Djamal LOUANI
Professeur,
Université de Reims-champagne-Ardennes
LSTA, Université de Paris 6

INTRODUCTION AU LOGICIEL R

Avril 2013

1- Introduction

R est un langage de programmation interactif interprété et orienté objet contenant une large collection de méthodes statistiques et des facilités graphiques importantes.

R est un clone gratuit du logiciel S-PLUS commercialisé par MathSoft et développé par STATISTICAL SCIENCES autour du langage S conçu par les laboratoires BELL.

Initié dans les années 90 par Robert Gentleman et Ross Ihaka (Département de statistique, Université d'Auckland, Nouvelle Zélande), auxquels sont venus depuis s'ajouter de nombreux chercheurs, le Logiciel **R** constitue aujourd'hui un langage de programmation intégré d'analyse statistique. Le site Internet la « R core-development Team », http://www.r-project.org, est la meilleure source d'information sur le logiciel. Il contient différentes versions du logiciel, de nombreuses bibliothèques de fonctions et une documentation.

2- Interface d'utilisation de R sous Windows

L'application Rgui.exe forme une interface utilisateur simple pour l'environnement **R**. Elle est structurée autour d'une barre de menu et de diverses fenêtres.

Les menus sont très peu développés. Ils ont comme objectif de fournir un raccourci vers certaines commandes parmi les plus utilisées, mais leur usage n'est pas exclusif, ces mêmes commandes pouvant être pour la plupart exécutées depuis la console.

Le menu **File** contient les outils nécessaires à la gestion de l'espace de travail, tels que la sélection du répertoire par défaut, le changement de fichiers sources externes, la sauvegarde et le chargement d'historiques de commandes, etc…

Le menu **Edit** contient les habituelles commandes de **copier-coller**, ainsi que la boite de dialogue autorisant la personnalisation de l'apparence de l'interface, tandis que le menu **Misc** traite de la gestion des objets en mémoire et permet d'arrêter une procédure en cours de traitement.

Le menu **Packages** automatise la gestion et le suivi des librairies de fonctions permettant leur installation et leur mise à jour de manière transparente à partir du site Internet CRAN (Comprehensive R Archive Network, http://cran.r-project.org) ou de toute autre source.

Enfin, les menus **Windows** et **Help** assument des fonctions similaires à celles qu'ils occupent dans d'autres applications Windows, à savoir la définition spatiale des fenêtres et l'accès à l'aide en ligne et aux commandes de références de **R**.

Parmi les fenêtres, on distingue la console, fenêtre où sont réalisés par défaut les entrées de commandes et sorties de résultats en mode texte. A celles-ci peuvent s'ajouter un certain nombre de fenêtres facultatives, telles que les fenêtres graphiques et les fenêtres d'informations (historique des commandes, aide, visualisation de fichier, etc …), toutes appelées par des commandes spécifiques via la console.

Pour quitter **R**, on utilise la commande en ligne

```
> q()
```

3- Sensibilité des commandes R

Sur le plan technique, R est un langage avec une syntaxe très simple. Il est sensible aux majuscules. Ainsi, A et a sont des symboles différents et référent donc à des variables différentes. L'ensemble des symboles qui peuvent être utilisés avec R dépend du système d'exploitation utilisé. Normalement, tous les symboles alphanumériques sont autorisés ainsi que le point « . ». La seule restriction se résume au fait qu'un nom ne peut commencer par un digit.

Sur une même ligne, les commandes sont séparées par un point-virgule « ; ». Les commentaires peuvent êtres insérés n'importe où en commençant par le symbole « # ».

Si une commande n'est pas complète à la fin d'une ligne, R le signale en affichant par défaut
+.

4- Exécution de commandes à partir d'un fichier

Si les commandes sont stockées dans un fichier externe, appelons le 'commandes.R', situé dans le répertoire de travail 'Work', elles peuvent être exécutées à n'importe quel moment dans une session **R** avec la commande

> source("commandes.R")

La commande **source** est aussi disponible dans le menu **File.** La fonction **sink,**

> sink ("record.lis")

détourne les sorties de la console vers un fichier externe 'record.lis'. La commande

> sink()

restore à nouveau les résultats sur la console.

5- Données permanentes et objets destructibles

Les entités que **R** crée et manipule sont des objets. Elles peuvent être des variables, des tableaux de nombres, des chaînes de caractères, des fonctions ou plus généralement des structures construites à partir de ces composants.

Dans une session **R**, les objets sont créés et stockés par noms. La commande **R**

> objects()

peut être utilisée pour lister les objets actuellement stockés par R. On peut aussi utiliser la commande

> ls()

L'ensemble des objets actuellement stockés est appelé "workspace" (espace de travail). Pour détruire des objets, on utilise la fonction **rm** comme suit :

> **rm(**foo**)**

foo étant l'objet à détruire.

Tous les objets créés dans une session **R** peuvent être stockés de façon permanente dans un fichier pour une utilisation future dans des sessions **R**. A la fin de chaque session **R**, vous

avez la possibilité de sauvegarder tous les objets disponibles dans cette session. Les objets sont stockés dans un fichier avec l'extension '**.Rdata**' dans le répertoire courant.

Au démarrage de **R** dans une session nouvelle, il recharge l'espace de travail à partir de ce fichier. Au même temps, il recharge l'historique des commandes associées.

Il est recommandé de séparer les répertoires de travail pour des analyses faites avec **R**. Il est commun que des objets de noms **x** et **y** soient créés pendant une analyse. Il est difficile de leur attribuer une signification quand plusieurs analyses sont conduites dans un même répertoire.

6- Objets

Les éléments de base du langage **R** sont des objets qui peuvent être des données (vecteurs, matrices, …), des fonctions, des graphiques …

Les objets de **R** se différencient par leur mode, qui décrit leur contenu, et leur classe, qui décrit leur structure. Les objets atomiques sont de mode homogène et les objets récursifs sont de mode hétérogènes.

Les différents modes sont :

Null (objet vide), **logica**l, **numeric**, **complex**, **character**

Les principales classes d'objets sont :

Vector, **matrix**, **array**, **factor**, **times-series**, **data-frame**, **list**

On peut avoir des vecteurs, matrices, tableaux, variables catégorielles, séries chronologiques de mode **null** (objet vide), **logical**, **numeric**, **complex**, **character** mais une liste ou un tableau de données peuvent être hétérogènes.

6.1- Les vecteurs

Il s'agit de l'objet de base dans **R**. Un vecteur est une entité unique formée d'une collection ordonnée d'éléments de même nature. Un vecteur peut être constitué d'éléments numériques, logiques ou alphanumériques.

La constitution manuelle d'un vecteur peut être faite grâce à la fonction **c()** comme suit

> **x** <- **c(**elt1,elt2, …**)**

où elt1, elt2, … sont les composants du vecteur noté ici **x**. Les nombres décimaux doivent être encodés avec un point décimal, les chaînes de caractères entourés de doubles guillemets " ", et les vecteurs logiques sont codés par les chaînes de caractères **TRUE** et **FALSE** ou leurs abréviations respectives **T** et **F**. Enfin, les données manquantes sont codées par la chaîne de caractères **NA**.

