Université de Provence

Licence MASS Semestre 5

Initiation au logiciel R

Matthieu Kowalski - poly écrit par Clément Marteau

Année 2008-2009

Table des matières

Chapitre 1

Prise en main du logiciel

1.1 Présentation et commandes élémentaires

R est un logiciel statistique apparu en 1996. Il propose à la fois un langage et un environnement permettant un traitement et une exploitation avancée de données statistiques (tests, régression, classification et apprentissage, analyse de variance, etc.).

Contrairement à d'autres logiciels couramment utilisés dans le monde de l'entreprise (SAS, ...), **R** est un logiciel libre et gratuit. Ce dernier peut-être téléchargé à l'adresse :

http ://cran.cict.fr/

et installé sur la plupart des systèmes d'exploitation présents sur le marché : Linux, Windows, Mac-Os, etc. Le site contient également une quantité non négligeable d'informations : documentation, forums, FAQ et aide en ligne.

L'interface **R** se présente sous la forme de consoles : commandes, programmes et graphiques. Les instructions sont tapées directement dans la console commande. Nous verrons dans les sections suivantes comment utiliser les graphiques et la programmation.

Les opérateurs arithmétiques usuels s'utilisent de manière très intuitive :

```
> 2+2
[1] 4
> 3*10
[1] 30
> 25/5
[1] 5
```

Pour assigner une valeur à un objet (nous aborderons cette notion plus en détail dans la section suivante), il suffit d'utiliser la commande $<-$ ou $->$:

```
> (10-3)*7 -> a
> a
[1] 49
> babar <- 12/3
> babar
[1] 4
```

R fait la distinction entre les majuscules et les minuscules. Un nom d'objet doit toujours commencer par une lettre mais peut comporter des chiffres ou encore des points.

La fonction *ls* permet d'afficher l'ensemble des objets en mémoire :

```
> ls()
[1] "a"   "babar"
```

Pour plus de détails, il est également possible d'utiliser ls.str :

```
> ls.str()
a :  num 49
babar :  num 4
```

Il est possible d'effacer des objets de la mémoire à l'aide de la commande rm : rm(a) pour effacer l'objet a ou rm(list=ls()) pour l'ensemble des objets contenus dans la mémoire.

Enfin, comme pour tout logiciel, la commande la plus utile est celle permettant d'afficher l'aide en ligne :

```
> help("ls")
ls                    package:base                    R Documentation

List Objects

Description:

     'ls' and 'objects' return a vector of character strings giving the
     names of the objects...
```

La commande *help* renvoie donc un certain nombre d'informations sur la fonction entre guillemets à savoir : une brève description de cette dernière, l'usage que l'on peut en faire, les arguments, le type d'objet retourné ainsi que quelques exemples. Bien souvent, cette rubrique est la plus intéressante.

1.2 Les objets sous R

1.2.1 Création d'objets

Le logiciel **R** fait la distinction entre différents types de données que l'on désignera sous le terme générique d'objets. Ces derniers sont caractérisés par leur nom, leur mode (numérique, logique, caractère ou nombre complexe) et leur longueur. Les objets les plus couramment utilisés sous **R** sont présentés ci-dessous :

Vecteur. Un vecteur est une variable dans le sens courant du terme. Pour créer un vecteur, on utilise la commande vector qui a deux arguments : le mode et la longueur de l'objet crée.

```
> a <- vector(mode="numeric", length=2)
> ls.str()
a :  num [1:2] 0 0
```

Cette commande crée un vecteur numérique de longueur 2. Par défaut, les coordonnées du vecteur valent 0.

Matrices. Une matrice est en fait un vecteur plus la donnée du nombre de lignes et de colonnes.

```
> x <- 1:20
> A <- matrix(x,4,5)
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Le vecteur x est créé de manière implicite. Il contient tous les entiers compris entre 1 et 15. La matrice A est construite en remplissant les lignes et les colonnes avec les valeurs de x. Pour remplir les lignes en premier, on affectera la valeur TRUE à l'option byrow :

```
> B <- matrix(x,4,5,byrow=TRUE)
> B
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
```

Data frames. Un data frame est un ensemble de vecteurs ayant tous la même longueur mais pouvant être de mode différent. Ce type d'objet peut par exemple contenir des données ainsi que leur dénominations respectives et sera donc très utile par la suite.

