TP de Statistiques: Utilisation du logiciel R

Année 2006-2007

Table des matières

Introduction

La liberté, c'est l'obéissance à la loi que l'on sait prescrite.

Jean-Jacques Rousseau.

Présentation

R est un logiciel libre basé sur le logiciel commercial S (Bell Laboratories), avec qui il est dans une large mesure compatible (http ://www.r-project.org). R est un environnement dédié aux statistiques et à l'analyse de données. Le terme environnement signifie que l'ensemble des programmes disponibles forme un tout cohérent, modulable et extensible au lieu d'être une simple association de programmes accomplissant chacun une tâche spécifique. R est ainsi à la fois un logiciel et un langage de programmation, permettant de combiner les outils fournis dans des analyses poussées, voire de les utiliser pour en construire de nouveaux. Un autre avantage est qu'il est très facile de se constituer sa propre boîte à outils que l'on utilisera sur plusieurs jeux de données, et ce sans avoir à réinventer la roue à chaque fois. R est disponible pour beaucoup de plates-formes, dont Unix/Linux, Mac-OS et Windows(95 et +). Pour un tutoriel simple, se référer à celui d'Émmanuel Paradis : R pour les débutants (http ://cran.r-project.org/doc/contrib/rdebuts_fr.pdf).

Démarrage

R fonctionne en mode console, c'est à dire qu'on interagit avec lui en tapant des commandes. Les résultats de ces commandes peuvent s'afficher directement dans la console, ou bien dans des fenêtres (pour les graphiques par exemple, voir figure 1). Sous Windows, on dispose d'une interface graphique, avec une fenêtre console qui est l'équivalent de celle de Linux, mais plusieurs fonctionnalités sont accessibles directement via les menus (dont l'aide !). Par la suite, on se focalisera sur la version Linux en donnant les noms de toutes les commandes utilisées, sachant que sous Windows il y a parfois plus simple...

Quelques règles simples pour travailler sous Linux

Un des avantages de Linux, c'est qu'il est possible de faire chaque chose de plusieurs manières différentes. Cette diversité peut cependant devenir très vite pénible (pour vos chers enseignants). Voici donc quelques règles afin de travailler efficacement. Vous êtes en droit de ne pas les utiliser. Cela impliquera toutefois que vous sachiez ce que vous faites, et vous en serez tenu comme responsable !

Nous travaillerons en mode console. Une console (on dit aussi un terminal), est une application qui possède une simple fenêtre où l'on peut taper du texte. Au démarrage cette fenêtre est vide et possède seulement une invite (= un prompt) qui ressemble à ceci :

```
[user1@babasse user1]$
```

Entre crochets, on trouve tout d'abord le nom de l'utilisateur connecté et celui de l'ordinateur, puis le répertoire courant. Le prompt est ici le dollar. Par défaut, le répertoire courant est au début celui de l'utilisateur qui a lancé la console. On peut ensuite y taper des lignes de commandes, qui seront exécutées après avoir appuyé sur « Entrée ». Le terme « ligne de commande » ne doit pas effrayer, il s'agit souvent du simple nom d'un programme. Pour ouvrir une console, voir figure 2.

Nous allons créer un répertoire de travail, nommé « stats ». La commande a utiliser s'appelle « mkdir » (= **m**a**k**e **dir**ectory = crée un répertoire), et on procède comme suit :

```
mkdir stats
```

On va ensuite se déplacer dans ce répertoire en utilisant la commande « cd » (= **c**hange **d**irectory) :

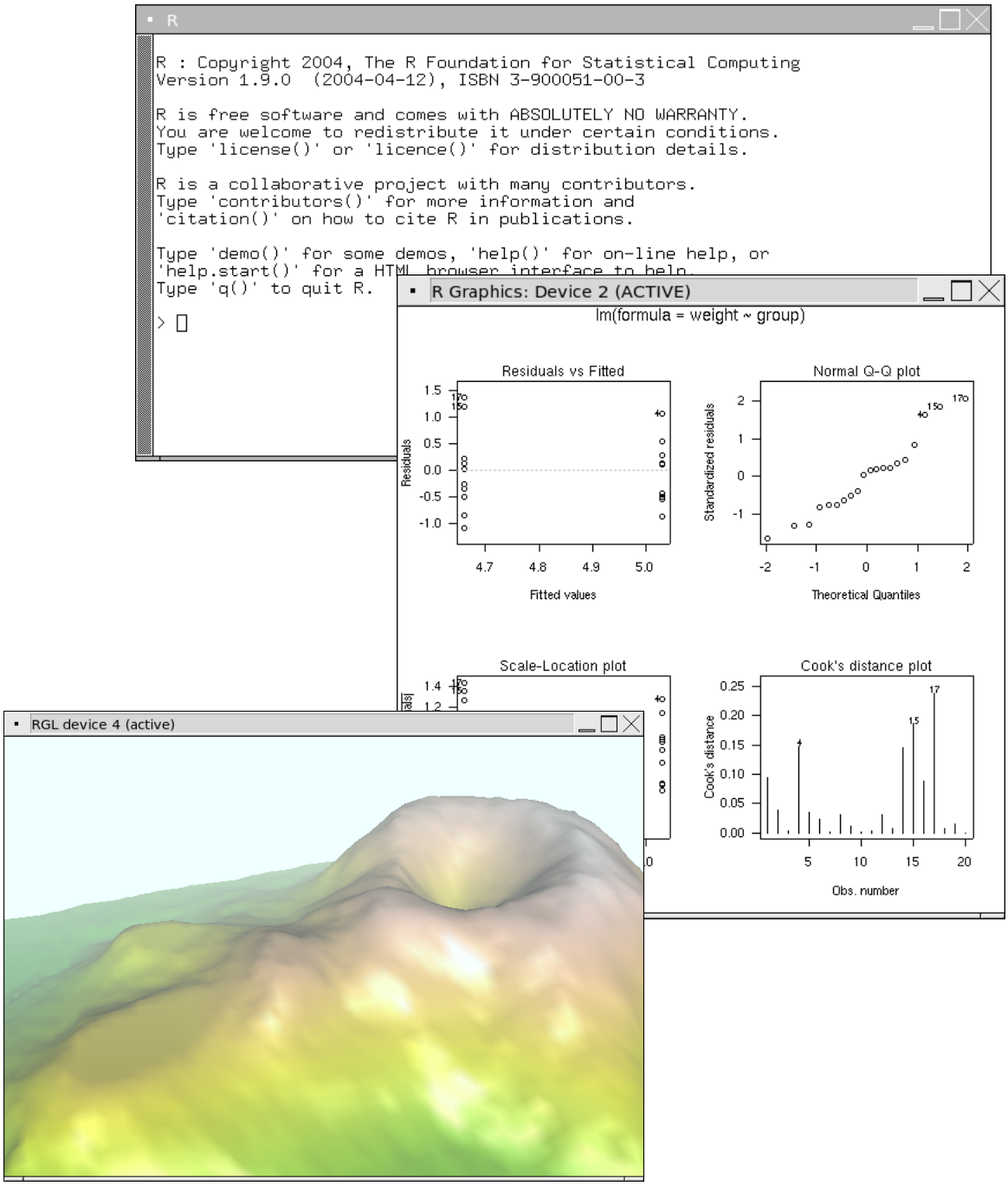


FIG. 1 – Quelques fenêtres sous R : 1) Démarrage de R, 2 et 3) fenêtres graphiques.

```
cd stats
```

Le répertoire courant est maintenant « stats ». On va ensuite créer un sous-répertoire nommé « data », qui contiendra des jeux de données :

```
mkdir data
```

Le répertoire « data » a été créé dans « stats », puisque c'est maintenant le répertoire courant. On peut lister le contenu d'un répertoire en tapant la commande « ls » [1] :

```
[jdutheil@Deedlit jdutheil]$ mkdir stats
```

[1] Le prompt figuré ici(`[jdutheil@Deedlit jdutheil]$`) correspond à celui de l'ordinateur sous lequel ce document a été écrit, le votre est différent et correspond à votre ordinateur !

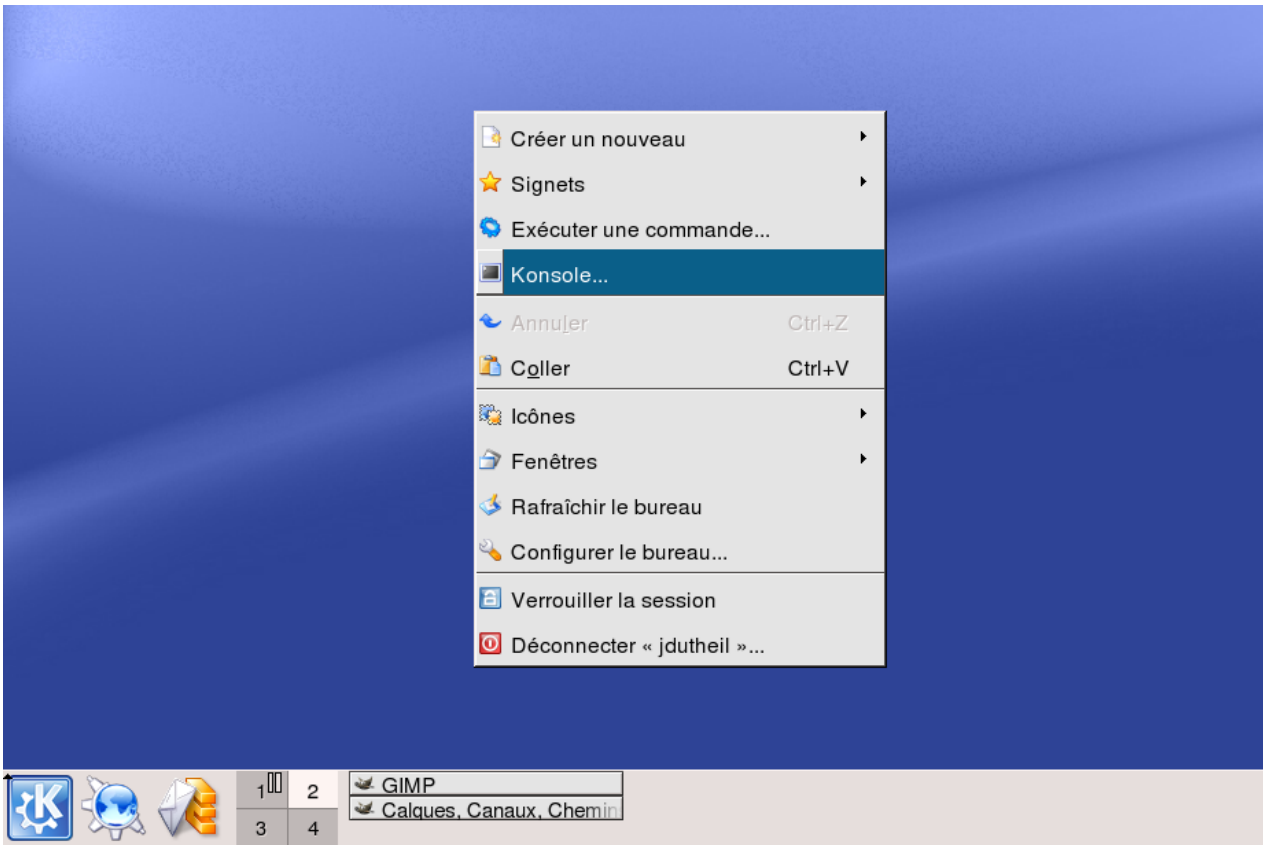


FIG. 2 – Pour lancer un terminal, cliquer sur le bureau avec le boutons droit de la souris. Un menu contextuel apparaît, cliquer sur *konsole* pour lancer la console de KDE.

```
[jdutheil@Deedlit jdutheil]$ cd stats/
[jdutheil@Deedlit stats]$ mkdir data
[jdutheil@Deedlit stats]$ ls
data/
[jdutheil@Deedlit stats]$
```

Le répertoire « stats » contient pour l'instant un seul sous-répertoire « data ». Nous allons en créer un deuxième nommé « scripts », qui sera utile pour sauvegarder votre travail :

```
mkdir scripts
```

Démarrer et arrêter R

Fermer toutes les consoles ouvertes. Lancer un nouvelle console. Se déplacer dans le répertoire de travail « stats » :

```
cd stats
```

Lancer R en tapant la commande « R ». Quelque chose comme ceci doit apparaître :

```
[jdutheil@Deedlit jdutheil]$ R
R : Copyright 2006, The R Foundation for Statistical Computing
Version 2.3.1 (2006-06-01)
ISBN 3-900051-07-0

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.
```

```
R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.

>
```

Ce message peut varier, suivant la version installée, l'ordinateur utilisé, *etc*.

Important : Le prompt a changé, il s'agit maintenant du signe $>$. Cela signifie que vous êtes sous R et non dans la console. Le fonctionnement est similaire, excepté que la console attend des lignes de commande R et non des lignes de commande de Linux.

La première commande que nous allons apprendre est celle pour arrêter R et retourner à la console :

```
1 q()
```

Plusieurs remarques de portée générale :

1. Dans ce document, toutes les commandes R apparaîtront dans un encadré avec lignes numérotées, comme ci-dessus. Le prompt ($>$) n'est pas figuré.
2. Une commande R comprend toujours des parenthèses. Parfois, comme ici, ces parenthèses seront vides. Dans la plupart des commandes cependant, elles contiendront des arguments (on dit aussi paramètres).

Une fois appuyé sur 'Entrée', la question suivante apparaît :

```
Save workspace image? [y/n/c]:
```

R vous demande si vous voulez sauvegarder l'espace de travail. Les réponses possibles sont :
y = yes (oui !)
n = no (non !)
c = cancel (annulé !)
Si vous répondez « oui » (en tapant « y » puis « Entrée »), R vas enregistrer votre travail. Lorsque vous relancerez R, vous retrouverez ce que vous avez fait la fois précédente. Les séances de TP sont largement indépendantes, aussi vous n'aurez pas besoin de ce que vous avez fait la séance d'avant. Néanmoins, il est recommandé de sauvegarder votre travail, particulièrement si vous ne l'avez pas terminé la séance précédente.

Important !
- Si vous quittez R en cliquant sur la croix de la console, cela revient à avoir choisi l'option « no ».
- Ce système ne permet que de sauvegarder le contenu de la mémoire de R, ainsi que les commandes précédemment tapées. Il est très peu pratique pour garder une trace de votre travail, et ne permet pas de le déplacer sur un autre ordinateur par exemple. Nous allons donc voir une autre manière de sauvegarder, que nous vous RECOMMANDONS VIVEMENT D'UTILISER.

Sauvegarder son travail : utilisation des scripts

Il est possible d'écrire des commandes R dans un fichier, et de demander à R de lire ce fichier et d'exécuter toutes les commandes qui sont dedans. Un tel fichier de commande s'appelle un *script*. Ce système est très intéressant car il constitue un très bon et très simple moyen de sauvegarde. Il permet aussi de stocker toute une analyse, et ainsi de trouver facilement d'éventuelles erreurs. Un autre intérêt, et non des moindres, est qu'il permet de refaire sans se fatiguer une analyse sur un autre jeu de données, ou bien lorsqu'on se rend compte d'une erreur dans les données. En pratique, vous aurez surtout besoin de ce système lorsque vous vous attaquerez aux projets. Son utilisation précoce vous garrantira un gain de temps pour la suite.

Voici comment créer un script :

1. Ouvrir une nouvelle console
2. Se déplacer dans le répertoire « stats/scripts » que vous avez créé :

   ```
   cd stats/scripts
   ```

3. Ouvrir un éditeur de texte. Il en existe plusieurs, nous allons utiliser « kwrite », l'éditeur de KDE :

   ```
   kwrite &
   ```

   Note : ne pas oublier le "&" à la fin, qui dit à Linux de lancer le programme et de retourner à la console ensuite. Si vous l'oubliez, vous devrez fermer l'éditeur avant de pouvoir réutiliser la console.

« kwrite » ressemble beaucoup au bloc-notes de Windows. Vous pouvez y taper vos commandes, puis sauvegarder votre travail en utilisant le menu « Fichier → enregistrer sous ». Enregistrer votre fichier dans le répertoire « stats/scripts ». Il est recommandé de nommer les scripts « nom_de_fichier.R » ou « nom_de_fichier.r », mais ce n'est pas obligatoire. Vous pouvez simplement les nommer « nom_de_fichier.txt ».

Vous pouvez ensuite copier-coller vos commandes dans R (CRTL + c pour copier, CTRL + v pour coller). Vous pouvez ainsi exécuter plusieurs lignes de commande à la fois, et refaire ainsi toute un analyse très rapidement. Nous verrons qu'il existe aussi une autre manière de faire.