On peut aussi assigner des composantes à un vecteur en utilisant la fonction **assign()**.

> **assign(**"x",c(7,5,5.6,1,4,-5**)**

Exemple :

> **a** <- **c(**7,5,5.6,1,4,-5**)**

Cette commande crée l'objet **a** recevant un vecteur numérique de dimension 6 et de coordonnées 7,5,5.6,1,4,-5. Pour afficher l'objet **a**, on exécute la commande

> a

Pour afficher la 1er coordonnée de **a,** on exécute la commande

> a[1]

On crée un vecteur **d** de dimension 3 prenant des composantes du vecteur **a** en exécutant la commande

> **d=a[c(**1,3,5**)]**

On peut multiplier et diviser les coordonnées d'un vecteur par une constante ou diviser une constante par les différentes composantes d'un vecteur en exécutant les opérations suivantes

> 2*a
> e = a/3
> f = 5/d

On peut aussi multiplier terme à terme des vecteurs. Cela est fait par les opérations suivantes

> a*d
> e/d

Les opérateurs arithmétiques élémentaires sont +, -, *, / et ^ pour la puissance. Les opérations sont faites comme pour les réels.

La somme et le produit des éléments d'un vecteur, sa dimension et son vecteur transposé sont donnés par les fonctions suivantes

> **sum(d)**
> **prod(d)**
> **length(d)**
> **t(d)**

La fonction **sort()** permet d'ordonner dans l'ordre croissant les composantes d'un vecteur.

Le produit scalaire entre les vecteurs **d** et **e** est obtenu en exécutant la commande

> **t(d)%*%e**

La commande

> **bool=d==5**

crée un vecteur booléen **bool** de même dimension que **d** recevant **TRUE** si **d[i]=5** et **FALSE** sinon.

La commande

> **text = c(**"grand","petit"**)**

crée un vecteur caractère **text** de dimension 2 de coordonnées *grand* et *petit*.

La fonction

> **is.vector(d)**

renvoie l'expression logique **TRUE** si son argument est un vecteur et **FALSE** sinon

Le logiciel **R** dispose de beaucoup de facilités pour générer des suites de nombres. Par exemple

> a = 1:30

est équivalent à la commande

> a <- **c(**1,2,…,30**)**

La commande > b = 2*1:15 est équivalente à > b <- **c(**2,4,…,28,30**)**.
La construction d'une suite peut être dans le sens inverse ,ie, 30:1.

La fonction **seq()** est celle qui donne le plus de facilités pour générer des suites. Elle a cinq arguments et seulement certains d'entre eux peuvent être spécifiés dans un appel. Les deux premiers donnent le début et la fin de la suite. On peut écrire

s**eq(**1,30**)** ou **seq(**from=1,to=30**)**.

Les deux arguments suivants sont **by** et **length** qui spécifient l'incrémentation et la longueur de la suite. On peut écrire

> **seq(**-5,5,by=.2**)** -> S

ou

> U <- **seq(**length=51,from=-5,by=.2**)**

La fonction **rep()** permet de répliquer un objet de différentes façons. Sa forme la plus simple est

> v <- **rep(**x,times=5**)**

qui fait 5 copies de l'objet **x** et l'affecte à l'objet **v**.

Vecteurs logiques

Comme avec les vecteurs numériques, R permet aussi de manipuler des quantités logiques. Les éléments d'un vecteur logique sont TRUE, FALSE et NA (pour donnée manquante).

Les vecteurs logiques sont générés de façon conditionnelle. Par exemple

> temp <- x > 13

définit temp comme un vecteur de même longueur que x avec la valeur FALSE correspondant aux éléments de x pour lesquels la condition n'est pas satisfaite et avec la valeur TRUE dans les cas où la condition est satisfaite.

Les opérateurs logiques sont

<, <=, >, >=, ==, != (pour l'inégalité), & (pour l'intersection), | (pour l'union).

Valeurs manquantes

Quand une composante d'un vecteur n'est pas disponible, la valeur spéciale **NA** lui est assignée.

La fonction **is.na(x)** donne un vecteur logique de même taille que **x** avec la valeur **TRUE** quand l'élément en question est **NA**.

On note que l'expression logique **x==NA** est différente de **is.na(x)** puisque **NA** n'est pas réellement une valeur mais un marqueur pour une quantité qui n'est pas disponible. En fait, **x==NA** est un vecteur de même taille que **x** avec toutes ses composantes mises à **NA**.

Il existe un second type de valeurs manquantes produites elles par des calculs numériques, ce qui est noté **NaN** qui veut dire "Not a Number". Cela est produit par les opérations telles que

0/0, inf – inf

La fonction **is.na(x)** renvoie la réponse **TRUE** aussi bien quand la valeur est **NA** que **NaN**. Pour cibler les valeurs **NaN**, on utilise la fonction **is.nan(x)**.

Vecteurs de caractères

Les vecteurs de caractères sont souvent utilisés dans R, par exemple pour labelliser les

graphiques. Les suites de caractères sont mises entre doubles côtes. Un vecteur caractère peut être défini en utilisant la fonction **c()**.

La fonction **paste()** prend un nombre arbitraire d'arguments et les concatène un par un en une chaîne de caractères. Par défaut, les arguments sont séparés par un blanc, mais cela peut être modifié en utilisant le paramètre, **sep**=*string*, qui le change en string. Par exemple, on peut avoir

> labs <- **past(**c("X","Y"), 1:10, sep=""**)**

qui affecte à labs le vecteur caractère

c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10").

Sélection et modification d'ensembles de données

Un sous ensemble d'éléments d'un vecteur peut être sélectionné en accolant au nom du vecteur un vecteur index entre deux crochets. On distingue 4 façons différentes d'indexer des vecteurs.

1. *Un vecteur logique* : le vecteur index doit être de même longueur que le vecteur concerné. Les valeurs correspondant à **TRUE** dans le vecteur index sont sélectionnées et celles correspondant à **FALSE** omises.
   Par exemple **> y <- x[!is.na(x)]** crée un objet **y** contenant les valeurs non manquantes de x.
   L'instruction > **(x+1)[(!is.na(x)) & x>0]** -> **z** crée un objet **z** dans lequel elle place les valeurs du vecteur **x+1** qui sont à la fois non manquantes et positives.

2. *Un vecteur d'entiers naturels positifs* : Dans ce cas, les valeurs du vecteur index doivent être dans l'ensemble **{**1,2, …, **length(x)}**. Ainsi, **x[6]** désigne le 6ème composant de **x**. > **x[1:10]** sélectionne les 10 premiers éléments de **x**.

3. *Un vecteur d'entiers négatifs* : Un tel vecteur index spécifie les composants à exclure. Ainsi, > **y** <- **x[-(1:5)]** donne un objet **y** avec les éléments de **x** autres que les 5 premiers.