```
> voitures <- 1:4
> camions <- 3:6
> data.frame(voitures,camions)
  voitures camions
1        1       3
2        2       4
3        3       5
4        4       6
```

Pour donner un nom aux lignes, on utilisera l'option row.names qui doit être un vecteur de mode caractère et de longueur égale au nombre de lignes du data frame.

Série temporelle. Les séries temporelles permettent de collecter des données qui évoluent avec le temps (population d'un pays, chiffre d'affaires d'une entreprise, etc.). Un série temporelle peut être crée à l'aide de la commande *ts()*. Nous reviendrons de manière plus détaillée sur cette dernière d'ici la fin du cours.

Il existe d'autres types d'objets reconnus par **R** mais ces derniers ne seront pas abordés dans ce cours.

1.2.2 Manipulation d'objets et fonctions élémentaires

Il peut parfois être utile d'accéder de manière sélective aux différents éléments d'un objet. Ce type de requête est très intuitive sous **R**. Par exemple, pour accéder à la troisième coordonnée d'un vecteur v, on utilisera v[3]. De même pour une matrice ou un data.frame A, A[i,j] renvoie l'élément situé à la i-ème ligne et la j-ème colonne. Reprenons la matrice B définie précédemment :

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

La quasi totalité des fonctions usuelles est disponible sous **R** ($\log, \exp, \cos, \sin, \tan, \ldots$). Nous aurons également besoin de fonctions permettant l'exploitation de données vectorielles :

- sum(x) : somme des coefficients du vecteurs x,
- prod(x) : produit des éléments de x,
- min(x) : minimum des coefficients de x (idem pour max(x)).

Les quantités statistiques usuelles (variance, médiane, covariance) sont obtenues par les commandes respectives var, median et cov.

Certaines commandes et fonctions sont dédiées au calcul matriciel. Le logiciel **R** offre à ce titre de nombreuses possibilités. Pour la multiplication de deux matrices, on utilisera l'opérateur %*%.

```
> A <-matrix(2,2,2) ; B <- matrix(3,2,2)
> A;B
     [,1] [,2]
[1,]    2    2
[2,]    2    2
     [,1] [,2]
[1,]    3    3
[2,]    3    3
> A %*% B
     [,1] [,2]
[1,]   12   12
[2,]   12   12
```

On notera également l'existence de la fonction diag permettant la création et la manipulation de matrice diagonales (cf l'aide en ligne pour plus de détails).

1.3 Générer et exploiter des données

1.3.1 Générer des données

Il peut parfois être utile de pouvoir directement générer des données de manière "régulière" ou encore aléatoire. Nous illustrerons en TP quelques théorèmes probabilistes fondamentaux de convergence (Théorème de la limite centrale ou la loi des grands nombres).

Nous avons vu précédemment qu'il était possible de générer un vecteur contenant tous les entiers compris entre deux nombres a et b à l'aide de la commande a :b :

```
> 5:18 -> x
> x
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18
```

Pour générer des séquences de nombres réels dans le même esprit, on utilisera la commande seq :

```
> seq(2,5,0.25)
 [1] 2.00 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00
```

Le vecteur créé contient tous les nombres réels compris entre 2 et 5 et ayant pour incrément 0.25. On pourra également construire des séquences en rentrant manuellement les valeurs de ces dernières :

```
> c(2,7,7.4,7.5,7.9,8,10)
[1]  2.0  7.0  7.4  7.5  7.9  8.0 10.0
```

Enfin, la commande rep définit un vecteur dont tous les coefficients sont identiques :

```
> rep(2,12)
 [1] 2 2 2 2 2 2 2 2 2 2 2 2
```

Étant donné un vecteur, une matrice, ou encore un data.frame appelé par exemple X, il est possible de modifier chacune des valeurs de cet objet à l'aide de l'instruction data.entry(X). Cette dernière ouvre une console de type *tableur*. Il est alors très facile d'accéder au différentes composantes de l'objet X.

L'ensemble des commandes présentées précédemment permet de créer des séquences déterministes. Il peut être parfois utile de travailler avec des données aléatoires (illustration de Théorème

probabilistes, approximation par méthode Monte-Carlo,etc.). Le logiciel **R** permet de générer une large gamme de variables aléatoires. Ci-dessous, un récapitulatif des différentes densités de probabilité disponibles :

- normale : rnorm(n,mean=0,sd=1)
- exponentielle : rexp(n,rate=1)
- Poisson : rpois(n, lambda=1)
- Student : rs(n,df=2)
- uniforme : runif(n,min=0,max=1)
- binomiale : rbinom(n,size=5,proba=0.5)

Pour chaque commande, n renseigne la taille du vecteur souhaité. Les paramètres spécifiques à chaque densité de probabilité doivent ensuite être spécifiés. Par exemple, si l'on souhaite créer un vecteur de taille 10 dont les coefficients suivent une loi uniforme sur l'intervalle $[2, 3]$, on utilisera :

```
> runif(10,min=2, max=3)
 [1] 2.883957 2.450405 2.362389 2.889726 2.844320 2.838443 2.490955 2.123026
 [9] 2.264480 2.069791
```

Toutes les fonctions présentées ci-dessus peuvent être utilisées pour obtenir la densité de probabilité en un point ou le quantile en remplaçant respectivement le r par d ou q.

Il sera parfois utile de convertir un vecteur, une matrice ou n'importe quel type de donnée sous la forme d'une série temporelle. Cette opération sera réalisé à l'aide de la commande ts (pour *time serie*). Cette fonction admet plusieures options :

- data : un vecteur ou une matrice
- start : date de la première observation
- end : date de la dernière observation
- frequency : fréquence des observations par unité de temps
- names : précise le nom des différentes séries dans le cas d'une série multiple

Exemple de création de série temporelle :