Tout ce qui suit le cractère « ♯ » sera ignoré par R, ce qui permet de « documenter » votre code. Là encore, ce n'est pas obligatoire, mais VIVEMENT RECOMMANDÉ, de suivre le patron suivant :

```
# Fichier: TP1.R
# Auteur: Etudiant Studieux
# Date: 30 septembre 2006
# Objet: code du TP N°1

#+----------------+
#|   Exercice 1   |
#+----------------+

blabla
```

Vous pouvez organiser votre répertoire « scripts » en créant un fichier par séance de TP par exemple.

Obtenir de l'aide

La deuxième commande R de ce document est la commande « help » . Elle prend en argument le nom de la commande pour laquelle on veut des informations. Par exemple :

```
help(q)
```

Par défaut, l'aide est affichée dans la console. On peut utiliser les flèches pour faire défiler le texte, et on tape « q » (sans parenthèses cette fois) pour quitter l'aide et retourner dans R [2]. Une manière plus conviviale consiste à utiliser l'aide au format « html » (web). Pour cela, on dit d'abord à R d'ouvrir un navigateur (ici « konqueror », le navigateur de KDE) :

```
help.start(browser="konqueror")
```

Cette commande a pour effet d'ouvrir une fenêtre externe (après un petit laps de temps). On peut alors naviguer dans l'aide comme sur une page web ordinaire. Ensuite, si on tape

```
help(q)
```

l'aide s'affichera automatiquement au format html (voir figure 3). Note : l'aide est en anglais !

[2] Attention ! ne pas confondre le « q » pour quitter l'aide (qui permet juste de basculer l'affichage), avec le « q() » pour quitter R ! Ce système est propre à Linux. Sous Windows, l'aide s'affiche dans une fenêtre à part.

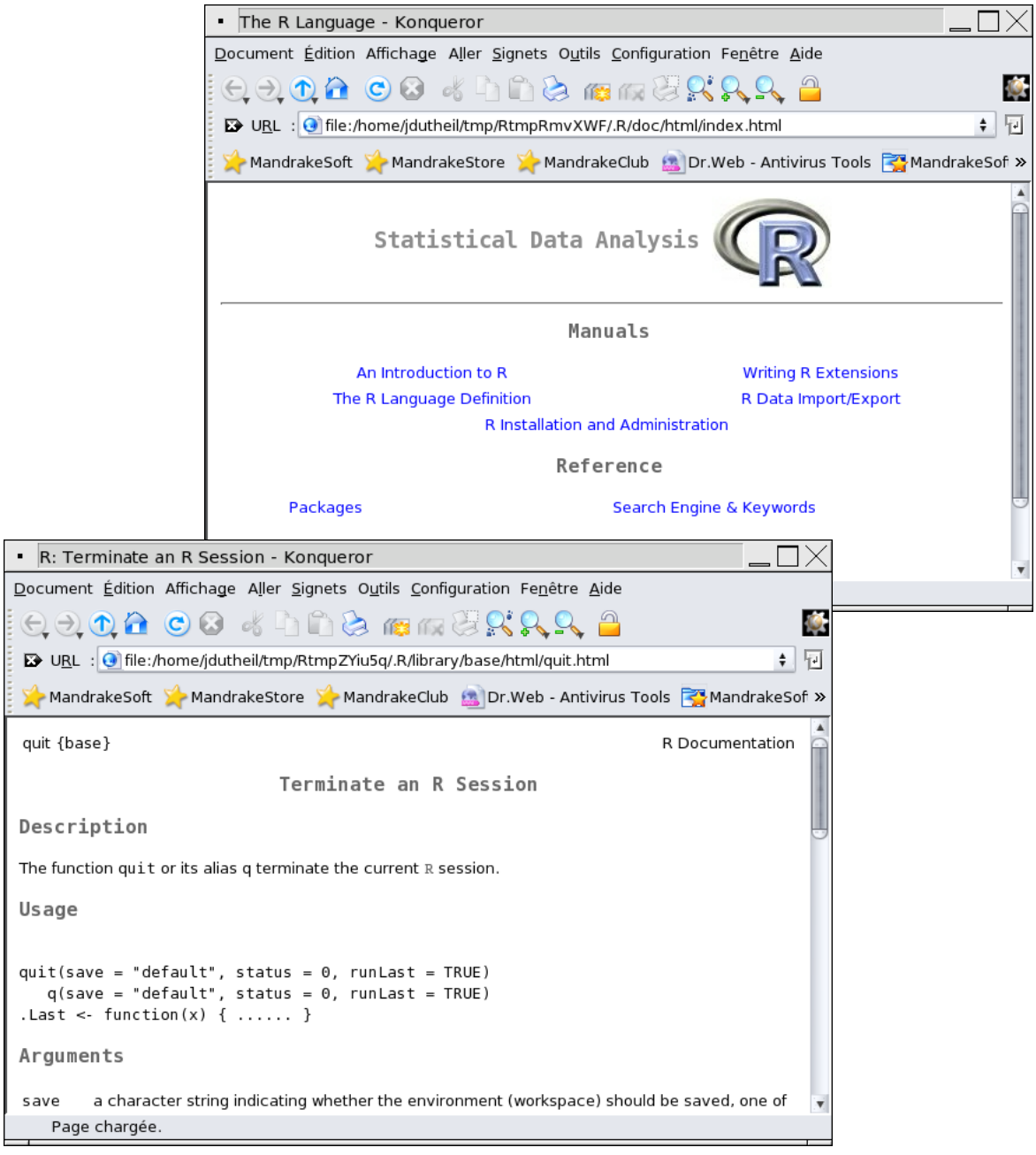


FIG. 3 – Aide au format HTML.

Chapitre 1

Premiers pas avec R : présentation du logiciel et manipulation de données

Qu'est-ce que R ?

Cette première séance fait un petit tour d'horizon des capacités de R. On expose quelques notions de base, fastidieuses mais nécessaires pour la suite.

1.1 R est une calculatrice

R permet de faire les opérations de calcul élémentaire. Essayez les commandes suivantes :

```
3+5
2*4
3-8
2.1-6*2.5
3*2-5*(2-4)/6.02
2^2
```

etc.

1.2 R est une calculatrice scientifique

R permet de faire des calculs plus élaborés. Il utilise pour cela des fonctions. Plusieurs fonctions prédéfinies sont disponibles. Que font les fonctions suivantes ?

```
sqrt(4)
abs(-4)
log(4)
sin(0)
exp(1)
round(3.1415,2)
```

R possède aussi en mémoire la valeur de quelques constantes mathématiques :

```
pi
cos(2*pi)
```

1.3 R est un langage de programmation

1.3.1 Création de variables

On peut stocker en mémoire des données, résultats, etc. :

```
x<-3.14
```

On a ainsi créé une variable de nom *x* contenant la valeur 3.14. La flèche, obtenu en tapant $<$ et $-$ est appelée opérateur d'affectation. Cet opérateur dit que la variable *x* doit contenir la valeur 3.14. Si cette variable n'existe pas, elle est créée. Si elle existe, son contenu est remplacé par la nouvelle valeur.

Pour rappeler le contenu d'une variable, il suffit de taper son nom :

```
> x
[1] 3.14
```

Nous reviendrons plus tard sur la signification du "[1]".

Les variables ainsi créées peuvent être appelées n'importe quand, y compris lors de la création de nouvelles variables :

```
y<-2*x
```

```
> y
[1] 6.28
```

Toutes les variables créées sont stockées dans la mémoire de R. On peut obtenir la liste des objets stockés par la fonction `ls` (**ls**te)[1] :

```
ls()
```

```
> ls()
[1] "x" "y"
```

On peut effacer une variable avec la fonction `rm` (**rm**ove) :

```
rm("x")
ls()
```

```
> rm("x")
> ls()
[1] "y"
```

1.3.2 Notion d'objet

R est un langage dit "orienté objet". Cette notion a plusieurs implications importantes que nous ne détaillerons pas ici. Ce qu'il faut retenir, c'est qu'on peut manipuler des variables de plusieurs *types* dans R. Au cours des TP, nous en utiliserons plusieurs.

Le premier type, le plus simple, est le *vecteur*. Un vecteur est un tableau à une dimension, par exemple [1,2,5,3,-5,0] est un tableau de 6 nombres.

Création d'un vecteur

Voici comment on crée un tel tableau dans R :

```
t<-c(1,2,5,3,-5,0)
```

La fonction `c` prend en argument une liste de valeurs et les colle dans un tableau. Ici, ce tableau est stocké dans la variable *t*. On peut afficher son contenu en tapant son nom :

```
> t
[1]  1  2  5  3 -5  0
```

Il y a d'autres manières de créer des vecteurs... Étudiez les lignes de commande suivantes :

```
rep(2.34, 5)
2:7
6:1
seq(from=1, to=2, by=0.25)
seq(from=1, to=2, length=7)
```

[1] Noter la différence avec la commande ls de la console, qui liste le contenu d'un répertoire.

Important : Lorsqu'on affiche un vecteur de grande taille, R l'écrit sur plusieurs lignes :

```
seq(from=1, to=100, by=1)
```

```
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100
```

Le numéro entre crochets désigne l'**indice** (= la position dans le vecteur) de la première valeur affichée sur la ligne. Il est à noter que dans R, tous les nombres sont des vecteurs de taille 1. C'est pourquoi quand vous affichez une valeur (un vecteur de taille 1 donc), un 1 entre crochets apparaît au début de la ligne.

Opérations sur les vecteurs

Tous les opérateurs et les fonctions vues précédemment peuvent être utilisés sur des vecteurs.

Exercice 1 *Opérations sur les vecteurs*

- *créer un vecteur x quelconque. Que font les commandes suivantes ?*

```
x+1
x*2
log(x)
x>2
```

- *créer un vecteur y de même taille que x. Que font les commandes suivantes ?*

```
x+y
x*y
```

- *créer un vecteur z de taille différente de x. Que font les commandes suivantes ?*

```
x+z
x*z
```

Accès aux éléments d'un vecteur

On peut accéder aux éléments d'un vecteur grâce à l'opérateur "[]".

```
x<-c(1,3,-2,0,1.3,-6.43)
x[1]
x[3]
```

```
> x<-c(1,3,-2,0,1.3,-6.43)
> x[1]
[1] 1
> x[3]
[1] -2
```

Cet opérateur peut prendre en arguments des valeurs négatives, ou même plusieurs valeurs, on parle alors de *masque*. Examinez les commandes suivantes. Que font-elles ?

```
x[-4]
x[c(1,3)]
x[c(-4,-5,-1)]
x[c(TRUE,FALSE,TRUE,TRUE,FALSE,FALSE)]
x[c(TRUE,FALSE)]
x[x<1]
```

1.4 R est un langage pour les statistiques

1.4.1 Fonctions statistiques

R possède plusieurs fonctions statistiques. Que font les fonctions suivantes ?

```
sum(x)
length(x)
mean(x)
sd(x)
median(x)
var(x)
summary(x)
```

Exercice 2 *Moyenne et variance*
- *créez un vecteur x de taille* > 10
- *calculez la moyenne des éléments de x, sans utiliser la fonction* `mean`*. Comparer les résultats à ceux de la fonction* `mean`*.*
- *calculez la variance des éléments de x, sans utiliser la fonction* `var`*. Comparez les résultats à ceux de la fonction* `var`*.*

1.4.2 Importation de données dans R

Format des données

R lit des fichiers *plats*, c'est-à-dire contenant seulement du texte et non des données binaires. En d'autres termes, pas de données Word ou Excel ! Ces fichiers plats sont assez simples : chaque ligne correspond à une ligne du tableau, et les colonnes sont séparées par un espace (voir figure 1.1). Les colonnes peuvent aussi être séparées par des tabulations, des virgules, des points-virgules... A noter que la première ligne contient les noms des colonnes.

```
"beak_size" "survival"
10,81 1
8,74 0
10,04 0
9,75 0
9,76 0
9,55 0
8,7 0
9,62 0
9,22 0
8,79 0
9,61 0
9,02 0
7,85 0
9,01 0
...
```

FIG. 1.1 – Exemple de fichier de données 'texte'.

R peut lire n'importe quel fichier texte suivant ce format. On utilise pour ce faire la commande `read.table`. Celle-ci prend plusieurs arguments, dont le nom du fichier à lire et des arguments optionnels :
- **`sep`** (caractère) caractère de séparation de colonnes ;
- **`dec`** (caractère) caractère de décimale ("." ou ",") ;
- **`quot`** (caractère) caractère de délimitation du texte("\"" ou "'" ou "" (vide)) ;
- **`header`** (booléen) dit si la première ligne est une entête ;
- **`row.names`** (entier) numéro de la colonne à considérer comme noms de lignes. Si absent, les lignes sont numérotées.
- etc ...

Jusqu'à la fin de cette séance, nous allons travailler sur un jeu de données réel. Ce jeu de données, représenté partiellement sur la figure 1.1 comporte les tailles de bec de pinsons de Darwin dans l'archipel des Galapagos. Pour chaque oiseau capturé, on mesure la largeur du bec dans une première campagne de capture. L'oiseau est marqué puis une deuxième campagne de capture est organisée. Si au terme de cette deuxième campagne un oiseau bagué n'est pas revu, il est considéré comme mort. Chaque ligne du tableau représente un oiseau. La première colonne contient la taille du bec, et la deuxième colonne la survie (un 1 signifie que l'oiseau a été revu lors de la deuxième campagne).

Ce jeu de données est stocké dans un fichier nommé "beak_size.cvs", téléchargeable depuis le site web de l'université. Enregistrez le dans votre répertoire 'data'.

On importe ensuite ce fichier dans R avec la commande suivante :

```
becs <- read.table("data/beak_size.csv", header=TRUE, sep="\t", dec=",")
```

On précise ici que le fichier possède une entête, que les colonnes sont séparées par des tabulations et que les décimales sont codées par une virgule (et non un point). Le jeu de données sera stocké dans la variable *becs*.

Notion de dataframe

La variable *becs* est un objet de type *dataframe*, analogue à une feuille de calcul Excel ou OpenOffice. Tout comme les vecteurs, les dataframes sont des objets, mais plus élaborés et spécialement désignés pour le stockage de données. Il s'agit plus ou moins d'un tableau à 2 dimensions (une matrice), mais avec en plus des noms de colonnes, des noms de lignes, *etc*. Très généralement, et ce sera toujours le cas en ce qui nous concerne, les lignes d'un dataframe seront des *individus* (ici des pinsons) et les colonnes des *variables* (ici la taille du bec et la survie). Ainsi, à la ligne 5, colonne 2, on a la survie du pinson n°5. On accède à cet élément en tapant

```
becs[5,2]
```

L'opérateur [] fonctionne exactement comme pour les vecteurs, sauf qu'il prend 2 arguments séparés par une virgule. Le premier désigne les lignes, le second les colonnes. Étudiez les lignes de commandes suivantes :

```
becs[c(1,2,5,10),1]
becs[-(10:750),1]
```

Il est aussi possible d'omettre le terme des lignes ou des colonnes (mais attention à ne pas oublier la virgule !). Que font les commandes suivantes :

```
becs[4,]
becs[,2]
```

Il est également possible d'utiliser les noms de lignes et de colonnes au lieu des numéros. Vérifiez que les 2 lignes de commandes suivantes sont équivalentes :

```
becs[1:5, 1]
becs[1:5, "beak_size"]
```

On peut également utiliser la syntaxe suivante :

```
becs$beak_size
```

Il existe beaucoup d'autres fonctionnalités propres aux dataframes. Nous nous contenterons de celles-ci pour le moment. Prenez simplement note des fonctions suivantes :

```
dim(becs)
nrow(becs)
ncol(becs)
row.names(becs)
col.names(becs)
```

1.4.3 Représentations graphiques

Finalement, pour terminer cette première séance, nous allons faire un petit tour des fonctionnalités graphiques de R.

La fonction `plot`

La fonction `plot` est la fonction générique de graphe. Littéralement, elle représente graphiquement un objet. son résultat dépend donc du type de l'objet lui-même. Comparer son effet sur un vecteur :

```
plot(1:10)
```

à celui sur un dataframe :

```
plot(becs)
```

Il est possible de paramétrer l'affichage obtenu (couleurs, *etc*). Voici un bref aperçu des options de la fonction `plot`, vous pouvez regarder l'aide pour plus de détails. Nous aurons de toute façon l'occasion de revenir sur l'utilisation de cette fonction.

