sous Windows.
L'utilisation d'un package nécessite son installation (une seule fois par ordinateur).Ceci peut se faire simplement à partir de R avec la commande ci-dessous directement dans la console :

```
install.packages("RODBC")
```

Suivez ensuite les instructions : 1) choisir un site de téléchargement, 2) attendre la fin de l'installation et c'est tout !

6.4.2 Ouvrir une connexion à une base de données dans R

Avant d'exécuter une requête il faut établir une connexion à la base de données :
Pour accéder à une base de données ODBC :

```
library(RODBC)
Connex=odbcConnect(dsn="nom-source-donnees",uid="nom-utilisateur",pwd="mot-de-passe")
```

ou, pour accéder directement à un fichier Access :

```
library(RODBC)
Connex=odbcConnectAccess("nom-fichier-mdb")
```

Quelques explications
- **library** (**RODBC**) : permet de charger le package RODBC pour disposer des fonctions d'accès aux bases de données. Ce package doit préalablement avoir été installé. Le chargement du package est nécessaire une seule fois par session R.
- **dsn** : c'est le nom qui décrit comment accéder à votre base de données. Ce peut être un nom de fichier dans le cas des fichiers Access (sans l'extension .mdb) ou bien des adresses web si le serveur est distant,
- **uid** : si nécessaire, donnez le nom de l'utilisateur qui peut se connecter à la base de données,
- **pwd** : si nécessaire, donnez le mot de passe de l'utilisateur.

On récupère les paramètres de la connexion dans une variable **Connex** qui sera ensuite réutilisée comme paramètre des autres fonction du package.
En fin d'utilisation de la connexion à la base de données, il ne faut pas oublier de fermer celle-ci. Pour cela il existe deux fonctions :
- **odbcClose**(**Connex**) : on doit préciser la connexion à fermer, ici **Connex**
- **odbcCloseAll**() : utile si vous avez plusieurs connexions à des bases de données ouvertes, qui les fermera toutes sans avoir à préciser le nom de la connexion

Si vous souhaitez accéder à une base de données en ligne, il faut au préalable "enregistrer" cette base dans les sources de données ODBC.
Pour créer une source de données ODBC sous windows, il suffit d'aller dans Panneau de configuration, puis Système et maintenance, Outils d'administration et Sources de données (ODBC).

1. Tout d'abord, cliquez sur `Ajouter` et sélectionnez le pilote correspondant à votre base de données (il se peut parfois qu'il ne soit pas installé, il faut alors le télécharger),
2. Ensuite vous devez remplir la fenêtre suivante. Remplissez les différents champs. `Data source name` sera le nom à indiquer dans la fonction R.
3. La source de données (ODBC) est maintenant créée. Appliquez et quittez.

6.4.3 Exécuter des requêtes SQL dans R

Une fois la connexion ouverte, il est possible d'exécuter des requêtes SELECT, INSERT, UPDATE, ... En fait, toutes les manipulations de la base de données sont possibles.
Toutes ses requêtes utilisent la fonction **sqlQuery** qui prend en paramètre :
- la connexion à la base de données (identifie la base de données à interroger),
- une chaîne de caractères qui décrit la requête SQL.

```
resultat=sqlQuery(Connex,"SELECT * from mytable")
```

Pour les requêtes de type SELECT, les résultats est retourné sous forme de **list** . Elles peuvent donc être exploitées facilement dans R.

6.4.4 Construction d'une requête SQL

Il sera plus intéressant de construire une requête à partir de la valeur d'une variable. Pour cela, le problème consiste à construire une chaîne de caractères correspondant à la requête à faire à partir de variables.
On rappelle que la fonction **paste** permet de concaténer des chaînes de caractères.

```
library (RODBC)
Connex=odbcConnect(dsn="nom-de-source-de-donnees",uid="nom-d-utilisateur",pwd="mot-de-
    passe")
req=paste("SELECT champ1 FROM table WHERE champ2 ='",variable_R,"'",sep="")
resultat=sqlQuery(Connex,req)
odbcClose(Connex)
```