```
> X <- c(1,4,7,9,5,34,25,1,4,0,9)
> ts(X,frequency=12, start=c(2007,9))
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2007                                   1   4   7   9
2008   5  34  25   1   4   0   9
```

Remarque : dans l'exemple précédent, les observations commencent au mois de septembre. On a donné la valeur c(2007,9) à l'option start : la série commence donc au neuvième mois de l'année 2007.

1.3.2 Lecture de données

Le principal intérêt statistique de **R** est la possibilité de créer ou d'exploiter des données de grande taille. Ces dernières pourront être récupérées sous la forme de fichier texte (sources : INSEE, bureau d'études, etc.) et exploitées (tests, régression, graphiques, etc.). Ces aspects seront traités en détail dans la suite de ce cours.

Pour pouvoir récupérer des données, il est utile de connaître le répertoire de travail, c'est-à-dire le répertoire sous lequel les divers résultats seront sauvegardés par défaut. Ce dernier s'obtient à l'aide la commande getwd() :

```
> getwd()
[1] "C:/Program Files/R/R-2.5.1" (Windows)
[1] "/home/Enseignements/R" (Environnement Sun)
```

Pour changer le répertoire de travail, on utilisera la commande *setwd* :

```
> setwd("C:/Documents and Settings/Mes documents/Cours R")
> getwd()
[1] "C:/Documents and Settings/Mes documents/Cours R"
```

Remarque : R ne reconnaît que le caractère "/" pour spécifier le chemin d'accès d'un répertoire (même sous Windows).

Nous nous intéresserons uniquement à la lecture de fichiers textes, bien que **R** puisse lire des fichiers sous des formats aussi différents que SAS, excel, etc., ou encore accéder à des bases de données de type SQL.

La fonction read.table convertit un tableau de données dans un fichier texte en data.frame. Pour un fichier nommé "Tableau.dat", on utilisera la commande :

```
> Tab1 <- read.table("Tableau.dat")
```

qui crée un data.frame "Tab1". Si le fichier n'est pas placé dans le répertoire courant, on pourra spécifier le chemin d'accès de ce dernier directement dans la commande read.table. Cette dernière admet un certaines d'options :

- header : indique si la première ligne contient les noms de variables. Par défaut, la valeur de cette option est FALSE.
- sep : précise le séparateur de champ dans le fichier entre guillemets (" " par défaut).
- dec : le caractère utilisé pour les décimales ("." par défaut).
- row.names : indique par l'intermédiaire d'un vecteur de mode caractère le nom des lignes (par défaut : $1, 2, 3, \ldots$).
- col.names : idem pour les colonnes.
- na.strings : précise la valeur des données manquantes. Par défaut, la valeur de cette option est "NA".
- nrows : nombre maximum de lignes à lire.
- skip : nombre de lignes à sauter avant de commencer à lire des données. Cette option est utile quand le fichier texte contient par exemple un préambule.
- blank.lines.skip : si "TRUE", ignore les lignes blanches.
- comment.char : précise la caractère utilisé pour faire des commentaires. Toutes les lignes commençant par ce caractère ne seront pas prises en compte.

Cette liste d'options n'est pas exhaustive (cf l'aide en ligne) mais permet déjà d'analyser un nombre conséquent de fichiers textes. Il existe également un certain nombre de variantes de la commande read.table dont la valeur des options par défaut est quelque peu différente : read.csv, read.csv2, read.delim, read.delim2.

Imaginons que l'on souhaite récupérer les données d'un fichier intitulé donnees.dat dont le contenu est indiqué ci-dessous :

```
A B C D E
12 50 65 32 2
8 45 13 5 1
7 36 85 21 26
```

Si ce dernier est placé dans le répertoire courant, on utilisera :