- **`type`** (chaîne de caractères) "p" pour points, "l" pour lignes, "b" pour les deux (both) ;
- **`main`** (chaîne de caractères) un titre pour le graphique ;
- **`sub`** (chaîne de caractères) un sous-titre pour le graphique ;
- **`xlab`** (chaîne de caractères) un titre pour l'axe des abscisses ;
- **`ylab`** (chaîne de caractères) un titre pour l'axe des ordonnées ;
- **`col`** (vecteur de chaînes de caractères) la ou les couleur(s) à utiliser ;
- etc ...

Essayez la commande suivante :

```
plot(becs, xlab="Taille␣du␣bec", ylab="Survie",
     col="blue", main="Survie␣des␣Pinsons")
```

Cette représentation n'est pas très satisfaisante, on aurait préféré mettre la survie en abscisse. C'est bien sûr possible car `plot` accepte qu'on lui donne en argument des vecteurs d'abscisses et d'ordonnées :

```
plot(x=becs[,"survival"], y=becs[,"beak_size"],
     xlab="Taille␣du␣bec", ylab="Survie",
     col="blue", main="Survie␣des␣Pinsons")
```

Note : on aurait pu obtenir un résultat similaire en échangeant les colonnes dans *becs*, mais cette manière de faire est plus générale. Il en existe aussi une autre :

```
plot(becs[,"beak_size"] ~ becs[,"survival"],
     xlab="Taille␣du␣bec", ylab="Survie",
     col="blue", main="Survie␣des␣Pinsons")
```

Le symbole '$\sim$' signifie "en fonction de". Nous le reverrons plus loin, notamment pour le modèle linéaire.

Un autre problème avec notre représentation est que l'axe des abscisses est gradué (0, 0.2, 0.4, ..., 1). Cette graduation est inutile car nous savons que nous avons seulement les valeurs 0 et 1. En fait, ces valeurs 0 et 1 sont un code que nous avons défini (1 pour vivant, 0 pour mort). La valeur numérique n'a pas de sens, nous aurions très bien pu convenir de "A" pour vivant et "B" pour mort, ou même des mots "vivant" et "mort". En statistiques, une telle variable est dite *discrète* par opposition aux variables *continues* (comme la taille du bec). Dans R, les variables continues sont codées par des vecteurs, les variables discrètes par des *facteurs*. Un facteur est très similaire à un vecteur, et la plupart de ce qu'on a dit pour les vecteurs reste vrai pour les facteurs. Un facteur possède simplement en plus la liste des valeurs possibles qu'il peut prendre (ici "vivant" et "mort", ou "1" et "0"), on parle de *modalités*. Nous reviendrons également sur ce point.

Ce qui nous importe pour le moment, c'est que notre variable "survival" est un vecteur et non un facteur. Ceci est du au fait que, par défaut, R considère une variable numérique comme un vecteur. Nous n'aurions pas eu ce problème si on avait écrit dans le fichier "A" et "B" au lieu de "0" et "1". Néanmoins, pas besoin de tout changer, il suffit de dire à R de convertir la variable en facteur : ceci est effectué par la fonction `as.factor`.

```
plot(x=as.factor(becs[,"survival"]), y=becs[,"beak_size"],
     xlab="Taille␣du␣bec", ylab="Survie",
     col="blue", main="Survie␣des␣Pinsons", type="l")
```

Il est possible d'effectuer la conversion en facteur définitivement :

```
becs[,"survival"] <- as.factor(becs[,"survival"])
plot(x=becs[,"survival"], y=becs[,"beak_size"],
     xlab="Taille␣du␣bec", ylab="Survie",
     col="blue", main="Survie␣des␣Pinsons", type="l")
```

Si vous demandez à R d'afficher la variable "survival" :

```
becs[,"survival"]
```

vous voyez que l'affichage a changé (un peu !) :

```
> becs[,"survival"]
  [1] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [38] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
...
[704] 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[741] 1 0 1 1 1 1 1 1 1 1 0
Levels: 0 1
```

Il a rajouté la ligne `Levels: 0 1`, qui affiche les *modalités* évoquées plus haut. Il est possible d'afficher et de changer le nom des modalités avec la fonction `level` :

```
levels(becs[,"survival"]) <- c("Mort", "Vivant")
becs[,"survival"]
```

```
  [1] Vivant Mort   Mort   Mort   Mort   Mort   Mort   Mort   Mort   Mort
 [11] Mort   Mort   Mort   Mort   Mort   Mort   Mort   Mort   Mort   Mort
...
[721] Mort   Mort   Vivant Vivant Mort   Mort   Vivant Vivant Vivant Vivant
[731] Vivant Vivant Vivant Vivant Vivant Vivant Vivant Vivant Vivant Vivant
[741] Vivant Mort   Vivant Vivant Vivant Vivant Vivant Vivant Vivant Vivant
[751] Mort
Levels: Mort Vivant
```

Vous aurez remarqué que l'affichage a changé : `plot` a reconnu que l'abscisse est un facteur, et a utilisé un affichage en "boîte à moustaches". Ceci est lié au fait que `plot` est une fonction *générique*, il existe d'autres fonctions qui permettent d'effectuer pas mal de représentations (voir la figure 1.2 pour quelques exemples). Nous allons en voir une qui permet d'effectuer des histogrammes.

Création d'histogrammes

Les histogrammes sont effectués par la fonction `hist`. Elle prend en argument un vecteur numérique :

```
hist(becs[,"beak_size"])
```

Plusieurs arguments peuvent aussi être passés :

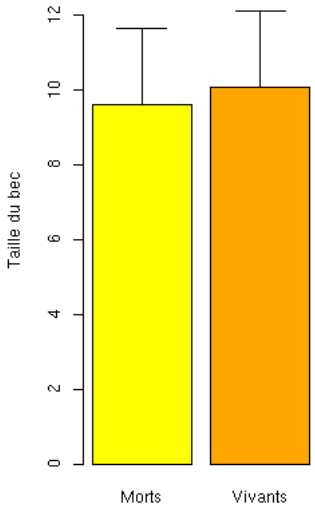
- **`freq`** (booléen) dit si les barres représentent les effectifs (TRUE) ou des pourcentages (FALSE) ;
- **`breaks`** (entier) dit le nombre (approximatif) de classes à utiliser ;
- **`labels`** (booléen) dit s'il faut étiqueter les barres ;
- **`main`** (chaîne de caractères) un titre pour le graphique ;
- **`xlab`** (chaîne de caractères) un titre pour l'axe des abscisses ;
- **`ylab`** (chaîne de caractères) un titre pour l'axe des ordonnées ;
- **`col`** (vecteur de chaînes de caractères) couleurs pour les barres ;
- etc ...

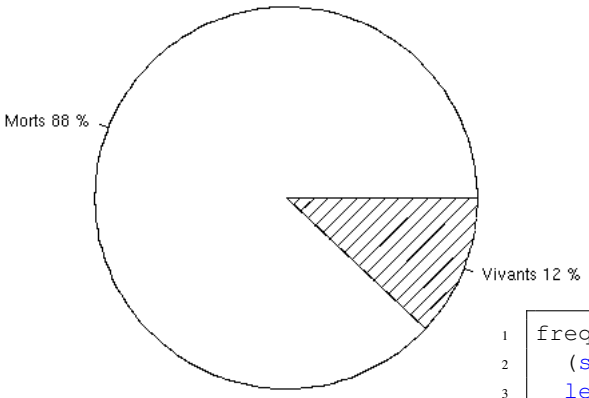
(se référer à la documentation pour une liste complète).

```
hist(becs[,"beak_size"], breaks=20, labels=TRUE, col="red",
     xlab="Taille du bec", ylab="Nombre d'observations",
     main="Pinsons de Darwin : taille du bec")
```

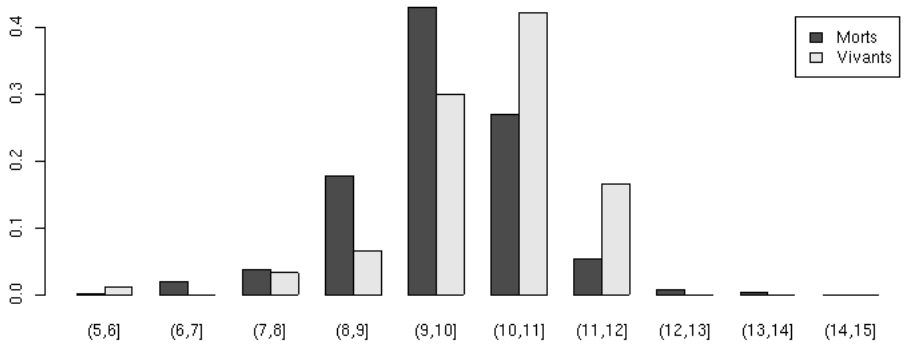
Exercice 3 *Concours d'esthétique.*
Tracer la courbe $f(x) = sin(x)$ pour $x \in [-2\pi, 2\pi]$, de telle sorte qu'elle soit le plus joli possible !

```r
means<-sapply(
  (split(becs[,"beak_size"], becs[,"survival"])),
  mean)
sds  <-sapply(
  (split(becs[,"beak_size"], becs[,"survival"])),
  sd)
names(means) <- c("Morts", "Vivants")
bp<-barplot(means, col=c("yellow", "orange"),
            ylim=c(0,13), ylab="Taille du bec")
f <- function(i) {
  x <- bp[i,]
  y <- means[i]+1.96*sds[i]
  lines(c(x,x,x-0.25,x+0.25), c(means[i], y, y, y))
}
f(1)
f(2)
```






```r
freqs<-sapply(
  (split(becs[,"beak_size"], becs[,"survival"])),
  length)
names(freqs) <- c("Morts", "Vivants")
pie(freqs, labels=paste(names(freqs),
    round(freqs*100/sum(freqs)),"%"),
    density=c(0,10))
```




```r
tailles<-cut(becs[,"beak_size"], breaks=5:15)
survies<-becs[,"survival"]
levels(survies) <- c("Morts", "Vivants")
t<-table(survies, tailles)
t[1,] <- t[1,]/sum(t[1,])
t[2,] <- t[2,]/sum(t[2,])
barplot(t, beside=TRUE, legend=TRUE)
```

FIG. 1.2 – Exemple de graphiques obtenus sous *R* Le code est donné en guise d'exemple, il n'est pas à étudier à ce stade.

Chapitre 2

Distributions et tests classiques

Dans ce TP nous allons voir comment effectuer des tests sous R.

2.1 Distributions

R possède plusieurs distributions de probabilité en mémoire, sous forme de fonctions. Nous verrons comment utiliser ces fonctions pour la loi normale, puis nous indiquerons les autres fonctions disponibles pour les autres lois, leur fonctionnement étant très similaire.

2.1.1 Distribution normale

La fonction `pnorm` est la fonction de répartition d'une loi normale. Elle prend en argument une valeur q (q pour quantile) et retourne la probabilité de l'évènement $-\infty < X \leqslant q$.

La fonction `qnorm` (fonction quantile) est la fonction inverse : elle prend en argument une probabilité p et renvoie q tel que $Pr(-\infty < X \leqslant q) = p$.

Ces deux fonctions prennent aussi deux autres arguments qui sont les paramètres de la loi normale : **`mean`** son espérance et **`sd`** son écart-type, initialisés par défaut à 0 et 1 (loi centrée réduite). Essayez les lignes de commande suivantes :

```
pnorm(0)
qnorm(0.5)
pnorm(1, mean=1, sd=0.1)
qnorm(0.5, mean=1, sd=0.1)
```

La fonction `dnorm` quant à elle prend les mêmes arguments que la fonction `pnorm`, mais renvoie la densité et non la fonction de répartition.

Finalement, la fonction `rnorm` génère des nombres aléatoires selon une loi normale (le paramètre **`n`** permet de spécifier le nombre de tirages à effectuer).

Exercice 4 *Calculs de probabilités*

Pour une loi normale d'espérance 5 et de variance 2 :

1. *Faire une représentation graphique de sa fonction de répartition et de sa densité sur* $[0, 10]$.
2. *Calculer la probabilité des évènements :*
   - $X \leqslant 0$
   - $X \leqslant 5$
   - $-1 < X \leqslant 3$
   - $X > 10$
3. *Calculer entre quelles valeurs 95% des tirages de* X *sont compris.*
4. *Même question que la précédente, mais pour une loi normale centrée réduite.*

2.1.2 Autres distributions

Voir le tableau 2.1 pour un résumé des distributions disponibles.

NB : Les fonctions `p*` et `q*` possèdent également un argument **`lower.tail`** qui par défaut prend la valeur 'vraie'. Cette option correspond au calcul de $Pr(X \leqslant x)$, alors que `lower.tail=FALSE` correspond au calcul $Pr(X > x) = 1 - Pr(X \leqslant x)$.

Exercice 5 *Tests 'à la main'...*

1. *Générer un vecteur u de 10 nombres aléatoires selon une loi normale d'espérance -1 et de variance 1, et un vecteur v d'espérance 1 et de variance 2.*
2. *Estimer les moyennes et variances des distributions à partir de ces tirages.*
3. *Comparaison de moyennes :*
   (a) *Calculer la statistique de Student :*
   $$t_{obs} = \frac{\hat{\mu_1} - \hat{\mu_2}}{\sqrt{\frac{\hat{\sigma^2}}{n_1} + \frac{\hat{\sigma^2}}{n_2}}},$$
   avec
   $$\hat{\sigma^2} = \frac{(n_1 - 1)\hat{\sigma_1^2} + (n_2 - 1)\hat{\sigma_2^2}}{n_1 + n_2 - 2}.$$
   (b) *Calculer la probabilité, sous l'hypothèse nulle d'égalité des moyennes, que cette statistique soit au moins égale à celle observée. La différence est-elle significative ?*
4. *Comparaison de variances :*
   (a) *Calculer le ratio des variances.*
   (b) *Calculer la probabilité, sous l'hypothèse nulle d'homoscédasticité, que le ratio attendu soit supérieur ou égal au ratio observé (rappel : le rapport de deux variances de lois normales suit une loi de Fischer, de paramètres $n_1 - 1$ et $n_2 - 1$, où n_1 et n_2 sont les tailles des échantillons observés). Les variances sont-elles significativement différentes ?*

2.2 Quelques tests

Grâce à ses fonctions de calcul numérique avancé telles que les fonctions de distributions de probabilité, R permet d'effectuer à peu près n'importe quel test statistique.

Néanmoins la procédure peut être laborieuse lorsque l'on effectue certains tests très utilisés (dits 'classiques') comme ceux proposés dans l'exercice 2. C'est pourquoi R possède un certain nombre de tests tout prêts sous forme de fonctions. Nous allons en voir quelques uns.

2.2.1 Comparaison de moyennes

Test de Student, dit test 't'

Le test de Student est effectué par la fonction `t.test`. Cette fonction permet en fait de faire plusieurs tests différents, selon les paramètres spécifiés :

– Comparer une moyenne d'échantillon à une moyenne théorique : on spécifie l'échantillon par le paramètre **`x`** et la moyenne théorique par le paramètre **`mu`** :

```
1 t.test(rnorm(10,mean=1),mu=1)
```

```
        One Sample t-test

data:  rnorm(10, mean = 1)
t = -0.0335, df = 9, p-value = 0.974
alternative hypothesis: true mean is not equal to 1
95 percent confidence interval:
 0.1586634 1.8167746
sample estimates:
mean of x
 0.987719
```

La fonction renvoie plusieurs informations : le type de test (one-sample t-test), les données, la valeur de la statistique (t), le nombre de degrés de liberté (df), le risque de première espèce (= p-value), l'hypothèse alternative, la moyenne estimée et son intervalle de confiance. Par défaut ce dernier est donné à 5%, mais on peut changer ce comportement en spécifiant le paramètre **conf.level** qui par défaut prend la valeur 0.95.

– Comparer deux échantillons entre eux passés par les paramètres **x** et **y**. Le paramètre **mu** désigne alors la différence de moyennes théoriques entre les deux échantillons (0 par défaut = égalité des moyennes).

```
t.test(rnorm(10,mean=5), rnorm(10,mean=1))
```

```
        Welch Two Sample t-test

...
t = 11.4811, df = 17.619, p-value = 1.311e-09
...
```

Là encore, remarquer la loquacité de la sortie de la fonction...

```
t.test(rnorm(10,mean=5), rnorm(10,mean=1), mu=4)
```

```
        Welch Two Sample t-test

...
t = 0.7913, df = 17.558,
 p-value = 0.4393
...
```

Il est à noter que par défaut, les échantillons peuvent être issus de distributions ayant des variances différentes (correction de Welch)... Pour effectuer un 'vrai' test de Student, il faut spécifier le paramètre **var.equal**=TRUE (FALSE par défaut).