Exemple 38 - Transformer un vecteur R en chaîne de caractères pour une insertion multiple

On crée une fonction qui formatte les données comme on le souhaite, ici 2 valeurs texte et 1 valeur numérique

```
insertion_multiple=function(vect)
{
  paste("(",toString(c(encodeString(c(vect[1],vect[2]),quote="'"),vect[3])),")",sep="")
}
```

On crée un jeu de données pour tester, sur lequel on applique la fonction

```
nom=c("Julie","Pierre","Jean","Marie","Vincent","Claire","Sandra","Thomas")
sexe=c("F","M","M","F","M","F","F","M")
age=c(15,15,16,17,14,15,17,16)

mat=cbind(nom,sexe,age)

insertMultipl=toString(apply(mat,1,virg))
```

L'affichage de **insertMultipl** *donnera maintenant :*

```
[1] "('Julie', 'F', 15), ('Pierre', 'M', 15), ('Jean', 'M', 16), ...
```

On réinjecte **insertMultipl** *dans la requête INSERT à exécuter (*req*) :*

```
req=paste("INSERT INTO table (nom,sexe,age) VALUES ",insertMultipl,sep="")
 library (RODBC)
Connex=odbcConnect(dsn="nom-de-source-de-donnees",uid="nom-d-utilisateur",pwd="mot-de-
    passe")
sqlQuery(Connex,req)
odbcClose(Connex)
```

6.4.5 Utilisation de RODBC pour le chargement de fichiers Excel

Sous Windows, il est également possible d'utiliser la library RODBC pour charger directement les données depuis un fichier Excel. A priori, il faut que le fichier de données soit bien structurés pour que cela fonctionne bien : les données doivent être organisées en colonnes avec la première ligne qui contient les noms des colonnes.
Dans l'exemple ci-dessous, le fichier `don_orange0506.xls` contient une feuille nommée `Feuil1`. Ces trois lignes permettent de récupérer les données contenues dans cette feuille dans une variable nommée **data** de type **data.frame**.

```
connect <- odbcConnectExcel("don_orange0506.xls")
data <- sqlFetch(connect, "Feuil1")
odbcClose(connect)
```

Si nécessaire, il est possible d'utiliser une requête pour sélectionner la plage de données qui sera récupérée :

```
fin <- file.choose()
connect <- odbcConnectExcel(fin)
data <- sqlQuery(connect, "select * from [Feuil1$D9:I15]")
odbcClose(connect)
```

Dans l'exemple précédent, on a utilisé l'instruction **file.choose**() qui permet à l'utilisateur de choisir son fichier à l'aide d'une fenêtre classique. Le fichier peut être utilisé ensuite pour récupérer les données.

Pour enregistrer des données dans un fichier Excel, il faut commencer par ouvrir le fichier en autorisant des modifications avec la fonction **odbcConnectExcel** (par défaut : **readOnly** = **TRUE**, on va le forcer à faux).
La ligne 3, permet d'insérer les données du **data.frame** nommé **data** dans une feuille nommée du fichier Excel `New`. Si cette feuille n'existe pas, elle sera créée.

```
connect <- odbcConnectExcel("fichier.xls", readOnly = FALSE)
data <- unique(data[,1:2])
sqlSave(connect, data, tablename = "New", rownames = FALSE)
odbcClose(connect)
```

6.5 Les graphiques

Les graphiques font partie également partie des entrée-sortie. C'est un type de sortie très fréquemment utilisé en R. Les fonctions qui sont présentées ci-dessous ne nécessite d'ajouter aucun *package* particulier, ce sont des fonctions de base de R. En revanche, si vous souhaitez réaliser des graphiques avancés, il peut être utile d'aller chercher d'autres *packages*. Personnellement, je recommande l'utilisation du *package* `ggplot2`[5] pour faire des affichages riches ... mais son utilisation n'est pas nécessairement immédiate !