4. *Un vecteur de chaîne de caractères* : Cela est possible seulement quand un objet possède des attributs nominaux pour identifier ses composants. Dans ce cas un sous-vecteur du vecteur des attributs peut être utilisé de la même façon que le vecteur des entiers positifs de l'item 2.
   > fruit <- c(5, 10, 1, 20)
   > names(fruit) <- c("orange", "banane", "pomme", "pêche")
   > lunch <- fruit[c("pomme", "orange")]

6.2 Facteurs

Un facteur est un objet vecteur utilisé pour spécifier une classification discrète de composantes d'autres vecteurs de même taille. On a des facteurs ordonnés et non ordonnés.

Un exemple spécifique :

On suppose que l'on dispose d'un échantillon de 30 comptables chargés des taxes dans l'ensemble des états et des territoires de l'Australie et que leur état d'origine est spécifié par un

vecteur caractère donné par

```
> state <- c("tas","sa","qld","nsw","nsw","nt","wa","wa","qld","vic","nsw","vic","qld",
            "qld","sa","tas","sa","nt","wa","vic","qld","nsw","nsw","wa","sa","act",
            "nsw","vic","vic","act")
```

On note que pour un vecteur caractère, le terme "ordonné" signifie l'ordre alphabétique.

Un facteur est crée en utilisant la fonction **facteur()** comme suit :

```
> statef <- factor(state)
```

La fonction **print()** traite les facteurs de façon légèrement différente par rapport aux autres objets :

```
> statef
  [1] tas sa qld nsw nsw nt wa wa qld vic nsw vic qld qld sa
  [16] tas sa nt wa vic qld nsw nsw wa sa act nsw vic vic act
  Levels : act nsw nt qld sa tas vic wa
```

Pour obtenir les niveaux d'un facteur, on utilise la fonction **levels()** comme suit :

```
> levels(statef)
  [1] "act"  "nsw"  "nt"  "qld"  "sa"  "tas"  "vic"  "wa"
```

La fonction tapply()

Pour continuer sur le même exemple, supposons que l'on dispose des revenus de ces comptables dans un autre vecteur :

```
> incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
                61, 61, 61, 58, 51, 48, 51, 48, 65, 49, 49, 41, 48, 52, 46, 59, 46, 58, 43)
```

Pour calculer la moyenne empirique de incomes pour chaque état, on peut utiliser la fonction **tapply()** comme suit :

```
> incmeans <- tapply(incomes, statef, mean)
```

Cela donne le vecteur moyen avec des composantes correspondant aux niveaux

```
Act   nsw   nt    qld  sa    tas   vic   wa
44.5  57.33  55.5  53.6 55.0  60.5  56.0  52.25
```

La fonction **tapply()** est utilisée pour appliquer une fonction, ici mean(), à chaque groupe de composants du 1er argument, ici incomes, définis par les niveaux du second composant, ici statef.

6.2 Tableaux et Matrices

6.2.1 Tableaux :

Un tableau peut être considéré comme une collection multi-indicée de données. Un vecteur peut être utilisé dans **R** comme un tableau seulement si sa dimension est vectorielle. Par exemple, si z est un vecteur de 1500 éléments, la commande

```
> dim(z) <- c(3,5,100)
```

lui attribue une dimension qui permet de le traiter comme un tableau 3x5x100.

Un élément d'un tableau peut être référencé en donnant le nom du tableau et sa position dedans entre crochets.

Exemple : Si le vecteur dimension d'un tableau **a** est **c(3,4,2)**, alors le tableau **a** a 3X4X2 entrées qui sont prises en compte dans le vecteur de données dans l'ordre suivant

a[1,1,1], a[2,1,1], …, a[2,4,2],a[3,4,2] .

Tableau index :

Les éléments d'un tableau peuvent être référencés en donnant le nom du tableau et leurs positions entre crochets.

Comme suite à l'exemple précédant, **a[2,,]** désigne un 4x2-tableau de vecteur dimension **c(4,2)** dont le vecteur de données contient les valeurs

C(a[2,1,1],a[2,2,1],a[2,3,1],a[2,4,1],a[2,2,1],a[2,2,2],a[2,3,2],a[2,4,2]**)**.

Comme avec un vecteur index pour une position quelconque, un tableau peut être utilisé avec un tableau index pour soit assigner un vecteur de quantités à une collection irrégulière d'éléments dans le tableau ou bien extraire une collection irrégulière comme un vecteur.

Exemple : Dans le cas d'un tableau à doubles entrées, une matrice index, consistant en deux colonnes et autant de lignes que l'on désire, est donnée. Les entrées de la matrice index sont les indices des lignes et des colonnes. On suppose que l'on dispose d'une matrice X de taille 4x5 et on voudrait

- Extraire les éléments X[1,3], X[2,2] et X[3,1] avec une structure de vecteur,
- Remplacer ces entrées dans le tableau X par des 0.

Dans ce cas on a besoin d'un tableau d'indices de dimension 3x2. Les instructions à exécuter sont :

```
> X <- array(1:20, dim=c(4,5)) # génère un tableau 4x5
> X                            # imprime X
> I <- array(c(1:3,3:1), dim=c(3,2)) # I est un tableau  3x2
> I                            # imprime I
> X[I] <- 0                    #remplace les éléments par des 0.
> X                            #imprime X
```

La fonction array() :

La fonction **array()** sert à construire des tableaux. Elle se présente sous la forme

> Z <- **array(**data_vector, dim_vector**)**

Par exemple, si le vecteur H contient 24 éléments ou moins, alors la commande

> Z <- array(H, dim=c(3,4,2))

utilisera H pour installer un tableau 3x4x2 dans Z. Si la taille de H est plus petite que 24, ses valeurs seront recyclées pour compléter jusqu'à 24. L'exemple extrême est celui mettant toutes les valeurs d'un tableau Z à 0. Cela est fait par l'instruction suivante :

> Z <- array(0,c(3,4,2)).

Transposée généralisée d'un tableau :

La fonction **aperm(a,***perm***)** peut être utilisée pour permuter les éléments d'un tableau **a**. L'argument *perm* doit être une permutation des entiers {1,2,…,k}où k est le nombre d'indices de **a**. Cette opération généralise la transposition de matrices. Si A est une matrice, alors B donné par

> B <- **aperm(**A, c(2,1)**)**

est juste la transposée de A qui est équivalent à B <- t(A).

6.2.2 Matrices :

Une matrice est un tableau avec deux indices. Cependant, R dispose de divers opérateurs et fonctions applicables uniquement aux matrices comme les fonctions t(X), nrow(X) et ncol(X).

Multiplication :

L'opérateur de multiplication de matrice est **%*%**. Le produit des matrices A et B est fait comme suit :

```
> C <- A%*%B
```

Si A et B sont deux matrices carrées, alors l'instruction

```
> D <- A*B
```

donne une matrice D avec des éléments qui sont produits des éléments de A et B.

La fonction crossprod() appliquée à X et Y donne l'équivalent de

```
> t(X)%*%Y
```

La fonction diag() :

Si **v** est un vecteur, la fonction **diag(v)** donne une matrice diagonale dont la diagonale contient les éléments du vecteur **v**.

Si **M** est une matrice, **diag(M)** donne un vecteur contenant les éléments diagonaux de **M**.

Si **k** est un entier positif, **diag(k)** donne une matrice identité de dimension **kxk**.