Il reste deux paramètres à décrire : le premier, **alternative** permet de spécifier si on veut un test bilatéral ("two.sided") ou unilatéral ("less" ou "greater" suivant le sens). Le second paramètre permet d'effectuer un test modifié pour des données appariées (**paired**=TRUE, FALSE par défaut).

Le test de Mann-Whitney Wilcoxon

Le test de Mann-Whitney Wilcoxon (non-paramétrique) s'effectue par la fonction `wilcox.test`. Cette fonction accepte les mêmes arguments que `t.test`. Deux arguments supplémentaires sont également disponibles :

– **exact** p-value exacte calculée, sinon approximation normale. Par défaut, l'approximation est effectuée si la taille des échantillons est > 50.
– **correct** [Correction de continuité si approximation normale.]

2.2.2 Comparaison de 2 variances

Test de Fisher, dit test 'F'

La fonction `var.test` permet de comparer les variances de deux échantillons (Test de Fischer). Les paramètres de cette fonction sont :

– **x** et **y** les échantillons à comparer (vecteurs) ;
– **alternative** = "two.sided" (défaut), "less" or "greater" ;
– **conf.level** le seuil de l'intervalle de confiance du rapport de variances observé (0.95 par défaut) ;
– **ratio** le rapport théorique des variances (1 par défaut = égalité des variances).

Cette fonction est donc similaire à `t.test`, de même que sa sortie :

```
var.test(rnorm(10,sd=1),rnorm(10,sd=2))
```

```
        F test to compare two variances

data:  rnorm(10, sd = 1) and rnorm(10, sd = 2)
F = 0.2839, num df = 9, denom df = 9, p-value = 0.07462
```

```
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.07052118 1.14305165
sample estimates:
ratio of variances
         0.2839179
```

Comparaison de 2 variances : tests non-paramétriques

Il existe deux versions non-paramétriques du test F, le test de Ansari-Bradley et le test de Mood. Ils sont effectués sous R en utilisant les fonctions `ansari.test` et `mood.test`, qui fonctionnent de manière similaire à `var.test`

Note : Les fonctions `t.test`, `wilcox.test`, `var.test`, `ansari.test` et `mood.test` prennent en argument deux échantillons sous forme de deux vecteurs. Il est également possible de les utiliser en leur passant un vecteur de données et un facteur à deux modalités. Les fonctions compareront alors automatiquement les données de la modalité 1 avec les données de la modalité 2, exemple : comparez

```
1 v1 <- rnorm(10)
2 v2 <- rnorm(12)
3 t.test(v1,v2)
```

et

```
4 v <- c(v1, v2)
5 f <- as.factor(c(rep(1,10), rep(2,12)))
6 t.test(v~f)
```

Comparaison de k variances

Il existe des tests permettant de comparer la variance de plus de 2 échantillons. Ces tests sont notamment utilisés pour tester l'hypothèse d'homoscédasticité (= égalité des variances) lors d'une ANOVA. Il existe 2 tests, le test de Bartlett (paramétrique) et le test de Fligner-Killeen (non-paramétrique). Ces deux tests fonctionnent de la même manière : ils prennent en argument une formule du type $vecteur \sim facteur$, comme les tests précédents, sauf que cette fois-ci le facteur peut avoir plus de 2 modalités. Il est également possible d'appeler les fonctions en passant le vecteur et le facteur séparés par une virgule. A la manière des tests précédents, on aussi passer directement les échantillons, mais sous forme d'une liste[1]. Nous aurons l'occasion de revenir sur ces tests lorsque nous verrons l'ANOVA.

Exercice 6 *Reprendre les vecteurs u et v de l'exercice 5 et effectuer un test 't' et un test 'F'. Comparer les statistiques et les p-values obtenues.*

2.2.3 Test de normalité

Il existe plusieurs manières de tester la normalité d'une variable. Un des tests les plus utilisés est le test de Shapiro-Wilk. Son utilisation est très simple puisqu'il suffit d'utiliser la fonction `shapiro.test` sur le vecteur contenant les valeurs à tester :

```
1 shapiro.test(rnorm(100))
```

Attention : l'hypothèse nulle de ce test est que les données suivent une loi normale ! Une p-value significative implique donc que les données ne sont **pas normales**. Très généralement, on souhaite donc que le test ne soit pas significatif !

Exercice 7 *Survie et taille de bec.*

Utiliser le jeu de données "beak_size.cvs" du chapitre 1. Convertissez la variable "survival" en facteur. En cas de difficulté, se reporter au chapitre précédent.

En utilisant les tests vus précédemment, essayez de répondre à la question : La taille du bec a-t-elle une influence sur la survie des Pinsons ?

[1] Les listes constituent un nouveau type d'objet que nous verrons plus tard.

2.2.4 Comparaison de fréquences (=proportions)

La fonction `prop.test` permet de comparer des fréquences observées à des fréquences théoriques ou entre elles. Son fonctionnement est là encore très similaire aux fonctions vues précédemment. On peut passer en arguments au choix :

- **x** et **n** les effectifs des succès et les effectifs totaux. L'hypothèse nulle testée est alors que toutes les proportions $(\frac{x}{n})$ sont égales.
- **x**, **n** et **p**, où 'p' est le vecteur des proportions attendues. L'hypothèse nulle testée est que toutes les proportions observées sont égales aux proportions théoriques.

2.2.5 χ^2 sur table de contingence

La fonction `chisq.test` effectue un test de χ^2 sur table de contingence. On peut passer au choix en arguments :

- **x** et **p**, des vecteurs de probabilités (χ^2 d'ajustement, identique à `prop.test` 2)
- **x** et **y**, des vecteurs d'effectifs, (table de contingence à 2 dimensions)
- **x** où 'x' est une matrice d'effectifs (table de contingence de taille n)

Quelques fonctions utiles... La fonction `cut` permet de découper un vecteur et de le transformer en facteur, en faisant des catégories. Cette fonction accepte plusieurs arguments, comparer :

```
x<-1:10
cut(x,2)
cut(x,2,labels=c("inf","sup"))
cut(x,4)
cut(x,4,labels=1:4)
cut(x,breaks=c(0,3,7,10))
cut(x,breaks=c(0,3,7,10), labels=c("A","B","C"))
```

On peut bien sûr stocker les résultats dans une variable :

```
f<-cut(x,breaks=c(0,3,7,10), labels=c("A","B","C"))
```

Il est possible de représenter graphiquement un facteur. Que fait la commande suivante ?

```
plot(f)
```

La fonction `table` permet de construire une table de contingence. Elle prend en argument 1 ou plusieurs facteurs et construit le tableau croisé des effectifs :

```
x<-rnorm(100, mean=1)
y<-rnorm(100, mean=-2, sd=2)
fx<-cut(x,4)
fy<-cut(y,4)
t<-table(fx,fy)
t
```

```
                fy
fx               (-8.75,-5.55] (-5.55,-2.35] (-2.35,0.853] (0.853,4.05]
  (-1.99,-0.539]             2             0             4             1
  (-0.539,0.917]             1            19            21             4
  (0.917,2.37]               2            17            20             3
  (2.37,3.83]                0             3             1             2
```

t est un objet de type *table*, qui ressemble beaucoup à une matrice. Il existe une représentation particulière pour ces objets :

```
plot(t)
```

On dispose également de représentations 3D.... Voici par exemple une manière de représenter graphiquement une distribution binormale :

```
x<-rnorm(10000, mean=1)
y<-rnorm(10000, mean=-2, sd=2)
fx<-cut(x,15)
```

```
fy<-cut(y,15)
t<-table(fx,fy)
plot(t)
image(t)
contour(t)
persp(t, theta=30, phi=30, col=topo.colors(15))
```

La figure 2.1 représente les résultats de ces commandes.

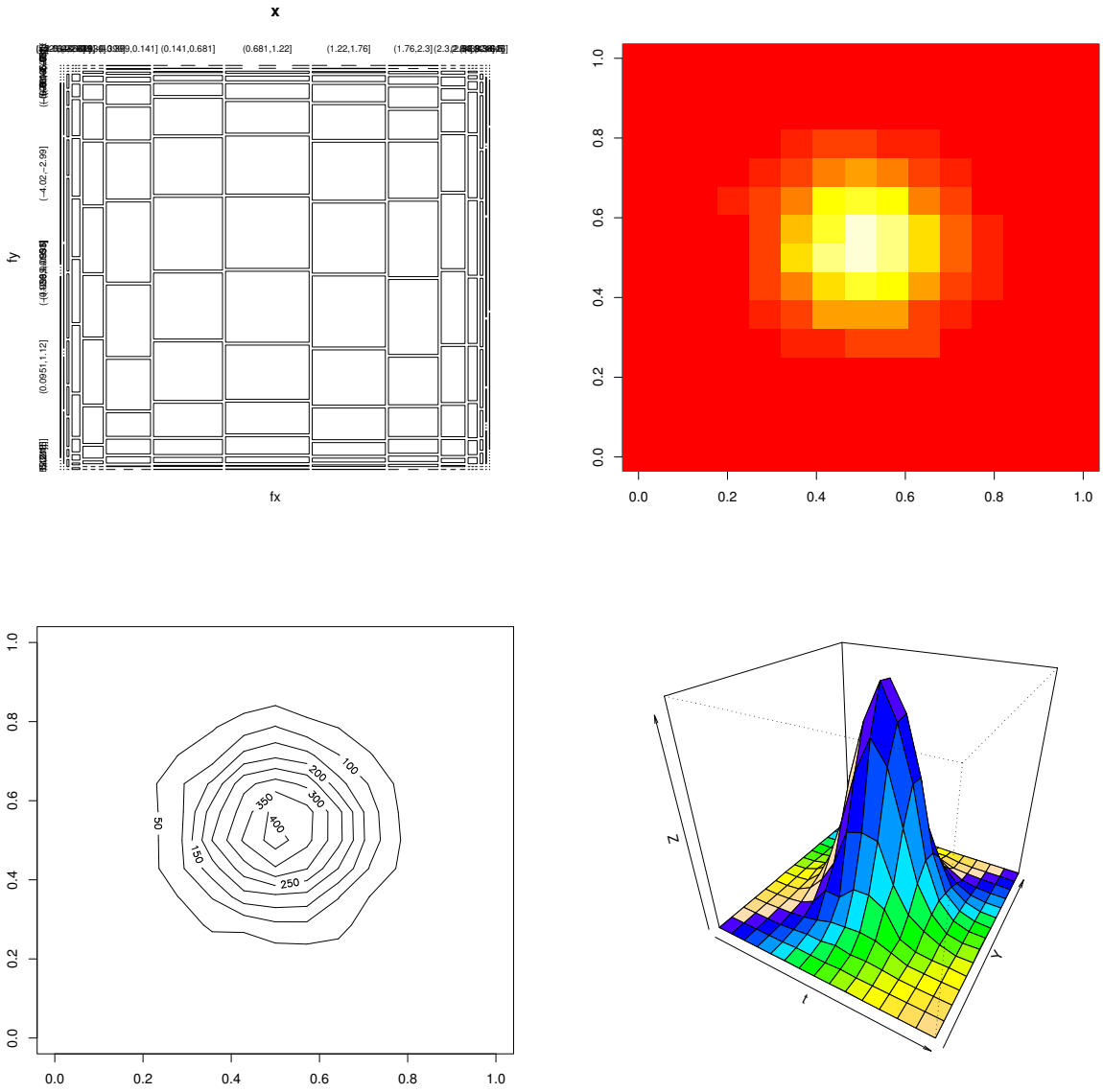


FIG. 2.1 – Exemples de représentations 3D.

Exercice 8 *Comparaison des distributions de tailles de bec. Comparez les distributions des tailles de bec des pinsons qui survivent et ceux qui meurent. Sont-elles identiques ?*

2.2.6 Comparaison de distributions

Le dernier test "classique" que nous allons voir est le test de Kolmogorov-Smirnov. Il est effectué par la fonction `ks.test`. Cette fonction prend deux arguments (+ les traditionnels **`alternative`** et **`exact`**) :

1. **`x`** le vecteur à tester
2. **`y`** un deuxième vecteur à comparer avec le premier, ou bien une chaîne de caractères contenant le nom d'une fonction de distribution. Dans ce cas on peut passer en arguments supplémentaires à `ks.test` les arguments de la fonction.

Important : pour fonctionner correctement, les vecteurs ne doivent pas contenir d'éléments répétés. On pourra donc utiliser la fonction `unique` si besoin. Il est possible d'utiliser ce test pour tester la normalité d'une fonction :

```
x<-runif(30)
ks.test(x, "pnorm", mean=2, sd=sqrt(2))
```

```
 One-sample Kolmogorov-Smirnov test

data:  x
D = 0.7629, p-value = 1.332e-15
alternative hypothesis: two.sided
```

Exercice 9 *Tests de normalité.*

Utiliser le jeu de données "beak_size.csv". Nous allons travailler sur la variable beak_size, afin de tester sa normalité.

- *Effectuez un test de Shapiro sur cette variable. Conclusion ?*
- *Découpez cette variable en une dizaine de classes égales (on pourra prendre `5:14` pour simplifier). Effectuer la représentation graphique correspondante.*
- *Calculer les effectifs théoriques de ces classes, si la variable suivait une loi normale de même moyenne et de même variance. Comparer les effectifs théoriques aux effectifs observés. Conclusion ?*
- *Ajouter au graphe précédent la courbe théorique. On pourra pour cela regarder la documentation de la fonction `lines`. Il est à noter que la fonction `plot` appliquée à un facteur renvoie les coordonnées des abscisses, que l'on peut récupérer dans une variable.*

Nom	Densité	Paramètres	Fonctions

Normale	$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	μ, σ, la moyenne et l'écart-type	`dnorm(x,mean=0,sd=1)`
			`pnorm(q,mean=0,sd=1)`
			`qnorm(p,mean=0,sd=1)`
			`rnorm(n,mean=0,sd=1)`

Uniforme	$f(x) = \begin{cases} 0, \ \forall x < a \\ \frac{1}{b-a}, \ \forall x \in [a,b] \\ 1, \ \forall x > b \end{cases}$	a, b les extrema	`dunif(x,min=0,max=1)`
			`punif(q,min=0,max=1)`
			`qunif(p,min=0,max=1)`
			`runif(n,min=0,max=1)`

Exponentielle	$f(x) = \lambda e^{-\lambda x}$	λ la vitesse	`dexp(x,rate=1)`
			`pexp(q,rate=1)`
			`qexp(p,rate=1)`
			`rexp(n,rate=1)`

Gamma	$f(x) = \frac{1}{s^a \Gamma(a)} x^{a-1} e^{-x/s}$	a, s la forme et l'échelle (=1/vitesse)	`dgamma(x,shape,rate=1)`
			`pgamma(q,shape,rate=1)`
			`qgamma(p,shape,rate=1)`
			`rgamma(n,shape,rate=1)`

Logistique	$f(x) = \frac{1}{s} \frac{e^{\frac{x-m}{s}}}{\left(1+e^{\frac{x-m}{s}}\right)^2}$	m, s la position et l'échelle	`dlogis(x,location=0,scale=1)`
			`plogis(q,location=0,scale=1)`
			`qlogis(p,location=0,scale=1)`
			`rlogis(n,location=0,scale=1)`

χ^2	$f_n(x) = \begin{cases} \frac{x^{n/2-1} e^{-x/2}}{2^{n/2}\Gamma(n/2)}, \ \forall x > 0 \\ 0, \ \forall x \leq 0 \end{cases}$	n le nombre de degrés de liberté	`dchisq(x)`
			`pchisq(q)`
			`qchisq(p)`
			`rchisq(n)`

t de Student	$f_n(x) = \frac{\Gamma\left(\frac{n+1}{2}\right)}{\sqrt{n\pi}\Gamma\left(\frac{n}{2}\right)} \left(1 + x^{\frac{2}{n}}\right)^{-\frac{n+1}{2}}$	n le nombre de degrés de liberté	`dt(x,df)`
			`pt(q,df)`
			`qt(p,df)`
			`rt(n,df)`

F de Fisher	$f_{n_1,n_2} = \frac{\Gamma\left(\frac{n_1+n_2}{2}\right)}{\Gamma\left(\frac{n_1}{2}\right)\Gamma\left(\frac{n_2}{2}\right)} \left(\frac{n_1}{n_2}\right)^{n_1/2} \times x^{n_1/2-1} \left(1 + \frac{n_1}{n_2} x\right)^{-\frac{n_1+n_2}{2}}, \ \forall x > 0$	n_1, n_2 nombres de degrés de liberté	`df(x,df1,df2)`
			`pf(q,df1,df2)`
			`qf(p,df1,df2)`
			`rf(n,df1,df2)`

TAB. 2.1 – Quelques distributions continues. On note Γ la fonction factorielle généralisée : $\Gamma(x) \int_0^\infty t^{x-1} e^{-} t dt$

Chapitre 3

Corrélation, régression simple et analyse de variance à un facteur

Ce TP aborde quelques tests de corrélation, ainsi que les régressions simples (linéaire et autres) et l'analyse de variance (paramétrique et non paramétrique). Dans ce chapitre, nous allons principalement continuer à travailler sur les Pinsons de Darwin. On utilise un nouveau jeu de données, nommé "beak_size_heritability.csv". Pour chaque individu, on a mesuré la taille de son bec, ainsi que celles de ses parents. Voici les variables disponibles :

- *id* L'identifiant de l'individu.
- *beak_size* Taille du bec.
- *survival* Survie (voir jeu de données "beak_size").
- *mother_id* Identifiant de la mère.
- *father_id* Identifiant du père.
- *mother_size* Taille du bec de la mère.
- *father_size* Taille du bec du père.