```
> exemple <- read.table("donnees.dat" , header=TRUE)
> exemple
   A  B  C  D  E
1 12 50 65 32  2
2  8 45 13  5  1
3  7 36 85 21 26
```

Par défaut, le nom des variables est $V1, V2, V3, \ldots$.

On peut également mentionner l'existence de la commande read.fwf permettant de lire des données dans un format à largeur fixé. Nous reviendrons plus en détail sur cette dernière dans une planche de TP.

1.3.3 Enregistrer des données dans un fichier

Nous avons vu dans les sections précédentes comment créer et lire des données. Après d'éventuelles modifications, il sera parfois utile de pouvoir sauvegarder ces dernières dans un fichier texte. C'est le chemin inverse de l'étape de lecture. Cette procédure peut être réalisée à l'aide de la commande write.table. Les différentes options disponibles sont détaillées ci-dessous :

- file : nom du fichier dans lequel écrire (vérifier le répertoire courant)
- append : prend une valeur logique. Si TRUE, **R** ajoute les données dans le fichier concerné sans effacer les précédentes. La valeur par défaut est FALSE.
- sep : précise le séparateur à utiliser.
- eol : caractère de fin de ligne. Par défaut prend la valeur "n" (retour chariot)
- na : caractère à utiliser pour les données manquantes.
- dec : précise le caractère à utiliser pour les décimales.

De nombreuses autres options sont disponibles pour cette fonction (cf aide en ligne pour plus de détails).

1.4 Les graphiques

1.4.1 Fonctions graphiques

On distingue sous **R** deux types de commandes graphiques : les fonctions de premier et de second ordre. Les commandes de premier ordre permettent de créer des graphiques à partir d'objets (vecteur, série temporelle, data.frame, etc.). Les commandes de second ordre ont une influence sur des graphiques déjà existants : modifications du titre, de la couleur du fond, etc. et servent essentiellement à améliorer le rendu général.

Voici une liste des principales commandes pour la création de graphiques :

- plot(x) : graphe des valeurs de x (sur l'axe des y) ordonnée par leur position dans le vecteur.
- plot(x,y) : crée un graphe bivarié de x (sur l'axe des x) et de y (sur l'axe des y).
- pie(x) : création d'un graphe en "camembert". Suivant les versions de **R**, on utilise plutôt la commande piechart.
- boxplot(x) : graphique de type "boites et moustaches".
- plot.ts(x) : permet de tracer la graphique d'une série temporelle. Cette dernière peut-être multi variée mais les différentes séries doivent avoir dans ce cas les mêmes fréquences et dates.
- ts.plot(x) : mêmes propriétés que la commandes précédentes sauf que les séries peuvent avoir des dates différentes.
- hist(x) : histogramme des fréquences de x.
- barplot(x) : histogramme des valeurs de x.

La commande lines est également très utile pour tracer des fonctions connues (densité de probabilité, *log*, exponentielle, etc.). Nous aurons souvent l'occasion de l'utiliser dans les planches de TP.

Il est possible de préciser certaines options lors de la création de graphiques. Les options présentées ci-dessus sont communes à la plupart des fonctions introduites ici :

- add si TRUE, superpose le graphique crée à celui existant (valeur FALSE par défaut).
- type : précise la forme des points utilisée : "p" pour des points, "l" pour des lignes, "b" pour des points reliés par des lignes, etc.
- xlim et ylim précise les limites inférieures des axes.
- xlab et ylab : variable de mode caractère, précise le nom à donner aux axes.
- main : précise le titre du graphique créé.

Pour plus de détails : cf l'aide en ligne...

Ces options permettent d'intervenir sur le rendu d'un graphique dès sa création. Il est également possible d'intervenir sur un graphique après sa création. Les commandes permettant de réaliser de telles actions sont dites de second ordre. Voici une liste non-exhaustive des fonctions de ce type les plus utilisées :

- title() : ajoute un titre au graphique courant.

- points(x,y) : ajoute un point à la coordonnée (x, y).
- abline(a,b) : trace un droite de coefficient directeur b et d'ordonnée à l'origine a. Pour un droite horizontale, passant par l'ordonnée y on utilisera abline(h=y). Idem pour une droite vertical en remplaçant h par v.
- mtext(blabla,side=3,...) : ajoute le texte *blabla* dans la marge spécifiée par side. D'autres options sont disponibles.

1.4.2 Manipulation de graphiques

L'ensemble des commandes présentées précédemment permet d'intervenir sur un graphique à la fois. Il est également possible de "configurer" **R** par l'intermédiaire d'un certain nombre de paramètres graphiques. Ce paramétrage peut être réalisé par l'intermédiaire de la fonction par dont une partie des options disponibles est listée ci-dessous :

- bg : permet de choisir la couleur de l'arrière plan par défaut. Il existe 657 couleurs disponibles (colors()) pour toutes les afficher.
- lty : spécifie le type des lignes tracé, 1 pour des lignes continues (par défaut), 2 pour des tirets, 3 pour des points, etc.
- las : permet de paramétrer la façon dont sont disposés les noms des axes, 0 pour parallèles aux axes, 1 pour une notation horizontale, ...

Pour finir, nous allons voir comment gérer les fenêtres graphiques. Il peut en effet être utile de pouvoir travailler sur plusieurs dispositifs graphiques à la fois. Pour ouvrir une fenêtre graphique, on utilise la commande x11() (si aucune fenêtre n'est ouverte, l'exécution d'une des commandes présentées dans la section précédente ouvrira automatiquement une fenêtre).

Il est possible d'ouvrir autant de fenêtre que l'on souhaite. Dans ce cas, il est tout de même nécessaire de connaître le dispositif actif, c'est-à-dire celui qui sera utilisé lors de la prochaine instruction. La commande dev.list affiche la liste de tous les dispositifs ouverts, dev.cur renseigne sur le dispositif courant et enfin dev.set permet de changer de fenêtre.

```
> x11();x11();x11()
> dev.list()
x11 x11 x11
  2   3   4
> dev.cur()
x11
  4
> dev.set(2)
x11
  2
```

Lorsque l'on souhaite gérer plusieurs graphiques à la fois, il peut être gênant d'avoir à manipuler plusieures fenêtres. Une alternative consiste donc à partitionner le dispositif courant en plusieurs morceaux. La fonction layout prend en argument une matrice et partitionne la fenêtre en plusieurs parties.