Equations linéaires et inversion :

Pour résoudre l'équation linéaire b=A*x, on utilise la fonction solve() comme suit :

```
> x <- solve(A,b)
```

L'inverse A^{-1} de la matrice A peut être obtenu avec la commande

```
> solve(A)
```

Valeurs propres et vecteurs propres :

La fonction **eigen(S)** calcule les valeurs propres et les vecteurs propres d'une matrice symétrique **S**. Le résultat de cette fonction est une liste de deux composants nommés *Values* et *vectors*. La commande

```
> ev <- eigen(S)
```

assigne une liste à ev. Alors, **ev$val** est le vecteur des valeurs propres de S et **ev$vec** est la matrice des vecteurs propres correspondant. Pour obtenir uniquement les valeurs propres, on utilise la commande

```
> evals <- eigen(S)$values
```

evals contiendra alors les valeurs propres.

Ajustement des moindres carrés :

La fonction **lsfit()** retourne une liste donnant les résultats de la procédure d'ajustement des moindres carrés. La commande

```
> ans <- lsfit(X,y)
```

donne les résultats d'un ajustement des moindres carrés où **y** est le vecteur des observations et

X la matrice du plan d'expérience. On note que le terme moyen est automatiquement inclus sans que cela soit nécessaire de le spécifier comme une colonne de **X**.

Une autre fonction très proche de **lsfit()** est la fonction **qr()**. On considère les commandes suivantes :

```
> Xplus <- qr(X)
> b <- qr.coef(Xplus, y)
> fit <- qr.fitted(Xplus, y)
> res <- qr.resid(Xplus, y)
```

Cela calcule la projection orthogonale de **y** sur le domaine de **X** dans **fit**, la partie résiduelle est dans **res** et le vecteur des coefficients est dans **b**.

La matrice X n'est pas supposée être de rang égal au nombre de colonne. Les redondances sont recherchées et éliminées aussitôt qu'elles sont trouvées.

Les fonctions cbind() et rbind() :

Des matrices peuvent être construites en concaténant horizontalement ou verticalement des vecteurs ou d'autres matrices en utilisant respectivement **cbind()** et **rbind()**.
Dans la commande

```
> X <- cbind(arg-1,arg_2, …)
```

les arguments de **cbind()** peuvent être soit des vecteurs de longueurs quelconques ou des matrices ayant des colonnes de même taille.

Les résultats de **cbind()** et **rbind()** ont toujours un statut de matrice.

Tables de fréquences de facteurs :

Un facteur définit une partition en groupes. De façon similaire, une paire de facteurs définit une classification à double entrée et ainsi de suite. La fonction table() permet de calculer des tables de fréquences pour des facteurs de même taille. Si k arguments sont disponibles, alors le résultat est un tableau de fréquences à k entrées. La commande

```
> statefr <- table(statef)
```

donne, dans statefr, une table de fréquences de chaque état dans l'échantillon. Les fréquences sont ordonnées et labellisées par l'attribut du niveau de la catégorie.

6.3 Listes et structures de données :

6.3.1 Listes :

Une liste de **R** est un objet consistant en une collection ordonnée d'objets considérés comme ses composants. Il n'est pas nécessaire que les composants d'une liste soient de même type ou de même mode. Par exemple, une liste peut être constituée d'un vecteur numérique, d'une valeur logique, d'une matrice, d'un vecteur complexe, d'un tableau de caractère, d'une fonction, etc …

Exemple :

```
> lst <- list(name="Fred",wife="Marie", no.children=3, child.ages=c(4,7,9))
```

Les composants sont toujours numérotés et peuvent être référencés comme tels. Ainsi, la liste lst contient 4 composants, ils sont référencés individuellement par :

lst[[1]], lst[[2]], lst[[3]], lst[[4]].

En outre, comme lst[[4]] est un vecteur, sa première composante est référencée par lst[[4]][1].

Si lst est une liste, alors **length(lst)** donne le nombre de composants qu'elle contient.

Les composants d'une liste peuvent être nommés, dans ce cas un composant peut être référencé en donnant son nom comme une chaîne de caractère. On a alors

lst$name est équivalent à lst[[1]] qui donne "Fred"
lst$wife est équivalent à lst[[2]] qui donne "Marie"
lst$child.ages[1] est équivalent à lst[[4]][1] qui donne 4.

On peut aussi utiliser la syntaxe suivante

lst[["name"]] qui est équivalente à lst$name.

Construction et modification de listes :

De nouvelles liste peuvent être formées à partir d'objets existant en utilisant la fonction **list()**. La commande

> lst <- **list(**name_1=objet_1, …, name_m=objet_m**)**

construit une liste lst avec m composants dont les noms sont spécifiés. Si les noms ne sont pas mentionnés, alors les composants seront seulement numérotés. Une liste peut être étendue en spécifiant de nouveaux composants. Par exemple,

> lst[5] <-list(matrix=mat)

Concaténation de listes :

Pour concaténer les listes A, B, et C, on utilise la fonction c() comme suit :

> list.ABC <- c(list.A, list.B, list.C)

6.3.2 Structures de données :

Classes :

Un attribut spécial connu comme **classe** d'un objet est utilisé pour obtenir un style pour l'objet en programmation avec **R**. Par exemple, si un objet a la classe **"data.frame"**, il est imprimé d'une certaine façon. La fonction **plot()** l'affiche alors d'une certaine façon. D'autre fonctions dites génériques comme **summary()** réagissent à cet attribut comme si c'était un argument.

Pour éliminer temporairement les effets de la classe, on utilise la fonction **unclass()**.

Une structure de données est une liste de classe "data.frame". Il y a des restrictions sur les listes pouvant avoir l'attribut "data.frame". Ces restrictions sont :

- Les composants doivent être des vecteurs (numérique, caractère, logique), des facteurs, des matrices (numériques), des listes ou d'autres structures de données.
- Les matrices, les listes et les structures de données donnent à la nouvelle structure de données autant de variables qu'elles ont de colonnes, d'éléments ou de variables.
- Les vecteurs numériques et logiques et les facteurs sont inclus dans la structure de données comme ils sont. Un vecteur caractère est transformé en facteur dont les niveaux sont les seules valeurs apparaissant dans le vecteur.

- Les structures vectorielles apparaissant comme des variables dans une structure de données doivent avoir la même longueur et les structures matricielles doivent avoir la même taille de lignes.

Construire des structures de données :

On utilise la fonction **data.frame()**. Par exemple,

> accountants <- **data.frame(**home=statef, loot=income, shot=income**)**

Une liste dont les composants sont conformes aux restrictions imposées par une structure de données peut être transformée en structure de données en utilisant la fonction

As.data.frame()

La façon la plus simple de construire une structure de données est d'utiliser la fonction **read.table()** pour lire les données à partir d'un fichier externe.

Les fonctions attach() et detach() :

La notation $ pour faire référence aux composantes des listes n'est pas toujours convenable. Il est utile rendre une composante d'une liste ou d'une structure de données visible comme variable sous son nom de composante sans faire référence au nom de la liste ou de la structure explicitement.