3.1 Analyse de corrélation

La fonction `cor` calcule le coefficient de corrélation de deux vecteurs. Deux arguments peuvent être spécifiés :

- **`method`** prend pour valeur une chaîne de caractères précisant le type de corrélation à calculer. Par défaut sa valeur est "Pearson", mais on peut aussi calculer un coefficient de corrélation de Spearman ou de Kendall.
- **`use`** permet de préciser comment traiter les valeurs manquantes :
  - "all.obs" (défaut) utilise toutes les valeurs et conduit à une erreur si des données manquent,
  - "complete.obs" utilise seulement les données complètes dans chacun des vecteurs,
  - "pairwise.complete.obs" utilise seulement les paires pour lesquelles les 2 données sont disponibles.

La fonction `cor.test` permet de tester si un coefficient de corrélation est significativement différent de 0. Les arguments sont les mêmes que `cor`, avec en plus les arguments traditionnels des tests classiques (**`alternative`**, **`exact`**, **`conf.level`**, cf. chapitre précédent.).

Exercice 10 *Dimorphisme sexuel et homogamie chez les Pinsons de Darwin.*

- *Récupérez le fichier "beak_size_heritability.csv" sur le site de l'université et copiez le dans le répertoire* `stats/data`.
- *Importez ce jeu de données dans R, en le stockant dans une variable nommée becs.h.*
- *Question liminaire : y a-t-il un dimorphisme sexuel pour la taille du bec ?*
- *Testez s'il y a homogamie pour la taille du bec*[1].

3.2 Régression linéaire simple

R ne dispose pas de fonction de régression linéaire. Il dispose cependant d'une panoplie d'outils permettant de travailler sur le *modèle linéaire*, la régression linéaire simple en étant un cas particulier. Nous introduisons ici les outils de modèle linéaire sous R, qui seront appliqués au cas de la régression simple. Ils seront approfondis dans le chapitre suivant.

[1] On rappelle que l'homogamie désigne le fait que des individus ne s'apparient pas pour la reproduction de manière aléatoire, mais par ressemblance selon un ou plusieurs caractères.

3.2.1 Les objets `lm`

Les modèles linéaires sont des objets de classe `lm` que l'on crée par la fonction... `lm`(!) Cette fonction prend en argument une *formule*, puis éventuellement quelques options.

Le point important est la spécification du modèle par une formule. Pour le cas de la régression linéaire simple, c'est assez simple : la régression de y par x s'écrit tout simplement `y ~ x`, ou y et x sont des variables, vecteurs pour la régression, mais nous verrons plus tard que x peut aussi être un facteur. Il est à noter que par défaut, les modèles incluent toujours l'intersection avec l'origine. Si on veut effectuer une régression de type $y = a.x$ et non $y = a.x + b$, il faut explicitement enlever l'intersection dans la formule : `y ~ x - 1`.

Les variables à utiliser sont souvent dans un objet dataframe, aussi plutôt que d'écrire

```
lm(donnees[,"y"] ~ donnees[,"x"])
```

ce qui peut s'avérer fastidieux si on a beaucoup de variables, on écrira

```
lm(y ~ x, data=donnees)
```

La fonction `lm` peut prendre plusieurs arguments, notamment :

- **`weights`** un vecteur de poids pour l'ajustement selon les moindres carrés,
- **`na.action`** une fonction pour traiter les données manquantes (NA = Not Available). Attention, le fonctionnement est ici quelque peu différent de la fonction `cor`, puisqu'on passe une fonction et non une chaîne de caractères. Plusieurs fonctions sont prédéfinies et doivent être utilisées :
  - "`na.fail`" (défaut, équivalent à "all.obs") renvoie une erreur si une donnée manquante est rencontrée,
  - "`na.omit`" enlève toutes les observations ayant au moins une donnée manquante (équivalent à "pairwise.complete.obs" dans le cas de la régression simple),
  - "`na.exclude`" est similaire à la précédente, mais marque "NA" dans les sorties de fonctions (voir section suivante).
  - "`na.pass`" ne fait rien ! A ne pas utiliser donc...
- **`subset`** permet de travailler sur un sous-jeu de données : prend en argument un vecteur (ex : `c(1:10)` travaille seulement sur les 10 premières lignes, `c(-2,-4)` enlève les lignes 2 et 4, etc.)

3.2.2 Travail sur les modèles

Plusieurs fonctions prennent en argument un objet de classe `lm` :

- `coefficients` (ou simplement `coef`) permet de récupérer les coefficients de la régression, pente de la droite et ordonnée à l'origine.
- `fitted.values` (ou simplement `fitted`) récupère les valeurs estimées (les $\hat{y_i}$, soient les points sur la droite pour chaque x_i).
- `residuals` (ou simplement `resid`) renvoie les résidus (les $y_i - \hat{y_i}$, les distances de chaque point par rapport à la droite, selon les moindre carrés de type I).
- `summary` imprime un résumé du modèle, avec coefficients et tests de significativité.
- `print` imprime une courte description du modèle.
- `deviance` calcule la somme des carrés résiduelle.
- `predict` (prend en argument un deuxième dataframe avec les mêmes variables explicatives que celui utilisé pour ajuster le modèle) permet de prédire des valeurs à partir du modèle.
- `AIC` calcule le critère informatif d'Akaike[2]. Peut prendre plusieurs modèles en argument.
- `abline` ajoute la droite de régression à un graphe.

Étude sur un exemple :

Nous traiterons la question de l'homogamie chez les Pinsons (exercice 10), mais en effectuant une régression linéaire et non un test de corrélation.

Nous construisons un modèle, que l'on stocke dans une variable nommée *lm*, puis nous en affichons le contenu :

```
1 model <- lm(father_size ~ mother_size, data=becs.h)
2 model
```

[2] Akaike's Information Criterion, égal à $-2.ln(L) + 2K$, où L est la vraisemblance du modèle et K le nombre de paramètres. Voir le chapitre 4, section "sélection de modèles" pour l'utilisation de ce critère.

```
> model

Call:
lm(formula = father_size ~ mother_size, data = becs.h)

Coefficients:
(Intercept)  mother_size
     3.5371       0.5958
```

L'affichage fournit les coefficients du modèle. La droite de régression a pour équation $father\_size = 0.5958 \times mother\_size + 3.5371$. On peut représenter la droite en faisant :

```
3 plot(father_size ~ mother_size, data=becs.h, col="blue")
4 abline(model, col="red")
```

La figure 3.1 montre le résultat produit.

Notez l'utilisation de la formule dans `plot` : celle-ci est strictement équivalente à

```
plot(becs.h[,"mother_size"], becs.h[,"father_size"], col="blue").
```

Notez bien l'inversion des variables : dans un cas on déclare y en fonction de x, dans l'autre x puis y.

La fonction `termplot` est similaire à `abline`, mais ajoute notamment les intervalles de confiance, ainsi que des histogrammes de chaque variable. On se reportera à la documentation pour une description détaillée, un exemple est montré sur la figure 3.1(d).

Validation du modèle.

Nous allons vérifier *a posteriori* les hypothèses du modèle.

Normalité des résidus On peut récupérer les résidus grâce à la fonction `residuals`. On en fait ensuite un histogramme et on teste la normalité :

```
5 model.res <- residuals(model)
6 hist(model.res)
7 shapiro.test(model.res)
```

```
> shapiro.test(model.res)

        Shapiro-Wilk normality test

data:  model.res
W = 0.972, p-value = 0.7566
```

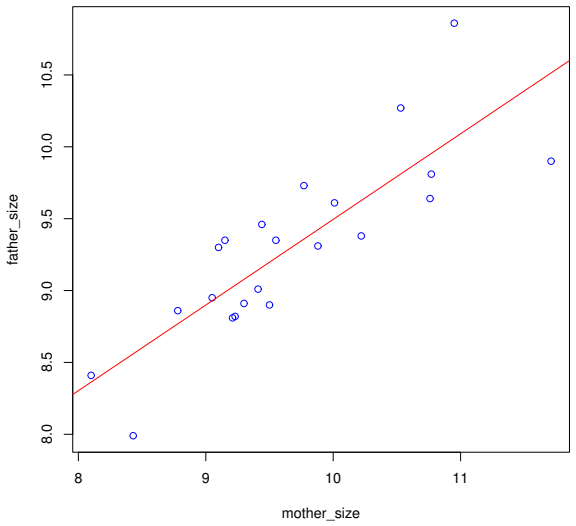
Les résidus ne sont pas significativement différents d'une loi normale.

La fonction générique `plot` appliquée à un objet de type *lm* trace 4 graphes, que l'on fait défiler en appuyant sur "entrée". On peut les afficher simultanément en procédant comme suit :

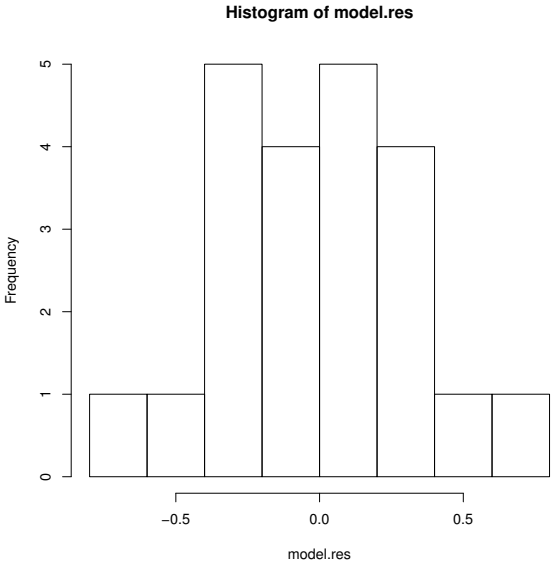
```
8 par(mfrow = c(2, 2), oma = c(0, 0, 2, 0))
9 plot(model)
```

Le résultat est montré dans la figure 3.1. On a :

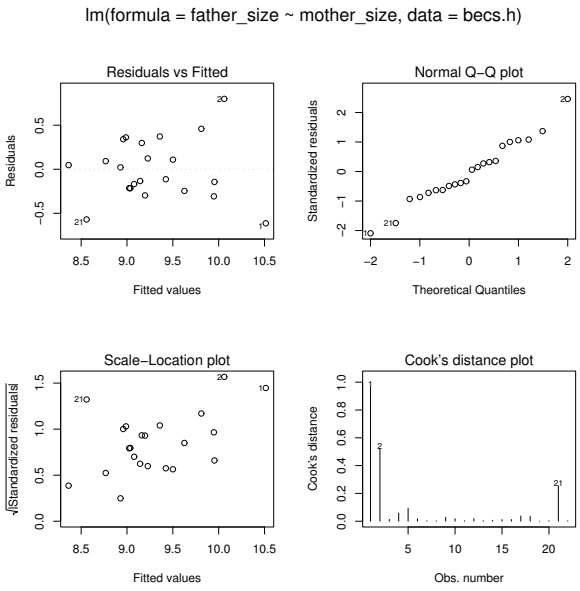
- En haut à gauche : graphe des résidus.
- En haut à droite : droite de Henry (Tracé Quantile-Quantile).
- En bas à gauche : Scale-Location plot (ou Spread-Location plot), graphe de la racine des résidus (moins biaisé que résidus simples).
- En bas à droite : distance de Cook (une longue distance indique un point mal modélisé et susceptible de "tirer" la corrélation (outlier)).



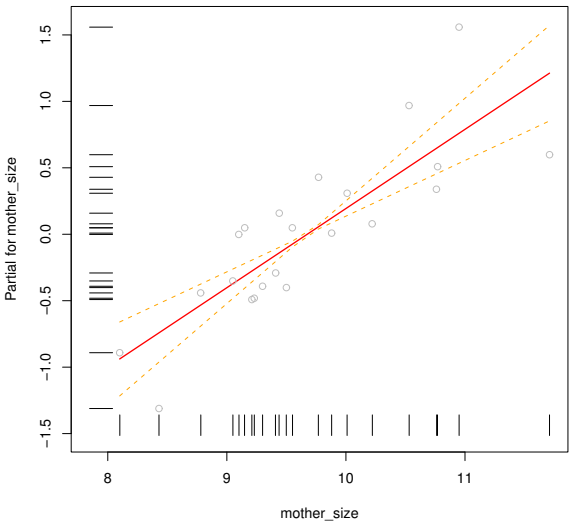
(a) Utilisation de la fonction `abline`



(b) Histogramme des résidus



(c) Résultat de la fonction `plot`



(d) Résultat de la fonction `termplot`

FIG. 3.1 – Quelques graphes obtenus sur un modèle.

Homoscédasticité Le paquet `lmtest` contient un ensemble de tests dédiés aux modèles linéaires. Plusieurs tests sont disponibles, on pourra utiliser par exemple le test de Harrison-McCabe, qui a pour hyomthèse nulle que les résidus suivent des lois normales de même variance :

```
10 hmctest(model)
```

```
> hmctest(model)

        Harrison-McCabe test

data:  model
HMC = 0.6731, p-value = 0.863
```

NB : il existe également le test de Breusch-Pagan, fonction `bptest`.

Indépendance des résidus Toujours dans le paquet `lmtest`, on trouvera le test de Durbin-Watson pour tester l'indépendance des résidus (hypothèse nulle : les résidus sont indépendants) :

```
11 dwtest(model)
```

```
> dwtest(model)

        Durbin-Watson test

data:  model
DW = 2.6265, p-value = 0.903
alternative hypothesis: true autocorrelation is greater than 0
```

Tests de significativité

Après avoir vérifié les hypothèses du modèle linéaire, on peut tester la significativité des coefficients de la régression, on utilise la fonction `summary` :

```
12 summary(model)
```

qui produit les résultats suivants :

```
> summary(model)

Call:
lm(formula = father_size ~ mother_size, data = becs.h)

Residuals:
     Min       1Q   Median       3Q      Max
-0.61379 -0.21576 -0.04625  0.25476  0.79901

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.53715    0.85782   4.123 0.000527 ***
mother_size  0.59578    0.08832   6.745 1.46e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3521 on 20 degrees of freedom
Multiple R-Squared: 0.6947,     Adjusted R-squared: 0.6794
F-statistic:  45.5 on 1 and 20 DF,  p-value: 1.461e-06
```

Dans cette sortie, on trouve plusieurs informations, plus ou moins utiles :

- Un rappel de la formule du modèle (Call),
- Des informations sur la distribution des résidus du modèle (Residuals),
- Les coefficients estimés, avec leurs intervalles de confiance, la valeur de la statistique et la p-value du test (Coefficients),
- Le nombre de degrés de liberté, le R^2.

Dans notre cas on a un effet très significatif de la taille du bec de la mère sur la taille de bec du père.

Question : Que reprocheriez vous à l'analyse effectuée ? Comment l'améliorer ?

Exercice 11 *Héritabilité de la taille du bec.*

Que pouvez-vous dire sur l'héritabilité de la taille du bec, du point de vue de la mère, puis du point de vue du père ? Pour les généticiens, on rappellera que $h = \sqrt{2b}$, où b est la pente de la droite de régression. Vérifiez bien toutes les hypothèses des modèles.

3.3 Analyse de variance (ANOVA) à un facteur

Pour terminer ce chapitre, nous allons aborder l'analyse de variance à un facteur. Comme pour la régression linéaire, il n'y a pas à proprement parler de fonction spécifique dans R, l'ANOVA étant un cas particulier du modèle linéaire.