```
> x11()
> layout(matrix(1:4,2,2))
> layout.show(4)
```

La commande layout.show(4) permet de visualiser la partition ainsi créée. La forme de la partition est complètement conditionnée par la nombre de lignes et de colonnes de la matrice. On pourra également penser à utiliser l'option byrow pour "remplir" différemment le dispositif. L'ensemble des graphiques construits seront affichés successivement sur la partition.

1.5 Fonctions et programmation

Le logiciel **R** permet également à l'utilisateur de créer ses propres fonctions. Cet aspect est particulièrement utile pour effectuer des tâches répétitives.

La création d'une fonction sous **R** est très intuitive. La syntaxe utilisée est très proche de celles du langage de programmation **C**.

```
> exemple1 <- function(n)
+   {
+     x <- 3*n+2
+     print(x)
+ }
> exemple1(1)
[1] 5
> exemple1(10)
[1] 32
```

La fonction print permet d'afficher les informations souhaitée. Comme pour tout langage de programmation, la présentation d'une fonction a énormément d'importance. Pour sauter une ligne sans exécuter les instructions précédentes, on utilisera "SHIFT+ENTREE".

Il est possible de créer des fonctions admettant plusieurs entrées et de différent types : vecteurs, caractères, matrices, data frame, etc. Il sera parfois utile de faire appel à des boucles pour automatiser des tâches longues et répétitives. La fonction permettant de faire la somme de deux vecteurs pourrait par exemple être écrite de la manière suivante :

```
> addition <- function(x,y)
+   {
+    if (length(x) != length(y)) print("erreur") else {
+         z <- numeric(length(x))
+         for (i in 1:length(x)) z[i]<-x[i]+y[i]
+         print(z) }
+   }
> x <- c(1,2,3)
> y <- c(4,5)
> addition(x,y)
[1] "erreur"
> y <- c(4,5,6)
> addition(x,y)
[1] 5 7 9
```

Il est également de faire des boucles "while" dont la syntaxe est :

```
while ( i>min) {
    ...
      }
```

On répète les instructions entre accolades tant que la condition logique i>min est satisfaite.

Une fonction doit être "compilée" à chaque nouveau démarrage de **R**. Si l'on est amené à utiliser régulièrement une fonction, il sera utile de sauvegarder cette dernière dans un fichier texte et d'utiliser un "copier-coller" à chaque nouvelle utilisation.

Chapitre 2

Test de Student-Fisher

On considère deux échantillons de taille respective n_1 et n_2, correspondant aux réalisations de 2 variables indépendantes X_1 et X_2 de moyenne et de variance respectives μ_1, μ_2 et σ_1,σ_2. On s'intéresse dans ce chapitre à la question suivante : peut on affirmer que la moyenne et la variance de X_1 et X_2 sont égales à partir du calculs des moyenne et des variances empiriques ? Ceci revient à tester les hypothèses :

$$\begin{cases} H_0 : \mu_1 = \mu_2 \text{ et } \sigma_1 = \sigma_2 \\ H_1 : \mu_1 \neq \mu_2 \text{ et } \sigma_1 \neq \sigma_2 \end{cases}$$

Le test de Student-Fisher se déroule en deux étapes : on teste dans un premier temps l'égalité des variances à l'aide du test de Fisher-Snedecor. Si l'on accepte l'hypothèse d'égalité, on teste alors l'égalité des moyennes à l'aide du test de Student en supposant que $\sigma_1 = \sigma_2$.

Rappel de cours sur ces 2 tests...

Test de Student-Fisher et R

Pour pouvoir mettre en place ce test sous **R**, nous aurons besoin de deux fonctions : var.test et t.test pour respectivement les tests de Fisher et Student. Ces dernières sont contenues dans le package **stats** chargé par défaut dans la plupart des distributions (library(help="stats") pour plus de détails).

La fonction var.test prend comme arguments deux vecteurs de même taille correspondant aux deux échantillons observés dont la variance est à tester. Il est également utile de renseigner les options suivantes :

- ratio : la valeurs du rapport à tester entre σ_1 et σ_2. Par défaut, vaut 1.
- alternative : une chaîne de caractère spécifiant le type d'hypothèse alternative à choisir parmi *two.sided* (par défaut) pour un test bilatéral, *greater* ou *less*.
- conf.level : renseigne le niveau de confiance pour l'intervalle de confiance affiché en sortie. Le niveau par défaut est 0.95.

Supposons par exemple que l'on dispose des deux échantillons :

```
> x <- c(23.7,21.8,20.6,22.4,21,21,20.2,18.8)
> y <- c(24.8,22.75,22.52,22.8,20.7,23,23.1,22.96)
```

Pour tester l'égalité des deux variances, on utilise :

```
> var.test(x,y)

        F test to compare two variances

data:  x and y
F = 1.7649, num df = 7, denom df = 7, p-value = 0.4711
alternative hypothesis: true ratio of variances is not equal to 1
```

```
95 percent confidence interval:
 0.3533447 8.8156382
sample estimates:
ratio of variances
          1.764925
```

La fonction renvoie la valeur de la statistique de test, le nombre de degré de liberté du numérateur et du dénominateur, la p-valeur, l'intervalle de confiance, etc. Comme pour les tests vu dans le chapitre précédant, il est possible de récupérer l'ensemble des informations calculée dans cette fonction. Il suffit d'affecter la fonction à une variable :