La fonction **attach()**,ayant aussi bien un nom de répertoire comme argument, peut aussi avoir une structure de données. Ainsi, supposons que lentils est une structure de données avec 3 variables lentils$u, lentils$v et lentils$w. La commande

> **attach(**lentils**)**

place la structure de données en 2ème sur le chemin de recherche. Pourvu que les variables u, v, ou w ne soient pas en 1ère position, u, v et w sont alors des variables de la structure de données disponibles pour leurs propres comptes. Ainsi,

> u <- v+w

ne replace pas le composant u dans la structure de données, mais le masque avec une autre variable u dans le répertoire de travail en 1ère position du chemin de recherche. Pour effectuer un changement permanant dans la structure de donnée elle-même, la plus simple manière est d'utiliser la notation $:

lentils$u <- v+w

Cependant, la nouvelle valeur du composant n'est pas valable avant que la structure de donnée soit détachée et attachée de nouveau. Pour détacher une structure de données, on utilise la fonction

> **detach()**

Travailler avec des structures de données :

Voici quelques recommandations permettant de travailler de façon confortable avec plusieurs problèmes sur un même répertoire de travail.

- Rassembler toutes les variables réservées à un problème particulier dans une structure de données avec un nom bien identifiable
- Pendant le travail avec un problème, attacher la structure de données en 2ème position et utiliser le répertoire de travail en 1ère position pour les variables temporaires.
- Avant de quitter un problème, ajouter toutes les variables désirées pour de futures références à la structure de données en utilisant la notation $, et ensuite **detach()**.

- Finalement, détruire toutes les variables non nécessaires dans le répertoire de travail en la tenant aussi propre que possible.

7- Lecteur de données à partir de fichiers :

Pour lire les données dans un fichier, on utilise les fonctions **read.table()** et **scan()**.

La fonction read.table() :

Pour lire directement une structure de données entière, le fichier externe doit avoir une forme particulière.

- La première ligne du fichier doit avoir un nom pour chaque variable de la structure de données.
- Chaque ligne supplémentaire du fichier doit avoir comme 1er item la label de la ligne et des valeurs pour chacune des variables.

A l'exception des labels des lignes, les items numériques sont lu comme des variables numériques et les variables non numériques sont lues comme des facteurs.

Un exemple de lecture est donné par

> Hp <- **read.table(**"houses.data"**)**

Si la colonne des labels des lignes est absente, le fichier est lu comme suit :

> HP <- **read.table(**"houses.data", header=TRUE**)**

La fonction scan() :

Supposons que trois vecteurs de données sont de même longueur et sont lus en parallèle. En outre, supposons que le 1er est de mode caractère et que les deux autres sont de mode numérique. Le nom du fichier est 'input.dat'. la 1ère étape est d'utiliser la fonction scan() pour lire les trois vecteurs comme une liste, ie,

> inp <- **scan(**"input.dat", list("",0,0)**)**

Le 2ème argument établit les modes des 3 vecteurs à lire. Pour séparer les données en 3 vecteurs, on utilise les affectations suivantes :

> label <- inp[[1]]; x <- inp[[2]]; y <- inp[[3]]

On peut aussi utiliser la commande suivante :

> inp <- **scan(**"input.dat", list(id="", x=0, y=0)**)**

Accès aux ensembles de données de R :

Une cinquantaine d'ensembles de données sont disponibles dans R et d'autres ensembles se trouvent dans les packages. Pour voir la liste des ensembles de données disponibles, on utilise la fonction

> **data()**

et pour charger un des ensembles, on utilise la commande

> **data(**infert**)**

Charger des données à partir de packages de R :

Pour accéder aux données de packages, on utilise l'argument package, par exemple,

> data(package="nls")
> data(Puromycin, package="nls")

8- Lois de probabilité usuelles :

Loi	Nom	Paramètres	Valeurs par défaut
Beta	Beta	P_1, P_2	
Binomiale	binom	n, p	
Binomiale négative	nbinom	n, p	
Cauchy	cauchy	Location, scale	0, 1
Khi-deux	chisq	df	
Exponentielle	exp	1/mean	1
Fisher	f	df_1, df_2	
Gamma	gamma	Shape, 1/scale	-, 1
Géométrique	geom	Prob	
Hypergéométrique	hyper	m, n, k	
Log-Normale	lnorm	mean, sd	0, 1
Logistique	logis	Location, scale	0, 1
Normale	norm	Mean, sd	0,1
Poisson	pois	lambda	
Student	t	df	
Uniforme	unif	Min, max	0, 1
Weibull	weibull	shape	
wilcoxon	wilcox	m, n	

Pour chacune de ces distributions, on dispose de quatre commandes préfixées par une des lettres d, p, q, r et suivi du nom de la distribution :

dnomdistrib : il s'agit de la fonction de densité pour une distribution de probabilité continue et de la fonction de probabilité P(X=k) dans le cas d'une distribution discrète.

pnomdistrib : il s'agit de la fonction de répartition P(X<x).

qnomdistrib : il s'agit de la fonction des quantiles, i.e., l'inverse de la fonction de répartition.

rnomdistrib : génère des réalisations aléatoires indépendantes de la distribution nomdistrib.

Exemple :

```
> qnorm(0.975); pnorm(1.96)
> rnorm(20); rnorm(10,mean=5,sd=0.5)
> x=seq(-3,3,0.1); pdf=dnorm(x)
> plot(x,pdf,type="l")
```

9- Quelques éléments de programmation :

9.1- Expressions groupées :

R est un langage à base d'expressions dans le sens où son seul type de commande est une fonction ou expression qui retourne un résultat. Même un assignement est une expression où

le résultat est la valeur assignée.

Des commandes peuvent être regroupées entre accolades, {$expr_1$, …, $expr_m$}, et dans ce cas la valeur du groupe est le résultat de la dernière expression évaluée dans le groupe. Comme un tel groupe est aussi une expression, il peut être lui-même inclus dans un autre groupe d'expressions.

9.2- Commande de contrôle :

R dispose des commandes if, for, repeat et while. Ces commandes se programment de la façon suivante :

> **if** ($expr_1$) $expr_2$ **else** $expr_3$

> **for** (name **in** $expr_1$) $expr_2$

> **repeat** expr

> **while** (condition) expr

On peut utiliser la commande **break** pour arrêter une boucle. La commande **next** peut être utilisée pour arrêter un cycle particulier en passant au cycle suivant.

10- Construction d'une nouvelle fonction :

Il est possible de définir des fonctions personnalisées soit directement au départ de la console ou bien via un éditeur de texte externe grâce à la commande

> **fix(**nom_fonction**)**

La seconde possibilité permet la correction du code en cours d'édition tandis que la première s'effectue sans retour en arrière possible.

De manière générale, la définition d'une nouvelle fonction est fait à l'aide de l'expression suivante :

> nom-fonction=**function(**arg_1=[arg_1],arg_2=[arg_2],…**){**
>
> blocs d'instructions
>
> **}**

Les accolades indiquent le début et la fin des instructions de la fonction. Les crochets ne font pas partie de l'expression mais indiquent le caractère facultatif des arguments.