6.5.1 Affichage d'un jeu de données numériques

Admettons que vous disposer d'un vecteur **x** qui contient des valeurs (on se restreint ici à des valeurs numériques). Il existe plusieurs manière d'afficher ce jeux de données :
– les histogrammes,
– les boîtes à moustache,
– les camemberts,
– les graphiques en barres.
L'exemple ci-dessous illustre les commandes pour la création de ces type de graphique, et la Figure 2 illustre les résultats d'affichage :

```
1  # creation de jeux de donnees aleatoires
2  x <- runif(100)
3  y <- 0.7*runif(100)+0.2
4
5  # construction de la l'histogramme des valeurs de x
6  hist(x, breaks=10, main="Mon histogramme")
7
8  # boite a moustache des deux jeux de donnees (on peut se contenter d'un vecteur en entree)
9  boxplot(x, y, names=c("x", "y"), main="Mes boites a moutaphes", ylab="Poucentages")
10
11 # construction d'un camembert
12 vec <- c(23, 45, 5, 34, 3) #la somme peut etre differente de 100
13 labels <- c("France", "Italie", "UK", "Allemagne", "Espagne")
14 pie(vec, labels=labels, main="Mon camembert")
15
16 # construction d'un graphique en barre
17 barplot(vec, names.arg=labels, main="Mes barres paralleles", ylab="Poucentages")
```

5. Package `ggplot2` : `http://ggplot2.org/`.

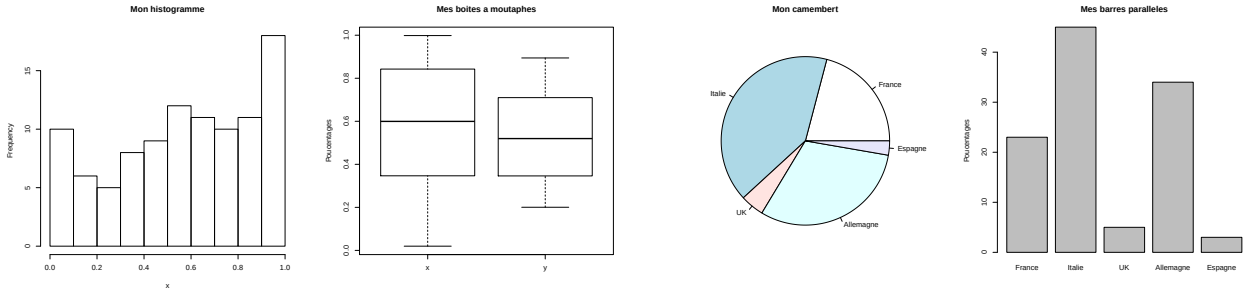


FIGURE 2 – Exemples de graphiques construits par l'exemple d'utilisation des fonctions graphiques de R.

La difficulté d'utilisation de ces fonctions réside dans l'usage correct des paramètres. L'exemple ci-dessous illustre certains d'entre eux, mais chacune des fonctions a de nombreux autres paramètres spécifiques. En situation pratique, le plus simple sera de vous reporter à l'aide de R ou bien à des exemple en ligne. On peut néanmoins faire quelques commentaires généraux sur les fonctions et leurs paramètres :

– le premier paramètre de la fonction est toujours le vecteur qui contient les données à afficher
– le paramètre **main** définie le titre principal du graphique
– les paramètres **xlab** et **ylab** permettent de définir les noms des axes

Pour la plupart des autres paramètres, il n'y a rien de très générique (et c'est dommage !). On constate en particulier que pour donner un nom aux secteurs du graphique en camember, il faut utiliser le paramètre **labels** alors que pour le diagramme en barre, il faut donner les noms au travers du paramètre **names.arg** ... mais pourquoi sont-ils si méchants ? !