3.3.1 1 facteur, 2 modalités

Nous allons utiliser le jeu de données "beak_size.csv" :

```
becs<-read.table(file="data/beak_size.csv", header=T, sep="\t", dec=",")
becs[,"survival"]<-as.factor(becs[,"survival"])
levels(becs[,"survival"])<-c("mort", "vivant")
```

Nous allons tester s'il y a une différence de taille de bec entre les pinsons morts et les pinsons vivants :

```
model.anova<-lm(beak_size~survival, data=becs)
model.anova
```

Le modèle se fait exactement de la même manière que pour la régression. L'unique différence est que ici la variable explicative (*survival*) est discrète et non continue, et donc codée par un facteur et non un vecteur. La sortie écran du modèle linéaire n'est pas très explicite, il faut utiliser la fonction `anova` :

```
anova(model.anova)
```

```
Analysis of Variance Table

Response: beak_size
           Df Sum Sq Mean Sq F value    Pr(>F)
survival    1  19.47   19.47  17.715 2.878e-05 ***
Residuals 749 823.25    1.10
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

La sortie est cette fois la table d'ANOVA. On y voit un effet significatif de la variable *survival*.

Exercice 12 *Vérifiez les hypothèses de cette analyse de variance.*

3.3.2 1 facteur, plusieurs modalités

On considère le jeu de données "chickwts" fourni avec R et qui contient des masses de poulets en fonction de leur alimentation (cf. figure 3.2) :

```
plot(weight~feed, data=chickwts, col="lightgray")
```

On veut tester s'il y a un effet du type de nourriture. On construit pour cela le modèle suivant :

```
model.chickwts<-lm(weight~feed, data=chickwts)
```

Validation du modèle

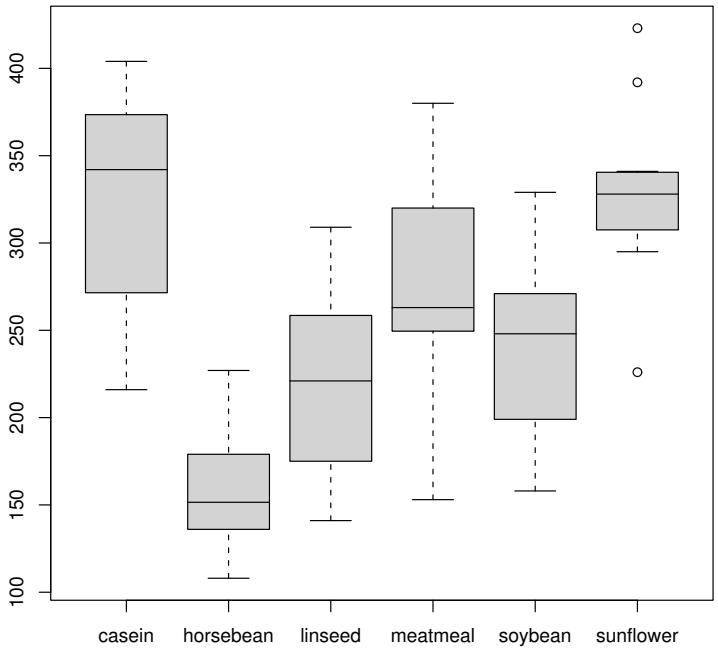
Test de la normalité des résidus : On effectue un test de Shapiro sur les résidus du modèle :

```
shapiro.test(residuals(model.chickwts))
```

```
        Shapiro-Wilk normality test

data:  residuals(model.chickwts)
W = 0.9862, p-value = 0.6272
```

On ne rejette donc pas l'hypothèse de normalité.



Nourriture :	Traduction :
casein	caséine
horsebean	fève
linseed	graine de lin
meatmeal	farine animale
soybean	soja
sunflower	tournesol

FIG. 3.2 – Boxplot du poids des poulets par type de nourriture.

Test de l'homoscédasticité : Puisque l'hypothèse de normalité n'est pas rejetée, on peut effectuer un test de Bartlett pour tester l'homogénéité des variances :

```
4 bartlett.test(weight~feed, data=chickwts)
```

```
        Bartlett test for homogeneity of variances

data:  weight by feed
Bartlett's K-squared = 3.2597, df = 5, p-value = 0.66
```

L'hypothèse nulle n'est pas rejetée, on ne rejette donc pas l'hypothèse d'homoscédasticité.

Interprétation

```
1 anova(model.chickwts)
```

```
Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value    Pr(>F)
feed       5 231129   46226  15.365 5.936e-10 ***
Residuals 65 195556    3009
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

On voit ainsi qu'on a un effet très marqué du type de nourriture. Plus précisément, cela implique qu'au moins 1 des types de nourriture est significativement différent des autres. Pour affiner les conclusions, on voudrait savoir quels sont les types de nourriture qui diffèrent des autres. On peut pour cela effectuer tous les tests 2 à 2, mais cette approche pose cependant un problème, celui dit des *tests multiples*. Le problème est que si on effectue 15 tests au seuil de 5%, on a $1 - \left(\frac{95}{100}\right)^{15} = 54\%$ de chance d'avoir rejeté au moins une fois l'hypothèse nulle à tord.

Pour remédier à ce problème, on peut utiliser le test de Tukey. On utilise pour cela la fonction `TukeyHSD` (Tukey's **H**onest **S**ignificant **D**ifference). Cette fonction ne fonctionne pas avec des objets *lm*, mais avec des objets *aov*, que l'on obtient de la manière suivante :

```
aov.chickwts<-aov(weight~feed, data=chickwts)
summary(aov.chickwts)
```

```
            Df Sum Sq Mean Sq F value    Pr(>F)
feed         5 231129   46226  15.365 5.936e-10 ***
Residuals   65 195556    3009
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

`aov` (**a**nalyse **o**f **v**ariance) est très similaire à `lm`, mais possède des arguments supplémentaires et surtout une sortie optimisée pour l'analyse de variance. Les résultats sont en tout point identiques à ceux de la fonction `lm`, ils sont seulement présentés différemment.

Utilisation de `TukeyHSD` :

```
hsd.chickwts<-TukeyHSD(aov.chickwts)
hsd.chickwts
```

Il existe aussi une fonction `plot` associée :

```
plot(hsd.chickwts)
```

Les résultats sont montrés sur la figure 3.3. Tous les intervalles de confiance qui ne recoupent pas 0 révèlent des différences significatives d'effets.

On peut récupérer les effets (grâce à la fonction `model.tables`) de chaque type de nourriture et les ordonner :

```
effects.chickwts<-model.tables(aov.chickwts)
sort(effects.chickwts[["tables"]][["feed"]])
```

```
feed
 horsebean    linseed    soybean   meatmeal     casein  sunflower
-101.10986  -42.55986  -14.88129   15.59923   62.27347   67.60681
```

Les résultats du test de Tukey montre qu'on peut classer les aliments en trois catégories :

Horsebean			c
Linseed		b	c
Soybean		b	
Meatmeal	a	b	
Casein	a		
Sunflower	a		

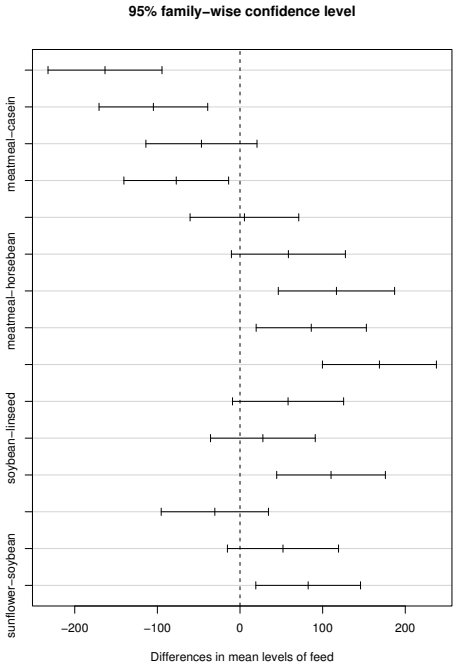
3.3.3 L'ANOVA non-paramétrique : le test de Kruskal-Wallis

La fonction `kruskal.test` permet d'effectuer le test de somme des rangs de Kruskall-Wallis. Les arguments de la fonction sont les mêmes que pour `lm` et `aov` :

```
kruskal.test(weight~feed, data=chickwts)
```

```
        Kruskal-Wallis rank sum test

data:  weight by feed
Kruskal-Wallis chi-squared = 37.3427, df = 5, p-value = 5.113e-07
```




```
> hsd.chickwts
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = weight ~ feed, data = chickwts)

$feed
                             diff         lwr       upr
horsebean-casein     -163.383333 -232.346876 -94.41979 (*)
linseed-casein       -104.833333 -170.587491 -39.07918 (*)
meatmeal-casein       -46.674242 -113.906207  20.55772
soybean-casein        -77.154762 -140.517054 -13.79247 (*)
sunflower-casein        5.333333  -60.420825  71.08749
linseed-horsebean      58.550000  -10.413543 127.51354
meatmeal-horsebean    116.709091   46.335105 187.08308 (*)
soybean-horsebean      86.228571   19.541684 152.91546 (*)
sunflower-horsebean   168.716667   99.753124 237.68021 (*)
meatmeal-linseed       58.159091   -9.072873 125.39106
soybean-linseed        27.678571  -35.683721  91.04086
sunflower-linseed     110.166667   44.412509 175.92082 (*)
soybean-meatmeal      -30.480519  -95.375109  34.41407
sunflower-meatmeal     52.007576  -15.224388 119.23954
sunflower-soybean      82.488095   19.125803 145.85039 (*)
```

FIG. 3.3 – Intervalles de confiance de Tukey.

3.4 Régression non-linéaire simple

R permet également d'ajuster n'importe quel modèle à des données, linéaire ou non, selon le critère des moindres carrés I. On utilise pour cela la fonction `nls` (non-linear least squares) au lieu de `lm`.

L'interface de la fonction `nls` est très similaire à la fonction `lm`, mais la formule doit préciser la fonction à ajuster, avec les paramètres adéquats. Pour effectuer une régression de type 'puissance', on écrira par exemple `y ~ a*x^b`, y et x étant les variables, et a et b les paramètres. On précisera de plus la liste des paramètres ainsi que leurs valeurs initiales par le paramètre **`start`**, par exemple `nls(y ~a*x^b, start=list(a=1, b=1))`. Plusieurs fonctions sont déjà définies pour cet usage, voir tableau 3.1. Ces fonctions commencent toutes par 'SS' (SelfStart), ce qui veut dire qu'il n'est pas nécessaire de préciser la liste des paramètres avec l'option **`start`** dans `nls`.

La plupart des fonctions vues précédemment fonctionnent également avec les objets de type `nls`.

Un exemple : cinétique d'une enzyme. Nous allons utiliser un jeu de données fourni avec R : *Puromycin*. Consulter l'aide pour le détail des variables :

```
1 help(Puromycin}
```

Nous allons ajuster un modèle de Michaelis-Menten afin de déterminer la constante cinétique de l'enzyme :

```
2 Pur <- Puromycin[Puromycin[,"state"]=="treated",1:2]
3 plot(Pur)
4 model.nls <- nls(rate ~ SSmicmen(conc, Vm, K), data=Pur)
5 summary(model.nls)
6 hist(residuals(model.nls))
```

```
Formula: rate ~ SSmicmen(conc, Vm, K)

Parameters:
    Estimate Std. Error t value Pr(>|t|)
Vm 2.127e+02  6.947e+00  30.615 3.24e-11 ***
K  6.412e-02  8.281e-03   7.743 1.57e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Fonction :	Fonction Math :	Paramètres :	Fonction R :	Paramètres :
Fonction logistique (Verhulst)	$f(x) = \frac{K}{1+e^{-r\times(x-\gamma)}}$	K = capacité limite, r = taux de croissance et γ = l'abscisse du point d'inflexion	`SSlogis`	**`Asym`** = K = Asymptote, **`xmid`** = γ = position du point d'inflexion et **`scale`** = $\frac{1}{r}$ = échelle
Fonction de croissance de Gompertz	$f(x) = Ke^{-e^{-r(x-\gamma)}}$	Mêmes paramètres que pour la logistique	`SSgompertz`	**`A`** = K = Valeur asymptotique, **`b2`** = $e^{r\gamma}$ = fixe l'ordonnée à l'origine ($b_2 = 1 \Rightarrow f(0) = 0$), **`b3`** = $\frac{1}{e^r}$ = paramètre d'échelle
Fonction de croissance de Weibull	$f(x) = \alpha - (\alpha - \beta)e^{-(\kappa x)^\delta}$	α est la valeur de l'asymptote supérieure, β l'asymptote inférieure, κ un paramètre d'échelle et δ contrôle le point d'inflexion.	`SSweibull`	**`Asym`** = α, **`Drop`** = $\alpha - \beta$, **`pwr`** = δ et **`lrc`** = $\delta \ln(\kappa)$
Cinétique de Michalis-Menten[3]	$f(x) = \frac{V_m \times x}{K+x}$	V_m = Vitesse maximale, K = constante cinétique de l'enzyme	`SSmicmen`	**`Vm`** = V_m, **`K`** = K

TAB. 3.1 – Quelques fonctions prédéfinies.

```
Residual standard error: 10.93 on 10 degrees of freedom

Correlation of Parameter Estimates:
      Vm
K 0.7651
```

L'histogramme des résidus est montré figure 3.4(a).

Il n'existe par contre pas d'équivalent à la fonction `abline` pour le cas des modèles non-linéaires. Pour tracer le courbe théorique, on peut utiliser la fonction `curve` qui permet de tracer un courbe, sachant son équation que l'on passe en argument :

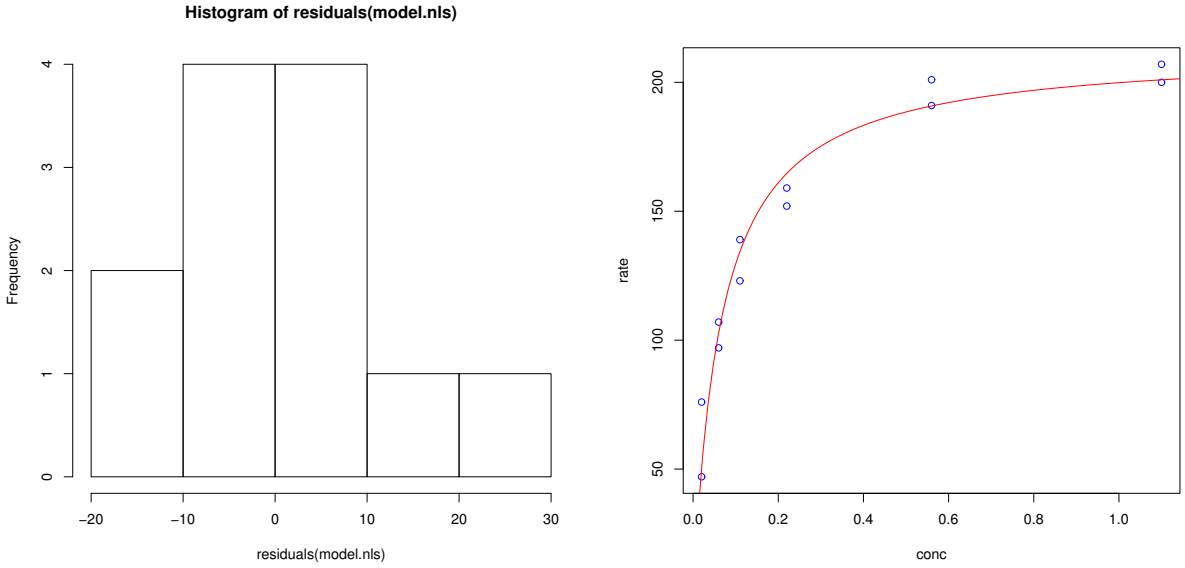
```
7 plot(rate~conc, data=Pur, col="blue")
8 co <- coefficients(model.nls)
9 curve(SSmicmen(x, Vm=co["Vm"], K=co["K"]), add=TRUE, col="red")
```

Le résultat est montré sur la figure 3.4(b).

Exercice 13 *Évolution démographique*

1. *Récupérez le jeu de données* `PopJanv.txt` *et importez les dans R. Ce jeu de données contient la population au 1er janvier de différents pays (source http ://www.ined.fr/bdd/demogr/progr3.php ?lan=F) depuis 1950.*
2. *On se propose d'étudier plusieurs modèles démographiques. Choisissez un pays, représentez graphiquement l'évolution de sa population et ajustez les modèles :*
   – *linéaire,*
   – *Malthusien (exponentiel),*
   – *de Verhultz (logistique),*
   – *de Gompertz.*
   (NB : comptez les années depuis 1950 pour éviter un débordement de capacité, i.e. 1950 devient 0, 1951 devient 1, etc.) Quel modèle décrit le mieux les données ?
3. *Quelle sera la taille de la population de ce pays au 1er janvier 2050 ? Comparez les prédictions des modèles.*

[3] Cinétique utilisée en enzymologie



(a) Histogramme des résidus

(b) Cinétique estimée

FIG. 3.4 – Exemple de régression non-linéaire : cinétique d'une enzyme traitée à la Puromycine.