```
> X <- var.test(x,y)
```

et de sélectionner les valeurs souhaitées : statistic pour la valeur de la statistique, parameter pour les degrés de liberté, p.value pour la p-valeur ou encore conf.int pour l'intervalle de confiance. Exemple :

```
> X$p.value
[1] 0.4711269
```

Concernant la manipulation de la loi de Fisher, on notera l'existence des commandes :

- pf(t,r_1,r_2) : donne la valeur de $P(F \leq t)$ pour F suivant une loi de Fisher de paramètres r_1 et r_2.
- qf(q,r_1,r_2) : valeur de t telle que $P(F \leq t) = q$.

Intéressons nous à présent à la mise en place du test de Student (dans l'éventualité ou le test précédant inciterait à conclure que les variances des deux échantillons sont égales). La fonction t.test prend comme argument les deux échantillons à tester. Les options sont essentiellement les mêmes que pour le test de Fisher. Il est cependant nécessaire de préciser que les variances sont supposées égales en donnant la valeur TRUE à la variable var.equal. Dans le cas contraire, c'est une variante du test de Student qui est utilisée.

Reprenons l'exemple précédant. Au vu de la p-valeur, il est raisonnable d'accepter l'hypothèse d'égalité des deux variances. Pour tester l'égalité des deux moyennes, on utilise :

```
> t.test(x,y,var.equal=TRUE)

        Two Sample t-test

data:  x and y
t = -2.5129, df = 14, p-value = 0.02484
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.0420811 -0.2404189
sample estimates:
mean of x mean of y
 21.18750  22.82875
```

On accepte ainsi l'hypothèse d'égalité des moyennes au niveau 1% mais pas au niveau 5%. Il est encore une fois possible d'accéder séparément à l'ensemble des informations fournies par cette fonction.

Chapitre 3

Tests du χ^2

3.1 Test du χ^2 d'indépendance

Petit rappel de cours : tableau de contingence, effectifs théoriques et observés, convergence en loi de la statistique de test, etc. (cf manuscript)

La fonction chisq.test
Cette commande travaille principalement avec le tableau de contingence que l'on souhaite tester. Ce dernier est passé en argument dans la fonction par l'intermédiaire d'une matrice.

```
> A <- matrix(c(12,22,22,25,55,12),2,3,byrow=TRUE)
> A
     [,1] [,2] [,3]
[1,]   12   22   22
[2,]   25   55   12
> chisq.test(A)

        Pearson's Chi-squared test

data:  A
X-squared = 13.7058, df = 2, p-value = 0.001056
```

En retour, on obtient :
- la valeur de la statistique de test (X-squared),
- le nombre de degrés de liberté (df pour degree of freedom),
- la p-valeur.

Au vu de ces informations, une décision peut immédiatement être prise. Il est cependant possible d'avoir accès à l'ensemble des calculs intermédiaires à l'aide des commandes :

```
> X <- chisq.test(A)
> X observed
     [,1] [,2] [,3]
[1,]   12   22   22
[2,]   25   55   12
> X expected
     [,1]     [,2]     [,3]
[1,]   14 29.13514 12.86486
[2,]   23 47.86486 21.13514
> X residuals
           [,1]      [,2]      [,3]
[1,] -0.5345225 -1.321885  2.546903
[2,]  0.4170288  1.031321 -1.987067
```

Dans cette situation, X représente une liste. On choisira observed pour les observations, expected pour les effectifs théoriques et residuals pour les résidus.

Quelques options additionnelles sont disponibles pour la fonction chisq.test :
- correct : Par défaut vaut FALSE. Dans le cas contraire, arrondie les valeurs théoriques à l'entier le plus proche dans le tableau de contingence.
- simulate.p.value : Si cette dernière vaut TRUE, indique que la p-valeur doit être approximée par la méthode de Monte-Carlo. Sinon, la p-valeur proposée et la plus proche pré-enregistrée : on perd donc en précision, ce qui peut nous amener à prendre une mauvaise décision.
- B : Dans le cas où simulate.p.value vaut TRUE, précise le nombre de réplications à utiliser pour approcher la p-valeur.

3.2 Test du χ^2 d'adéquation

Rappel de cours : distribution théorique connue ou non, lois continues et lois discrètes, etc (cf manuscript)

3.2.1 Lois discrètes

Comme pour le test du χ^2 d'indépendance, nous utiliserons ici la fonction chisq.test. La principale différence par rapport à la première partie réside dans les paramètres passés en argument. Cette fois ci on rentrera un vecteur x correspondant au fréquences observées pour les différentes classes et on précisera la probabilité théorique dans le champ p. Supposons que l'on dispose d'un échantillon et que l'on souhaite déterminer si ce dernier suit une loi théorique géométrique de paramètre $0,3$. La syntaxe à utiliser est alors :