Il est possible également de créer une fonction personnalisée à partir d'une fonction existante tout en conservant l'original intact grâce à la commande :

> Nom-fonction2=**edit(**nom-fonction1**)**; **fix(**nom-fonction2**)**

Lors de l'exécution, R renvoie par défaut le résultat de la dernière expression évaluée dans la fonction. Par ailleurs, les arguments sont passés à la fonction par valeur et leur portée ainsi que celle de toute assignement classique à l'intérieur d'une fonction est locale. Lors de l'exécution, une copie des arguments est transmise à la fonction, laissant les arguments originaux intacts.

Exemple :

Considérons une fonction calculer la *t*-statistique du double échantillon en montrant toutes les étapes. La fonction est définie comme suit :

```
> twosam <- function(Y1,Y2){
        n1 <- length(Y1); n2 <- length(Y2)
        Yb1 <- mean(Y1); Yb2 <- mean(Y2)
        S1 <- var(Y1); S2 <- var(Y2)
        S <- ((n1-1)*S1 + (n2-1)*S2)/(n1 + n2 -2)
        Tst <- (Yb1 – Yb2)/sqrt(S*(1/n1 + 1/n2))
        Tst
        }
```

Cette fonction peut être appelée pour effectuer le t-test comme suit :

```
> tstat <- twosam(data$male, data$female); tstat
```

La modélisation statistique dans R

R dispose de diverses facilités permettant d'ajuster des modèles statistiques de façon très simple. Auparavant, on introduit quelques fonctions de statistique exploratoire.

> mean() calcule la moyenne
> var() calcule la variance
> sd() calcule l'écart-type
> median() calcule la médiane
> quantile() calcule le quantile
> summary() donne le résumé d'une variable
> hist(., nclass=.) trace un histogramme
> boxplot() trace le diagramme en boite
> cor(.,.) calcule le coefficient de corrélation linéaire entre deux variables
> piechart() trace le diagramme en secteurs

Définition des modèles :

On considère le modèle de régression linéaire avec des erreurs indépendantes et homoscédastiques donné par

$y_i = \sum_{0 \le j \le p} \beta_j x_{ij} + e_i \quad i=1, \ldots, n \qquad ei \sim N(0,\sigma^2)$

En écriture matricielle cela donne :

$Y = X\beta + e$

Où Y est le vecteur réponse, X la matrice du modèle et β le vecteur paramètre.

Exemples :

On suppose que y, x, x_0, x_1, … sont des variables numériques, X est une matrice et A, B, C, … sont des facteurs. Les formules suivantes spécifient des modèles statistiques.

Pour une régression linéaire simple, on écrit :

y~x ou y~1+x

Pour une régression simple sans terme constant, on écrit

y~0+x

La régression multiple de la variable transformée log(y) sur x1 et x2 s'écrit

$Log(y) \sim x_1 + x_2$

La régression polynomiale du second degré de y sur x s'écrit

y~poly(x,2) ou y~1 + x + I(x^2)

La régression multiple de y avec la matrice du modèle formée par X et les termes x élevés au carré s'écrit

y~X + poly(x,2)

Pour une classification où les classes sont déterminées par le facteur A, on écrit

y~A

La forme générale d'un modèle linéaire ordinaire est donnée par

Reponse ~ op_1 $term_1$ op_2 $term_2$ op_3 $term_3$ …

Où,

- **Reponse** est un vecteur ou une matrice définissant la variable réponse

- **op$_i$** est un opérateur, + ou -, impliquant l'inclusion ou l'exclusion du terme dans le modèle
- **termi** est soit
    - un vecteur ou une matrice, ou 1
    - un facteur
    - une expression résultant de facteurs, vecteurs ou matrices liés par une formule.

La fonction de base pour ajuster un modèle linéaire est **lm()**, et l'appel se fait comme suit

> fitted.model <- **lm(**formule, **data**=data.frame**)**

Par exemple,

> fm2 <- **lm(**y~ x1 + x2, data=production**)**

permet d'ajuster une régression multiple de y sur x_1 et x_2.

Fonctions génériques pour extraire l'information du modèle :

Des informations du modèle ajusté peuvent être obtenues en utilisant les fonctions génériques suivantes

add1 coef effects kappa predict residuals
alias deviance family labels print step
anova drop1 formula plot proj summary

On donne un bref descriptif des fonctions les plus utilisées :

- **anova(object_1,object_2)** : compare un sous-modèle avec le modèle complet et produit une table d'analyse de variance.
- **coefficients(object)** : extrait les coefficients de la régression (coef(object)).
- **deviance(object)** : donne la somme des carrés des résidus avec les poids appropriés.
- **formula(object)** : extrait la formule du modèle.
- **plot(object)** : produis quatre graphiques montrant les residus, les valeurs ajustées et des diagnostiques.
- **predict(object, newdata=data.frame)** : la structure de donnée doit avoir des variables spécifiées avec les mêmes labels que l'original. La valeur est un vecteur ou une matrice de valeurs prédites correspondant aux valeurs de la variable déterminante dans la structure de données.
- **print(object)** : imprime une version concise de l'objet.
- **residuals(object)** : extrait les résidus avec les poids appropriés (resid(object)).
- **step(object)** : sélectionne un modèle convenable en ajoutant ou en retranchant des termes et en préservant les hiérarchies. Le modèle ayant la plus grande valeur pour le critère AIC est retourné.
- **summary(object)** : imprime un résumé des résultats de la régression.

Mise à jour de modèles ajustés :

La fonction **update()** permet de mettre à jour le modèle ajusté en ajoutant ou en retranchant quelque termes à sa forme initiale. Cela se fait comme suit :

> new.model <- **update(**old.model, new.formula**)**

Dans **new.formula**, on utilise **'.'** pour désigner la part de **old.model** dans la formule. Par exemple,

```
> fm5 <- lm(y ~ x1 + x2 + x3 + x4 + x5, data=production)
> fm6 <- update(fm5, . ~ . + x6)
```

Modèle linéaire généralisé :

Un modèle linéaire généralisé est modèle permettant de généraliser le modèle linéaire au cas où la distribution de la réponse n'est pas gaussienne et que la linéarité est obtenue via une transformation. Le modèle linéaire généralisé est décrit avec les propriétés suivantes :

- L'influence exercée par les variables explicatives $x_1, \ldots, x_p$ sur la réponse y du modèle se fait via une fonction linéaire qui s'écrit sous la forme
  $$\eta = \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p$$
- La densité de la variable y est de la forme
  $$f_Y(y,\mu,\varphi) = \exp[\{y\lambda - b(\lambda)\}w/\varphi + c(y,\varphi,w)]$$
  où φ est un paramètre d'échelle constant pour toutes les observations, λ est le paramètre du modèle, b() et c() sont des fonctions spécifiques correspondant au type de famille exponentielle et w est une pondération.
- Le paramètre λ est une fonction de la moyenne μ, i.e., $\lambda=\lambda(\mu)$, qui est déterminé uniquement par la famille exponentielle spécifique au travers de la relation $\mu= b'(\lambda)$.