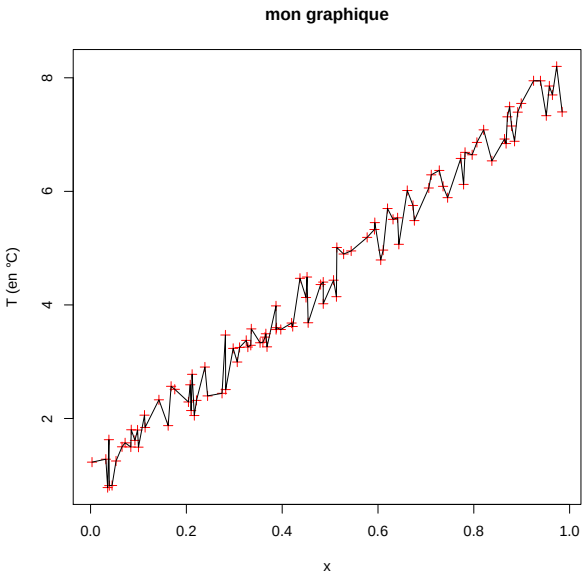
6.5.2 Affichage de graphique (x, y)

Pour afficher des données dans le plan pour lesquels on a les coordonnées dans le plan, R fournit les fonctions suivantes :

– **plot** : **créé** un nouveau graphique (x, y) (points ou courbe). À chaque nouvel appel de la fonction, un nouveau graphique est créé.
– **points** : **ajoute** des points/lignes au graphique courant. Cette fonction ne peut s'utiliser que si **plot** a été utilisé avant.

L'exemple ci-dessous illustre d'utilisation de ces fonctions :

```
1 #creation de jeux de donnees aleatoires
2 x <- runif(100)
3 x <- sort(x) #ordonne les valeurs de x
4 y <- 7*x+0.5 + runif(100)
5
6 #creation d'un graphique avec des croix rouges
7 plot(x,y, col='red', pch=3, main="mon
      graphique", xlab="x", ylab="T (en C)")
8 #ajout d'une ligne reliant les points
9 points(x,y, type='l')
```



Les fonctions **plot** ou **points** doivent être paramétrées avec deux vecteurs x et y de même taille. Dans cet exemple, on retrouve la même utilisation des paramètres **main** et **xlab**. Les autres paramètres sont les suivants :

- **type** : definit si on affiche une ligne `'l'` ou **p** si on veut juste les points (valeur par défaut)
- **col** : définit la couleur d'affichage de la courbe/les points
- **pch** : définit la forme des points

De nouveau, il existe de nombreux autres paramètres à ces fonctions que vous pourrez tester/-découvrir en utilisant l'aide de R.

6.5.3 Enregistrer un graphique dans un fichier utilisable pour vos rapports

Voici quelques possibilités pour intégrer un graphique dans un traitement de texte ou générer un fichier contenant le graphique avant intégration ou archivage.

- Sous Windows, un clic droit dans la fenêtre pour copier le graphique dans le presse papier avant de le coller dans un texte, mais cette solution n'est pas valable lorsque vous devez générer beaucoup de graphique !
- générer automatiquement une fichier image par votre programme :

```
jpeg("fichier.jpeg") #ou bmp(), png(), pdf()
# mettre maintenant vos commandes pour generer un graphique
plot(1:100, 1:2:200, type='l')
text(20,80,"abcdef")
point(c(34, 45), c(1,4), col="green")
dev.off() # fermeture indispensable du fichier
```

Si vous exécutez ce programme, vous verrez que les graphiques ne s'affichent plus comme avant. En fait, ils sont directement dessinés dans le fichier.

Exemple 39 - Visualiser ou enregistrer un graphique

Voici une solution simple pour choisir si vous voulez générer un fichier avec votre graphique ou bien le visualiser (le temps de le paufiner).

```
afficher <- TRUE

if (!afficher) {
  png("output.png")
}
#Faire ici vos dessins
plot(x,y, col='red', pch=3, main="mon graphique", xlab="x", ylab="T (en C)")
if (!afficher) {
  dev.off()
}
```

Une fois que votre programme fonctionne et que vous êtes satisfait du graphique, il vous suffira de passer la variable **afficher** *à* **FALSE** *et à réexecuter votre code pour avoir la figure dans un fichier.*

Guyet Thomas, AGROCAMPUS-OUEST, 5 mars 2014