Chapitre 4

Le modèle linéaire : régression multiple, ANOVA à plusieurs facteurs, analyse de covariance et sélection de modèle

Tous les modèles sont faux, mais certains sont utiles

George Box

Dans ce TP nous allons voir comment effectuer des régressions multiples, des ANOVAs à plusieurs facteurs et des ANCOVAs avec R. Ces techniques sont des cas particuliers de modèles linéaires, et sont traitées de manière très similaire par R. Nous verrons donc tout d'abord comment spécifier un modèle linéaire dans R, puis nous verrons comment faire des régressions, ANOVAs et ANCOVAs.

4.1 Spécifier un modèle dans R

Un modèle est composé de *paramètres* (on parle aussi d'*effets*) et de *variables*. L'*équation* d'un modèle relie les paramètres aux variables. L'équation contient en général 2 parties, la partie droite contient les variables explicatives (ou dépendantes), la partie gauche les variables à expliquer (ou indépendantes). La plupart des logiciels de statistiques (SAS, GLIM, etc.) représentent donc des modèles par des équations, R n'échappant pas à la règle (la syntaxe utilisée à cet égard est quasiment identique entre ces logiciels). Dans R, les variables sont des variables créées par l'utilisateur ou bien des colonnes d'un dataframe, et l'équation est un objet *formule* que nous avons déjà rencontré brièvement (cf chapitre 2 et 3). La table 4.1 donne un aperçu de la syntaxe utilisée.

`A+B`	effet de A + effet de B
`A:B`	interaction entre A et B
`A*B`	`A+B+A:B`
`A/B`	`A+A:B`
`^n`	toutes les interactions d'ordre $\leq n$
`(A+B)^2`	`A*B`
`(A+B+C)^2`	`A+B+C+A:B+B:C+A:C`
`(A+B+C)^3`	`A+B+C+A:B+B:C+A:C+A:B:C`
`A-1`	effet de A sans l'intercept
`(A+B)^2-B`	`A+A:B`

TAB. 4.1 – Opérateurs de formules dans R.

Exercice 14 *Quiz*

1. *Soient quatre variables A, B, C et D. Écrivez le plus succinctement possible les modèles suivant :*
   - *m1 = les effets de chaque variable*
   - *m2 = les effets de chaque variable + toutes les interactions 2 à 2*
   - *m3 = les effets de chaque variable + toutes les interactions possibles*

   - *m4 = tous les effets jusqu'à l'ordre 3 inclus*
2. *A quel modèle correspond* `A + A^2 + A^3 + A^4` *?*
3. *A quel modèle correspond* `A/B/C/D` *?*

4.2 Étude de variables quantitatives : La régression linaire multiple

Elle fonctionne exactement comme la régression linéaire simple ! Il suffit simplement de lui passer la liste des variables explicatives dans la formule.

Exercice 15 *Qualité de l'air New-Yorkais*

1. *Charger le jeu de données "airquality" fourni avec R.*
2. *Estimer les trois régressions suivantes :*
   - *Taux d'ozone en fonction de la température, du vent et de l'indice solaire.*
   - *Idem avec toutes les interactions doubles.*
   - *Idem avec toutes les interactions.*

   Dans chaque cas, effectuer un diagnostic et corriger en conséquence.
3. *Comparer les trois régressions et discuter.*

4.3 Étude de variables qualitatives : L'analyse de variance à plusieurs facteurs

Nous allons étudier un jeu de données contenant des temps d'intoxication au cyanure, en fonction de la température, de la concentration en cyanure, et de de la concentration en oxygène. Récupérez le fichier `Cyanure.csv` et importez-le dans R :

```
1 cyanure<-read.table("data/Cyanure.csv", header=T, sep="\t")
```

Ce fichier contient quatre variables :

1. "*Temp*" : température,
2. "*O2*" : concentration en O_2,
3. "*CN*" : concentration en CN^-,
4. "*Intox.Tps*" : temps d'intoxication.

Dans un premier temps, nous allons considérer chaque variable comme un facteur :

```
2 cyanure[,"Temp"] <- as.factor(cyanure[,"Temp"])
3 cyanure[,"CN"]   <- as.factor(cyanure[,"CN"])
4 cyanure[,"O2"]   <- as.factor(cyanure[,"O2"])
```

La fonction `replications` permet de tester combien d'observations sont disponibles pour chaque facteur. Elle prend en argument une formule (seulement la partie droite) :

```
5 replications(~Temp+CN+O2, cyanure)
```

```
Temp   CN   O2
  15    9   15
```

On a 15 observations pour chacun des 3 niveaux de température et des 3 niveaux de concentration en O_2, 9 observations pour chacune des 5 concentrations en CN^-. Les effectifs pour les interactions doubles :

```
6 replications(~Temp:CN + Temp:O2 + CN:O2, cyanure)
```

```
$"Temp:CN"
    CN
Temp 0.16 0.8 4 20 100
  5     3   3 3  3   3
  15    3   3 3  3   3
  25    3   3 3  3   3
```

```
$"Temp:O2"
    O2
Temp 1.5 3 9
  5    5 5 5
  15   5 5 5
  25   5 5 5

$"CN:O2"
      O2
CN     1.5 3 9
  0.16   3 3 3
  0.8    3 3 3
  4      3 3 3
  20     3 3 3
  100    3 3 3
```

Et les effectifs pour l'interaction triple :

```
replications(~Temp:CN:O2, cyanure)
```

```
$"Temp:CN:O2"
, , O2 = 1.5

    CN
Temp 0.16 0.8 4 20 100
  5     1   1 1  1   1
  15    1   1 1  1   1
  25    1   1 1  1   1

, , O2 = 3

    CN
Temp 0.16 0.8 4 20 100
  5     1   1 1  1   1
  15    1   1 1  1   1
  25    1   1 1  1   1

, , O2 = 9

    CN
Temp 0.16 0.8 4 20 100
  5     1   1 1  1   1
  15    1   1 1  1   1
  25    1   1 1  1   1
```

On a une mesure par concentration et température.
NB : on aurait pu tout obtenir d'un coup en tapant :

```
replications(~(Temp+CN+O2)^3, cyanure)
```

On peut faire une première table d'ANOVA en prenant en compte seulement les effets simples :

```
m2<-lm(Intox.Tps ~ Temp + O2 + CN, cyanure)
anova(m2)
```

```
Analysis of Variance Table

Response: Intox.Tps
```

```
           Df Sum Sq Mean Sq  F value    Pr(>F)
Temp        2  57116   28558 101.9688 1.485e-15 ***
O2          2   3759    1879   6.7105  0.003334 **
CN          4  55545   13886  49.5824 3.690e-14 ***
Residuals 36  10082     280
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Tous les effets sont significatifs, mais y a-t-il des interactions ?

```
m3<-lm(Intox.Tps ~ (Temp + O2 + CN)^2, cyanure)
anova(m3)
```

```
Analysis of Variance Table

Response: Intox.Tps
           Df Sum Sq Mean Sq  F value    Pr(>F)
Temp        2  57116   28558 441.5058 1.007e-14 ***
O2          2   3759    1879  29.0554 4.720e-06 ***
CN          4  55545   13886 214.6823 1.109e-13 ***
Temp:O2     4     97      24   0.3752 0.8229641
Temp:CN     8   3686     461   7.1229 0.0004605 ***
O2:CN       8   5265     658  10.1737 5.460e-05 ***
Residuals  16   1035      65
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

La triple interaction ne peut être testée car il n'y a pas de réplication. Seule l'interaction `Temp:O2` n'est pas significative, on peut représenter ces interactions grâce à la fonction `interaction.plot`. Cette fonction prend en argument un facteur qui sera mis en abscisse (**`x.factor`**), un facteur de trace (**`trace.factor`**), dont chaque modalité définira une droite, et un vecteur de réponse (**`response`**), qui sera mis en ordonnée :

```
interaction.plot(cyanure[,"Temp"], cyanure[,"O2"], cyanure[,"Intox.Tps"])
interaction.plot(cyanure[,"O2"], cyanure[,"Temp"], cyanure[,"Intox.Tps"])
interaction.plot(cyanure[,"CN"], cyanure[,"Temp"], cyanure[,"Intox.Tps"])
interaction.plot(cyanure[,"Temp"], cyanure[,"CN"], cyanure[,"Intox.Tps"])
interaction.plot(cyanure[,"O2"], cyanure[,"CN"], cyanure[,"Intox.Tps"])
interaction.plot(cyanure[,"CN"], cyanure[,"O2"], cyanure[,"Intox.Tps"])
```

[name=sb] Les graphes sont représentés sur la figure 4.1.

4.3.1 Effectifs non-balancés

En toute logique, l'appel à la fonction `anova` devrait renvoyer la même table d'analyse de variance quel que soit l'ordre dans lequel les facteurs explicatifs sont indiqués dans le modèle. En pratique cela n'est vrai que lorsque les effectifs sont « balancés ».

Prenons le cas d'une ANOVA à deux facteurs croisés A et B. Lorsque pour chaque combinaison des facteurs A et B un même nombre d'individus n ont été mesurés, le jeu de données est dit *balancé*. Dans ce cas là `anova(lm(Y ~A*B))` et `anova(lm(Y ~B*A))` produisent le même résultat. Si au contraire le nombre de mesures réalisées varie selon la combinaison des deux facteurs, `anova(lm(Y ~A*B))` et `anova(lm(Y ~B*A))` produisent deux tables d'ANOVA différentes.

La raison de cette différence est que R calcule les carrés des écarts en comparant séquentiellement des modèles où les facteurs sont ajoutés un à un, dans l'ordre d'introduction dans le modèle. Dans l'appel `anova(lm(Y ~A*B))`, la comparaison entre 1 et $1+A$ permet de calculer la variation due à A, puis la comparaison $1+A+B$ celle due à B. Dans l'appel `anova(lm(Y ~B*A))` la dispersion due à A est calculée en comparaison les modèles $1+B$ et $1+B+A$. Ces deux estimations de la dispersion due à A ne sont égales que si les effectifs sont balancés.

Pour résoudre ce problème, on doit estimer la dispersion due à A en comparant le modèle avec A au modèle complet dont le facteur A a été retiré. En procédant de la sorte l'estimation des dispersions ne dépend plus de l'ordre d'introduction.

Un première façon de procéder est de calculer les dispersions une à une, à peu près comme on l'a vu pour la sélection de modèle. Pour tester l'effet du facteur A on procéderait donc de la façon suivante :

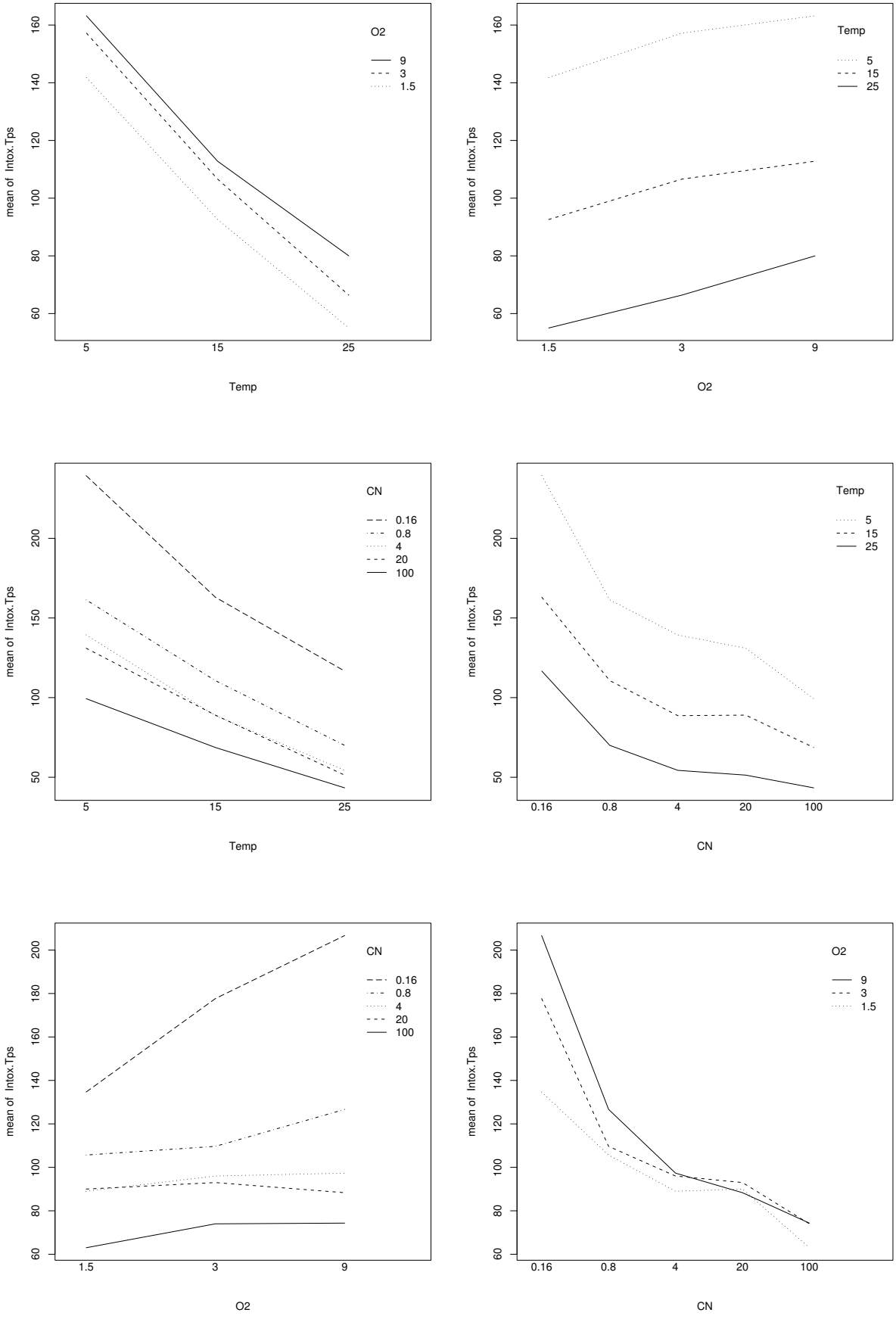


FIG. 4.1 – Interactions 2 à 2 entre les variables "*Temp*", "*O2*" et "*CN*". Les interactions sont d'autant plus fortes que les droites ne sont pas parallèles.

```
m1 <- lm(Y ~ B)
m2 <- lm(Y ~ B + A)
anova(m1,m2)
```

et pour tester celui du facteur B

```
m3 <- lm(Y ~ A)
anova(m3,m2)
```

Une seconde approche pour résoudre ce problème est d'utiliser la fonction `Anova` du paquet `car` (Companion to Applied Regression). Cette fonction calcule les sommes de carrés de type II et III (pour faire un parallèle avec la terminologie du logiciel SAS) qui ne dépendent pas de l'ordre d'introduction des facteurs dans le modèle. Les calculs réalisés sont en fait à peu près équivalents à ce qui est présenté ci-dessus. Cette fonction peut être appliquée aux objets produits par `lm` aussi bien qu'à ceux produit par `glm`.

4.3.2 ANOVA hiérarchisée

Nous allons là encore utiliser un jeu de données tiré du Biometry [2] (p 278). On a mesuré la longueur de l'aile gauche de femelles moustiques (*Aedes intrudens*), élevées dans 4 cages différentes. Pour chaque femelle, on a effectué 2 mesures indépendantes. Récupérez le fichier correspondant ("`Aedes.csv`") et importez-le dans R :

```
aedes<-read.table("data/Aedes.csv", header=T)
```

Ce jeu de données contient 4 variables :

1. "*Cage*" cage d'élevage,
2. "*Femelle*" identifiant de la femelle,
3. "*Mesure*" mesure effectuée,
4. "*LWL*" longueur de l'aile gauche.