```
> x <- c(40,29,8,10,13)
> proba <- c(0.3,0.21,0.147,0.103,0.240)
> chisq.test(x,p=proba)

        Chi-squared test for given probabilities

data:  x
X-squared = 14.4851, df = 4, p-value = 0.005897
```

Avec de tels résultats, on rejette l'hypothèse H_0 au niveau 0.01. La variable Y observée a peu de chance de suivre un loi géométrique. Comme pour le test d'indépendance, il est possible d'avoir accès au effectifs théoriques et au résidus :

```
> chisq.test(x,p=proba)$expected
[1] 30.0 21.0 14.7 10.3 24.0
> chisq.test(x,p=proba)$residuals
[1]  1.82574186  1.74574312 -1.74749578 -0.09347654 -2.24536560
```

Il est également d'approximer la p-valeur par la méthode de Monte-Carlo.

Remarque : Si la probabilité théorique des différentes classes est basée sur l'estimation de p paramètres, le nombre de degré de libertés de la statistique de test est diminué d'autant... ce qui n'est pas pris en compte par la fonction chisq.test. Il sera donc nécessaire de re-calculer *manuellement* la p-valeur.

3.2.2 Lois continues

Travailler avec des lois continues est un peu plus délicat dans la mesure où il n'existe pas de fonction dédiée. Il faut donc, préalablement à l'utilisation de la fonction chisq.test, découper l'échantillon en différentes classes et calculer pour chacune d'entres elles les probabilités associées (correspondant à des intervalles sur $[0,1]$) . Ce travail préliminaire peut-être réalisé à l'aide de la fonction qnorm (pour obtenir les quantiles). La fonction hist sera également très utile puisqu'elle calcule un certain nombre de quantités dont nous avons besoin.

Utilisation avancée de la fonction hist

La commande hist appliquée à un vecteur x crée en fait une liste contenant :

- breaks : les classes crées
- counts :nombre d'observations classe par classe
- density : valeur de la densité classe par classe
- etc.

Par exemple, pour avoir accès au découpage proposé par **R**, on tapera :

```
> hist(x)$breaks
```

Chapitre 4

Régression linéaire

Petit mot sur l'importance de la régression (historique ?)...

4.1 Modèle de régression linéaire simple

4.1.1 Introduction

Rappels sur le modèle linéaire : Y=aX+b+"bruit", retour sur les différentes quantités impliquées, etc.+ prédiction -> cf manuscrit

4.1.2 Fonctionnalités de la commande lm

L'étude d'un modèle de régression peut-être réalisé à l'aide de la commande lm. Cette dernière propose de nombreuses fonctionnalités que nous allons étudier à l'aide d'un exemple simple. Considérons les vecteurs x et y définis par :

```
> x <- c(1,4,6,8,2,4,8,9)
> y <- 2*x + rnorm(8,0,1)
```

Pour obtenir la régression de y en fonction de x, il suffit de taper :

```
> Reg <- lm(y~x)
> Reg

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
   -0.02772      2.00395
```

Utilisée telle quelle, la commande lm renvoie simplement les deux coefficients de régression estimés. A noter : la commande lm admet un certain nombre d'options que nous n'utiliserons pas dans ce cours (voir donc l'aide en ligne pour plus de détails).

Comme pour les commandes chisq.test ou var.test, lm produit plus d'informations que ce qui est affiché. En fait, cette commande est tellement importante qu'une classe particulière a été crée. Pour plus de détail, il est possible d'utiliser :

```
> attributes(Reg)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

```
$class
[1] "lm"
```

Pour accéder à chacune des informations contenues dans la rubrique names, on utiliser Reg$nom, où nom correspond à l'information désirée. Par exemple, pour obtenir les résidus, on tapera :

```
> Reg$residuals
         1          2          3          4          5          6          7
-0.3334642 -0.3041610  0.2804465  1.3167916 -0.5129902  1.1945033 -0.3511980
         8
-1.2899281
```

En ce qui nous concerne, les rubriques les plus utiles seront : coefficients, residuals, fitted.values et model (pour récupérer les données sous la forme d'un data.frame).

Avec toutes ces informations, il est maintenant possible de tracer la droite de régression. On utilise pour cela la commande abline permettant de tracer des droites affines :

```
> plot(x,y)
> abline(Reg$coef)
```

Remarque : La commande plot(Reg) permet d'afficher quatre graphiques, utiles pour une étude approfondie du modèle de régression. Nous n'aborderons pas l'utilisation de cette commande dans ce module.

4.1.3 Test du coefficient de corrélation linéaire

Ce test est équivalent à celui du caractère significatif de la régression (i.e. nullité de la pente de régression). La fonction utilisée pour réaliser cette opération est cor.test. Cette dernière prend en argument les deux vecteurs x et y. Il est également nécessaire de préciser le type de test utilisé par l'intermédiaire de l'option method. Reprenons l'exemple traité précédemment. Dans la mesure où nous nous sommes intéressés au test de Pearson, on utilise :

```
> cor.test(x,y,method="pearson")

        Pearson's product-moment correlation

data:  x and y
t = 16.0598, df = 6, p-value = 3.704e-06
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9357623 0.9980100
sample estimates:
      cor
0.9885675
```

Au vu de la p-valeur, nous sommes amenés à conclure à la non nullité du coefficient de corrélation linéaire.