Les familles :

Chaque distribution de la réponse admet une variété de fonctions de lien qui relie la moyenne μ avec le prédicteur linéaire. Les fonctions de lien disponibles sont

Famille exponentielle	**Fonction de lien**
Binomiale	Logit, probit, cloglog
Gaussienne	identité
Gamma	Identité, inverse, log
Gaussienne inverse	$1/\mu^2$
Poisson	Identité, log, racine carrée
Quasi	Logit, probit, cloglog, identité, inverse, log, $1/\mu^2$, racine carrée

La fonction glm() :

La fonction de R qui ajuste un modèle linéaire généralisé est la fonction **glm()** qui s'emploie comme suit

> fitted.model <- **glm(**formule, family=famille, data=data.frame**)**

La nouveauté par rapport à la fonction **lm()** est l'argument **family** qui est l'instrument par lequel la famille est décrite.

Exemples :

1- Famille gaussienne : L'appel

> fm <- glm(y ~ x1 + x2, family=gaussian, data = sales)

donne les même résultats que l'appel

> fm <- lm(y ~ x1 + x2, data = sales)

2- Famille binomiale : Si y est la variable à expliquer et x la variable explicative,alors les modèles logistique et probit sont de la forme

$$y \sim B(n, F(\beta_0 + \beta_1 x))$$

où n est le nombre d'observations, F est la fonction de répartition de la loi normale N(0,1) dans le cas du modèle probit et

$F(z) = e^z/(1 + e^z)$

Dans le cas du modèle logistique. L'appel se fait comme suit

```
> fmp <- glm(y ~ x, family = binomial(link=probit), data = data.frame)
```

dans le cas du modèle probit et

```
> fml <- glm(y ~ x, family = binomial, data = data.frame)
```

dans le cas du modèle logistique. Il n'est pas nécessaire de préciser la fonction link.

3- Famille de Poisson : Par défaut, la fonction de lien pour la famille de Poisson est la fonction log. En pratique, cette famille est utilisée pour ajuster le modèle log-linéaire de Poisson à des données de fréquences, qui suivent le plus souvent une loi multinomiale. L'ajustement du modèle se fait comme suit

```
> fmod <- glm(y ~ A + x, family = poisson(link=sqrt), data = data.frame)
```

4- Modèles de quasi-vraisemblance :

Pour toutes les familles, la variance de la variable réponse dépend de la moyenne. La forme de cette dépendance est une caractéristique de la distribution de la réponse; par exemple, pour la loi de Poisson, $Var(y) = \mu$.

Pour l'estimation dans le cas de la quasi-vraisemblance, la loi précise de la réponse n'est pas spécifiée, mais seulement la fonction de lien et la façon dont la variance dépend de la moyenne sont connues. Comme l'estimation dans le cas de la quasi-vraisemblence utilise les mêmes techniques que le cas gaussien, cette famille permet d'ajuster des modèles gaussiens avec des fonctions de lien non standard. Par exemple, considérons l'ajustement de la régression non linéaire

$$y = \theta_1 z_1/(z_2 - \theta_2) + e$$

qui peut être écrite sous la forme

$$y = 1/(\beta_1 x_1 + \beta_2 x_2) + e$$

où $x_1 = z_2/z_1$, $x_2 = -1/x_1$, $\beta_1 = 1/\theta_1$ et $\beta_2 = \theta_2/\theta_1$. Cette régression non linéaire s'ajuste comme suit

```
> nlfit <- glm(y ~ x1 + x2 - 1, family = quasi(link=inverse, variance=constant),
                                                    data = sales)
```

Ajustement et optimisation :

Dan de nombreux cas l'ajustement de courbes non linéaires est approché comme un problème d'optimisation non linéaire. Dans **R**, la routine d'optimisation non linéaire est **nlm()**. Nous traitant ci-dessous deux cas faisant appel à cette fonction.

Modèle des moindres carrés :

Une façon d'ajuster un modèle non linéaire est en minimisant la somme des carrés des erreurs. En exemple, on considère les données suivantes :

```
> x <- c(0.02, 0.02, 0.06, 0.06, 0.11, 0.11, 0.22, 0.22, 0.56, 0.56, 1.10, 1.10)
> y <- c(76, 47, 97, 107, 123, 139, 159, 152, 191, 201, 207, 200)
```

On ajuste le modèle

```
> fn <- function(p) sum((y – (p[1]*x)/(p[2] + x))^2)
```

Pour faire l'ajustement, il faut initialiser les paramètres. Pour obtenir les valeurs initiales adéquates, on utilise la méthode graphique qui consiste à deviner les valeurs des paramètres initiaux en représentant graphiquement les données et en superposant la courbe du modèle. Ici, les valeurs initiales 200 pour p[1] et 0.1 pour p[2] semblent adéquates.

```
> plot(x,y)
> xfit <- seq(0.02, 1.1,0.05)
> yfit <- 200*xfit/(0.1 + xfit)
> lines(spline(xfit, yfit))
> out <- nlm(fn, p = c(200, 0.1), hessian = TRUE)
```

Après l'ajustement, la somme des carrés des erreurs est donnée dans **out$minimun** et les paramètres estimés sont dans **out$estimates**. Pour obtenir les approximations des écart-type des estimateurs, on exécute la commande

```
> sqrt(diag(2*out$minimum/(length(y)-2) * solve(out$hessian)))
```

Maximum de vraisemblance :

La méthode du maximum de vraisemblance s'applique pour ajuster des modèles non linéaires même si les erreurs ne sont pas normalement distribuées. On détermine les valeurs des paramètres qui maximisent la log-vraisemblance. Dans cet exemple, on ajuste un modèle logistique à des données.

```
> x <- c(1.6907, 1.7242, 1.7552, 1.7842, 1.8113, 1.8369, 1.8610, 1.8839)
> y <- c(6; 13, 18, 28, 52, 53, 61, 60)
> n <- c(59, 60, 62, 56, 63, 59, 62, 60)
```

La log-vraisemblance à minimiser est :

```
> fn <- function(p) sum(-(y*(p[1] + p[2]*x) – n*log(1 + exp(p[1] + p[2]*x))
                        + log(choose(n,y))))
```

On choisit l'amorce et on fait l'ajustement

```
> out <- nlm(fn, p = c(-50, 20), hessian = TRUE)
```

Quelques modèles non standard :

Des packages spécifiques de **R** permettent de traiter quelques modèles non standard.

- **Modèles mixtes** : Le package **nlme** contient les fonctions **lme()** et **nlme()** qui permettent d'ajuster des modèles mixtes linéaires et non linéaires.
- **Régression non paramétrique** : la fonction **loess()** permet d'ajuster une régression non paramétrique. Elle se trouve dans le package **modreg**.
- **Arbre de décision** : La fonction **tree()** permet la modélisation à base d'arbre de décision. Les outils de cette modélisation se trouvent dans les packages **rpart** et **tree**.

Les procédures graphiques

1- Introduction :

R dispose d'une grande variété de facilités graphiques. Il est possible d'utiliser ces facilités pour afficher une large panoplie de graphiques statistiques et aussi d'en construire de nouveaux types de graphes. Les outils graphiques peuvent être utilisés en mode interactif comme en mode batch. Les commandes graphiques sont divisées en trois groupes de base.