Rappel : La nullité du coefficient de corrélation linéaire n'implique pas l'indépendance de X et de Y. On peut seulement en conclure qu'il n'y a pas de relation linéaire entre X et Y, d'autre liens sont cependant envisageables.

4.1.4 Prévision et fonctionnalités de la commande predict

Nous allons maintenant nous intéresser à la prévision de nouvelles valeurs. Cette opération peut être réalisée à l'aide de la commande predict. Cette dernière prend comme argument principale un objet dont la classe est issue de la fonction lm. L'instruction :

```
> predict(Reg)
        1         2         3         4         5         6         7         8
 2.391090  8.486159 12.549537 16.612916  4.422780  8.486159 16.612916 18.644606
```

renvoie simplement la valeur des estimateurs $\hat{Y}_i$. Pour obtenir un intervalle de confiance de ces derniers, il suffit d'utiliser :

```
> predict(Reg,interval="confidence")
        fit        lwr       upr
1  2.391090  0.8202835  3.961897
2  8.486159  7.5446918  9.427626
3 12.549537 11.6604158 13.438659
4 16.612916 15.4040754 17.821757
5  4.422780  3.1003714  5.745188
6  8.486159  7.5446918  9.427626
7 16.612916 15.4040754 17.821757
8 18.644606 17.2009493 20.088262
```

On obtient un data.frame précisant les intervalles de confiance pour les termes $\mathbb{E}[Y/X = x]$.

Si on souhaite maintenant prédire de nouvelles valeurs, il suffit de créer un data.frame que l'on rajoutera en argument de la fonction predict. Reprenons l'exemple introduit précédemment. Pour prédire les valeurs du modèle aux points 10 et 11, il suffit de taper les instructions suivantes :

```
> nouveau <- data.frame(x=c(10,11))
> predict(Reg,newdata=nouveau)
       1        2
20.67630 22.70798
```

Il est également possible d'obtenir un intervalle de confiance pour ces prédictions en rajoutant simplement l'option interval="confidence".

Important : Le vecteur contenu dans le data.frame doit avoir le même nom que le vecteur entré en argument dans la commande lm.

Enfin, il est possible d'obtenir des intervalles de confiance non plus pour la valeur moyenne $\mathbb{E}[Y/X = x]$ mais pour la valeur de Y au point $X = x$ c'est-à-dire un intervalle de confiance tenant compte des erreurs éventuelles d'échantillonnage. Il suffit pour cela d'utiliser plutôt l'option interval="prediction".

4.2 Régression linéaire multiple

Dans le cadre de la régression linéaire multiple, les commandes sous **R** restent essentiellement les mêmes que dans le cas de la régression linéaire simple. Introduisons par exemple :

```
> x1 <- rnorm(10,0,1)
> x2 <- rnorm(10,0,1)
> y <- 2*x1-x2+rnorm(10,0,0.5)
```

Pour effectuer la régression de y en fonction des deux variables explicatives $x1$ et $x2$ par la méthode des moindres carrés, c'est encore la commande lm qui intervient :

```
> lm(y~x1+x2)

Call:
lm(formula = y ~ x1 + x2)

Coefficients:
(Intercept)           x1           x2
     0.0140       1.9858      -0.9552
```

L'ensemble des fonctionnalités de la commande lm sont conservées. Il sera en particulier intéressant de vérifier que les graphes des résidus en fonction des variables explicatives ne laissent apparaître aucune tendance.

Chapitre 5

Analyse de variance

5.1 Introduction

Introduction rapide, cf notes de cours...

5.2 L'analyse de variance sous R

L'analyse de variance est essentiellement réalisée à l'aide de deux commandes bartlett.test pour le test d'égalité des variances et aov pour le test de comparaison des moyennes.

La commande bartlett.test prend en argument une liste de vecteurs numériques et renvoie la valeur de la statistique de test, le nombre de degrés de liberté ainsi que la p-valeur du test. Supposons par exemple que l'on dispose de trois échantillons contenus dans les vecteurs E1, E2 et E3. Pour comparer la variance de ces trois vecteurs, il suffit de taper :

```
> bartlett.test(list(E1,E2,E3))
```

L'interprétation des résultats est similaire aux autres tests abordés dans ce cours.

La commande aov prend quand à elle comme argument une formule permettant de préciser le facteur et la variable numérique ainsi qu'un data.frame contenant l'ensemble des observations. Ce dernier doit contenir deux colonnes de même tailles :

- la première colonne contient la modalité du facteur pour chaque élément de l'échantillon,
- la seconde colonne contient les observations

Supposons par exemple que les observations soient regroupées dans un data.frame appelé obs avec une colonne de modalités mod et une d'observations num. L'analyse de variance et obtenue en tapant :

```
> analyse <- aov(num~mod,data=obs)
> summary(analyse)
```

Cette dernière commande affiche le tableau complet de l'analyse de variance avec la p-valeur du test.