- Haut niveau : les fonctions graphique créent un nouveau graphique avec éventuellement des axes, des labels, des titres etc…
- Bas niveau : les fonctions graphiques ajoutent des informations à un graphique existant.
- Interactif : les fonctions graphiques permettent d'ajouter ou d'extraire, de façon interactive, de l'information à un graphique existant en utilisant comme moyen la souris.

En outre, **R** dispose d'une liste de paramètres graphique permettant de construire des graphiques sur mesure.

2- Commandes de graphiques de haut niveau :

Des fonctions graphiques de haut niveau sont conçues pour générer un graphique complet pour les données passées en argument de ces fonctions. Une commande graphique de haut niveau crée toujours un nouveau graphique, détruisant si nécessaire le graphique courant. Des axes, des labels et des titres sont automatiquement générés à moins que le contraire soit spécifié.

2.1- La fonction plot() :

C'est la fonction graphique la plus utilisée dans **R**. Le type de graphique produit dépend du type ou de la classe du 1er argument.

> **plot(x,y)** ou > **plot(xy)**

Si **x** et **y** sont des vecteurs, cette fonction produit une représentation graphique de **y** en fonction de **x**. La 2ème forme de la commande est utilisée quand on a une liste contenant deux éléments **x** et **y** ou une matrice avec deux colonnes.

> **plot(x)**

Si **x** est une série temporelle, cela produit un graphique de la série en fonction du temps. Si **x** est un vecteur numérique, cela produit le graphique des coordonnées de x en fonction de leurs places dans le vecteur.

> **plot(f)** ou **plot(f,y)**

Si **f** est un facteur et **y** un vecteur numérique, la 1ere forme produit un diagramme en barres alors que la seconde forme produit un diagramme en boite de **y** pour chaque niveau de **f**.

> **plot(df)** ou **plot(~ expr)** ou **plot(y ~ expr)**

df est une structure de données, **y** est un objet, **expr** est une liste de noms d'objets séparés par '+'. Les deux premières formes produisent les graphiques des distributions des variables

contenues dans la structure de données ou les objets nommés. La 3ème forme construit de graphique de **y** en fonction de l'objet nommé dans **expr**.

2.2- Graphiques de données multivariées :

Si **X** est une matrice ou une structure de données, la commande

> **pairs(X)**

produit des graphiques pour les colonnes de **X** prises deux-à-deux.

Si **a** et **b** sont des vecteurs numériques et **c** un vecteur numérique ou un facteur, la commande

> **coplot(a ~ b | c)**

produit des graphiques de **a** en fonction de **b** pour chacune des valeurs de **c**. Quand c est numérique, il est divisé en intervalles conditionnant et pour chaque intervalle, un graphique de **a** en fonction de **b** est produit pour les valeurs de **c** dans l'intervalle. Le nombre et la position des intervalle peuvent être contrôlés avec **given.values=argument**. La fonction **co.intervals()** est utile pour sélectionner des intervalles. On peut aussi utiliser la commande

> **coplot(a ~ b| c + d)**

pour produire des graphiques de **a** en fonction de **b** pour chaque domaine joint conditionnant de **c** et **d**.

2.3- Divers graphiques :

> **qqnorm(x)** ou **qqline(x)** ou **qqplot(x,y)**

produisent des graphiques de comparaison de distributions. La première forme compare la distribution de **x** à une loi normale. La seconde forme ajoute une ligne droite à un tel graphique passant par les quartiles des données.

> **hist(x)** ou **hist(x,nclas=n)** ou **hist(x, breaks=b, …)**

produisent un histogramme pour un vecteur numérique. On peut préciser le nombre de classes avec l'option **nclass = argument**, on peut aussi donner les extrémités des classes avec l'option **breaks = arguments**. L'option **probability = TRUE** précise que l'histogramme représente les fréquences relatives au lieu des fréquences absolues.

> **image(x,y,z,…)** ou **contour(x,y,z,…)** ou **persp(x,y,z,…)**

Graphiques de trois variables. La fonction **image** dessine une grille de rectangles en utilisant diverses couleurs pour représenter la valeur de **z**, la fonction **contour** dessine des lignes de contour pour représenter la valeur de **z** et la fonction **persp** dessine des surfaces dans l'espace.

2.4- Arguments des fonctions graphiques de haut niveau :

add=TRUE : force la fonction à agir comme une fonction graphique de bas niveau.
axes=FALSE : supprime les axes par défaut (utile si on utilise ses propres axes).
log="x" : opère un changement logarithmique de l'échelle de l'axe x.
xlab=string : labellise l'axe x (idem pour ylab=string)
main=string : donne un titre à la figure placé en haut du graphe.
Sub=string : donne un sous titre qui est placé en dessous de l'axe x.

Les arguments de contrôle du type de graphique produit sont indiqués dans l'option **type**.

type="p" : graphe en points
type="l" : graphe en lignes
type="b" : points reliés par des lignes

type="o" : points recouverts par des lignes
type="h" : segments verticaux des points à l'axe des x
type="s" : fonction en escalier
type="n" : aucun graphique mais les axes sont dessinés.

3- Les commandes graphiques de bas niveau :

Parfois, les fonctions graphiques de haut niveau ne produisent pas exactement le type de graphes désiré. Dans ce cas, les commandes graphiques de bas niveau peuvent être utilisées pour ajouter de l'information complémentaire au graphe courant.

> **points(x,y)**, **lines(x,y)**

ajoutent des lignes ou des lignes au graphe courant.

> **text(x,y,labels,…)**

ajoute du text au graphique aux points donnés par **x**, **y**. **Labels** est un vecteur numérique ou caractère pour lequel **labels[i]** est tracé au point **(x[i],y[i])**.

> **abline(a,b) abline(h=y) abline(v=x) abline(lm.obj)**

ajoute une ligne de pente **b** et d'ordonnée à l'origine **a** au graphe courant. L'option **h=y** peut être utilisé pour spécifier les ordonnées des lignes horizontales coupant le graphe, et **v=x** donne l'indication similaire pour les lignes verticales. Aussi, **lm.obj** peut être une liste avec un vecteur coefficient de longueur 2 considéré comme la pente et l'ordonnée à l'origine comme dans un modèle d'ajustement.

> **polygon(x,y,…)**

trace un polygone défini par ses cotés **(x,y)**.

> **legend(x,y,legend,…)**

ajoute une légende au graphe courant à la position spécifiée. Les caractères, les styles de lignes, les couleurs etc … sont identifiés avec des labels dans le vecteur caractère **legend**. Au moins un autre argument **v** (un vecteur de même longueur que **legend**) avec les valeurs correspondantes pour les unités de mesure du graphe doit aussi être donné, et cela comme suit

> legend(, fill=v) couleurs pour les boites pleines
> legend(, col=v) couleurs avec lesquelles les points et les lignes sont dessinés
> legend(, lty=v) styles de lignes
> legend(, lwd=v) épaisseurs de lignes
> legend(, pch=v) caractères pour le graphe

> **title(main, sub)**

ajoute un titre **main** en haut du graphe courant et un sous titre **sub** en bas du graphe.