Les trois premières sont des facteurs :

```
aedes[,"Cage"]<-as.factor(aedes[,"Cage"])
aedes[,"Femelle"]<-as.factor(aedes[,"Femelle"])
aedes[,"Mesure"]<-as.factor(aedes[,"Mesure"])
```

Ajustement du modèle : le facteur *Mesure* n'est pas pris en compte, on considérera les 2 mesures comme 2 répétitions. Ici, l'effet femelle est emboîté dans celui de cage : la femelle numéro 1 n'est pas la même dans chaque cage ! On ne testera donc pas l'effet "femelle n°1", mais les effets "femelle n°1 dans cage 1", femelle n°1 dans cage 2", *etc*. Un modèle hiérarchisé se déclare de la façon suivante :

```
aedes.lm<-lm(LWL~Cage/Femelle, data=aedes)
```

Ce qui se lit : *LWL* en fonction de *Femelle* dans *Cage*. Ce modèle est équivalent à `LWL~Cage+Cage:Femelle`. L'analyse des résultats est ensuite la même que pour l'ANOVA croisée :

```
anova(aedes.lm)
```

```
Analysis of Variance Table

Response: LWL
             Df  Sum Sq Mean Sq F value    Pr(>F)
Cage          2  665.68  332.84  255.70 1.452e-10 ***
Cage:Femelle  9 1720.68  191.19  146.88 6.981e-11 ***
Residuals    12   15.62    1.30
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

4.3.3 Effet fixe/aléatoire

Les tests effectués dans la section précédente sont ceux du modèle I (effet fixe). La notion de fixe/aléatoire n'intervient pas dans l'ajustement du modèle, seulement dans le calcul de la table d'ANOVA. Le test correspondant a un effet aléatoire de *Femelle* sont :

```
Finter <- 332.84/191.19
Finter
```

```
[1] 1.740886
```

```
pf(Finter, 2, 9, lower.tail=F)
```

```
[1] 0.2295346
```

La fonction `aov` (cf chapitre précédent) permet d'obtenir directement ce résultat. Elle accepte un argument supplémentaire qui spécifie le ou les niveau(x) d'"erreur" contre lequel on doit effectuer le test :

```
aedes.aov<-aov(LWL ~ Cage + Error(Cage:Femelle), aedes)
summary(aedes.aov)
```

```
Error: Cage:Femelle
          Df  Sum Sq Mean Sq F value Pr(>F)
Cage       2  665.68  332.84  1.7409 0.2295
Residuals  9 1720.68  191.19

Error: Within
          Df  Sum Sq Mean Sq F value Pr(>F)
Residuals 12 15.6200  1.3017
```

Exercice 16 *Rendement de Luzerne.*

Etudiez le jeu de données Alfalfa, fourni avec R (paquetage nlme). Le plan expérimental est le suivant : On a 6 blocs contenant chacun 4 sous-blocs. On dispose de 4 variétés de luzerne pour lesquelles on veut mesurer le rendement. Les quatre espèces on été assignées aléatoirement aux quatre sous-blocs, pour chacun des 6 blocs. Les rendements ont été mesurés à 4 périodes de l'année.

Que pouvez vous dire sur le rendement de la luzerne ?

4.4 Variables quantitatives et qualitatives : l'analyse de covariance.

L'ANCOVA est également un cas particulier de modèle linéaire, avec cette fois une variable qualitative et une quantitative. Elle s'effectue donc de manière similaire à l'ANOVA et la régression linéaire. Voici un exemple tiré du Biometry [2] (p505). Il s'agit d'une étude de potentiel de membrane en fonction de différents types d'électrolytes à différentes concentrations. Récupérez le fichier `MembranePotential.csv` et importez-le dans R :

```
mp<-read.table("data/MembranePotential.csv", header=T, dec=",")
```

On peut représenter les corrélations pour chaque type d'électrolyte avec la fonction `coplot`, qui prend une formule en argument :

```
coplot(logActivity ~ MeanMembranePotential|Cation, mp)
```

On peut aussi tout représenter sur un même graphe avec des figurés différents :

```
plot(logActivity ~ MeanMembranePotential, pch=as.numeric(Cation), mp)
```

Les graphes sont représentés sur la figure 4.2.

```
m.mp<-lm(logActivity ~ MeanMembranePotential * Cation, mp)
summary(m.mp)
```

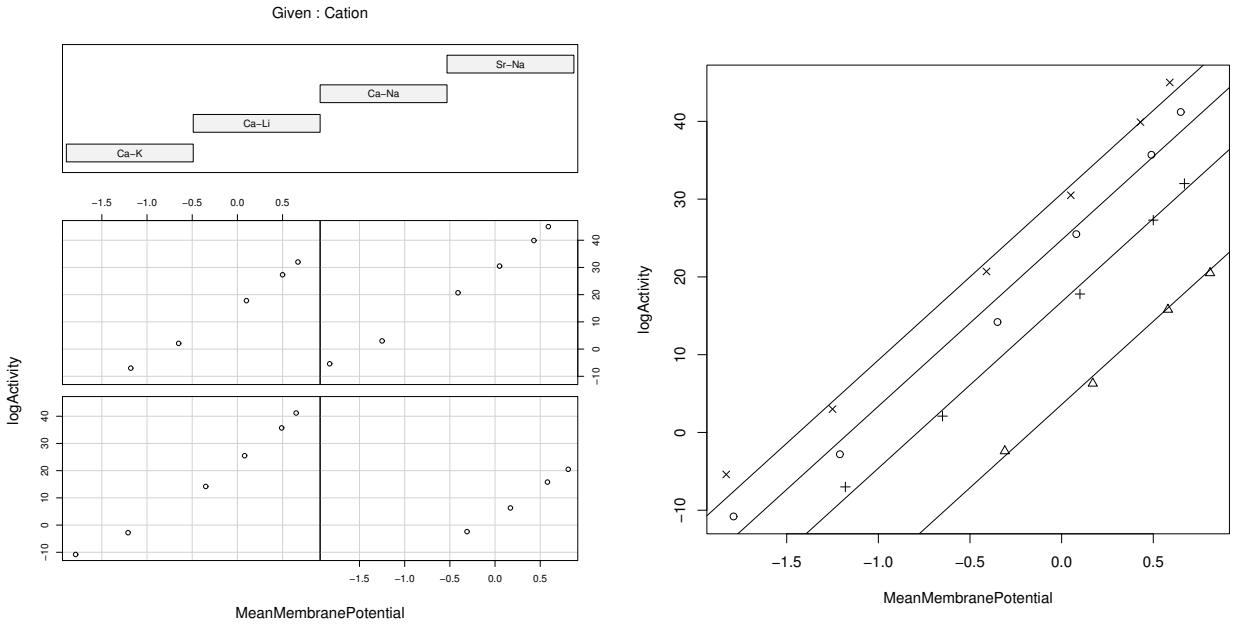


FIG. 4.2 – ANCOVA : graphes. Les droites sur le graphe de droite sont les droites de régression. Aucune différence de pente significative n'a été trouvée, par contre les ordonnées à l'origine sont significativement différentes.

```
Call:
lm(formula = logActivity ~ MeanMembranePotential * Cation, data = mp)

Residuals:
    Min      1Q  Median      3Q     Max
-3.0736 -1.1065  0.1687  1.0612  2.7338

Coefficients:
                                  Estimate Std. Error t value Pr(>|t|)
(Intercept)                        24.7616     0.8393  29.504 2.69e-13 ***
MeanMembranePotential              21.3941     0.8798  24.316 3.19e-12 ***
CationCa-Li                       -21.1696     1.4504 -14.595 1.93e-09 ***
CationCa-Na                        -7.9619     1.2046  -6.609 1.69e-05 ***
CationSr-Na                         5.9383     1.1995   4.950 0.000265 ***
MeanMembranePotential:CationCa-Li  -0.7282     2.4048  -0.303 0.766830
MeanMembranePotential:CationCa-Na  -0.3255     1.5012  -0.217 0.831707
MeanMembranePotential:CationSr-Na  -0.5266     1.2486  -0.422 0.680085
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.908 on 13 degrees of freedom
Multiple R-Squared: 0.9921,     Adjusted R-squared: 0.9879
F-statistic: 234.1 on 7 and 13 DF,  p-value: 1.210e-12
```

```
6 anova(m.mp)
```

```
Analysis of Variance Table

Response: logActivity
                          Df Sum Sq Mean Sq   F value    Pr(>F)
MeanMembranePotential      1 4197.0  4197.0 1152.7173 4.431e-14 ***
Cation                     3 1768.6   589.5  161.9151 1.521e-10 ***
```

```
MeanMembranePotential:Cation  3    0.8     0.3    0.0729    0.9735
Residuals                    13   47.3     3.6
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

On a un effet significatif du potentiel membranaire et de la nature de l'électrolyte, mais pas d'interaction. Autrement dit, la pente de la droite est la même pour chaque électrolyte, mais les ordonnées à l'origine diffèrent. On peut tracer ces droites sur le deuxième graphe (cf. figure 4.2).

```
co<-coef(m.mp)
co
```

```
                        (Intercept)                MeanMembranePotential
                         24.7615556                           21.3940534
                      CationCa-Li                          CationCa-Na
                      -21.1696382                           -7.9618774
                      CationSr-Na MeanMembranePotential:CationCa-Li
                        5.9383122                           -0.7281893
MeanMembranePotential:CationCa-Na MeanMembranePotential:CationSr-Na
                       -0.3254982                           -0.5266126
```

```
abline(co[1], co["MeanMembranePotential"])
abline(co[1]+co["CationCa-Li"], co["MeanMembranePotential"])
abline(co[1]+co["CationSr-Na"], co["MeanMembranePotential"])
abline(co[1]+co["CationCa-Na"], co["MeanMembranePotential"])
```

4.5 Selection de modèles

Nous avons vu au chapitre 3 que R peut calculer l'AIC d'un modèle et faire ainsi des comparaisons. Les fonctions `add1` et `drop1` permettent respectivement d'ajouter ou d'enlever un effet du modèle et de tester l'apport par rapport au modèle initial. Deux tests sont possibles : F et χ^2, se reporter à la documentation. On peut ainsi tester tous les effets et garder le modèle "minimal" qui offre le meilleur compromis entre "ajustement aux données" et "nombre de paramètres". La fonction `step` permet de sélectionner automatiquement un tel modèle. Exemple avec le jeu de données 'cyanure' :

```
drop1(m3, ~O2:CN, test="Chisq")
```

```
Single term deletions

Model:
Intox.Tps ~ (Temp + O2 + CN)^2
       Df Sum of Sq    RSS    AIC   Pr(Chi)
<none>              1034.9  199.1
O2:CN   8    5264.5 6299.5  264.4 2.706e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
drop1(m3, ~Temp:CN, test="Chisq")
```

```
Single term deletions

Model:
Intox.Tps ~ (Temp + O2 + CN)^2
        Df Sum of Sq    RSS    AIC   Pr(Chi)
<none>               1034.9  199.1
Temp:CN  8    3685.9 4720.8  251.4 1.073e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
3 drop1(m3, ~Temp:O2, test="Chisq")
```

```
Single term deletions

Model:
Intox.Tps ~ (Temp + O2 + CN)^2
        Df Sum of Sq     RSS     AIC Pr(Chi)
<none>               1034.93  199.09
Temp:O2  4     97.07 1132.00  195.13  0.4014
```

Il apparaît qu'enlever l'interaction `Temp:O2` ne diminue pas significativement l'ajustement du modèle. On peut enlever ce terme du modèle avec la fonction `update`.

```
4 new.m3<-update(m3, . ~ . - Temp:O2)
5 anova(new.m3)
```

```
Analysis of Variance Table

Response: Intox.Tps
          Df Sum Sq Mean Sq  F value    Pr(>F)
Temp       2  57116   28558 504.5595 < 2.2e-16 ***
O2         2   3759    1879  33.2049 4.412e-07 ***
CN         4  55545   13886 245.3422 < 2.2e-16 ***
Temp:CN    8   3686     461   8.1402 7.145e-05 ***
O2:CN      8   5265     658  11.6266 5.096e-06 ***
Residuals 20   1132      57
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

La même chose automatiquement avec la fonction `step` :

```
6 step(m3)
```

```
Start:  AIC= 199.09
 Intox.Tps ~ (Temp + O2 + CN)^2

          Df Sum of Sq    RSS    AIC
- Temp:O2  4      97.1 1132.0  195.1
<none>                1034.9  199.1
- Temp:CN  8    3685.9 4720.8  251.4
- O2:CN    8    5264.5 6299.5  264.4

Step:  AIC= 195.13
 Intox.Tps ~ Temp + O2 + CN + Temp:CN + O2:CN

          Df Sum of Sq    RSS    AIC
<none>                1132.0  195.1
- Temp:CN  8    3685.9 4817.9  244.3
- O2:CN    8    5264.5 6396.5  257.1

Call:
lm(formula = Intox.Tps ~ Temp + O2 + CN + Temp:CN + O2:CN, data = cyanure)

Coefficients:
 (Intercept)        Temp15        Temp25          O23          O29
      201.00        -76.33       -122.67        43.00        72.00
       CN0.8           CN4          CN20        CN100  Temp15:CN0.8
```

```
      -48.00        -66.78        -70.44       -109.11         25.67
Temp25:CN0.8    Temp15:CN4    Temp25:CN4   Temp15:CN20   Temp25:CN20
       31.33         25.67         37.67         34.33         43.00
Temp15:CN100  Temp25:CN100     O23:CN0.8     O29:CN0.8       O23:CN4
       45.67         66.67        -39.00        -51.00        -36.00
     O29:CN4       O23:CN20      O29:CN20      O23:CN100     O29:CN100
      -63.67        -40.00        -73.67        -32.00        -60.67
```

La fonction renvoie le modèle le plus simple retenu. Il est possible de paramétrer les critères de sélection (voir documentation de la fonction).

Exercice 17 *Fertilité des Suisses*
Étudier le jeu de données "swiss" fourni avec R. Quels facteurs expliquent le mieux la fertilité des Suisses ?

4.6 Non orthogonalité

Des problèmes statistiques surviennent s'il y a plus d'un niveau d'erreur et que les données ne sont pas balancées (= également distribuées dans chaque niveau = même nombre de répétitions dans chaque cas = plan orthogonal). On peut vérifier cela avec la commande `replications`.

4.6.1 Modèle I

Les moindres carrés de type 1 conduisent à des tests erronés lorsque les données ne sont (très) déséquilibrées. On utilise alors des moindres carrés de type 2 ou 3, selon la notation utilisée par le logiciel SAS. Le paquetage *car* dispose d'une fonction `Anova` qui fonctionne comme `anova`, mais en permettant de calculer des carrés moyens de type 2 et 3.

4.6.2 Modèle II

Si on a des effets aléatoires, il faut utiliser le paquetage *nlme*. Le paquetage *nlme* est dédié à l'analyse des modèles mixtes, c'est à dire ayant des effets fixes et aléatoires. Il peut être utilisé pour ajuster des modèles purement aléatoires, mais ne permet pas d'ajuster des modèles fixes (on utilise pour cela `lm`). *nlme* possède principalement 2 fonctions, `lme` et `nlme`, l'une pour les modèles linéaires, l'autre pour les modèles non-linéaires. Son fonctionnement est assez complexe, voici comment reproduire l'exemple ci-dessus avec `lme`. Le fonctionnement est identique si le modèle n'est pas balancé.

```
12 library(nlme) #Charge la bibliothèque
13 aedes.lme<-lme(LWL~1, random=~1|Cage/Femelle, data=aedes)
```

`lme` prend en argument une formule, comme `lm`, mais qui contient seulement les effets fixes (ici aucun). Les variables aléatoires sont précisées dans l'argument **`random`**. La formule est particulière : elle s'interprète comme ceci : un effet aléatoire par niveau, les niveaux étant définis par "*Cage*" et "*Femelle* dans *Cage*". On peut afficher la table d'ANOVA de ce modèle :

```
14 anova(aedes.lme)
```

```
            numDF denDF  F-value p-value
(Intercept)     1    12 321.7538  <.0001
```

La table n'est calculée que pour les effets fixes, donc ici, seulement l'intercept. Étant donné que `lme` est écrite pour le cas général où les effectifs ne sont pas balancés, `anova(aedes.lme)` ne donne pas les tests pour les effets aléatoires (il n'existe pas de test exact dans ce cas). On peut cependant faire de la comparaison de modèle.

Considérons tout d'abord le cas d'un modèle I :

```
15 aedes.lm1<-update(aedes.lm, .~.-Cage:Femelle)
16 aedes.lm2<-update(aedes.lm1, .~.-Cage)
```

La fonction `update` construit un modèle à partir d'un autre, elle prend en argument un objet `lm` (ou assimilé) et une formule décrivant le nouveau modèle. Dans cette formule, on peut utiliser le caractère '`.`' qui signifie "même chose que la formule d'origine". Ici *aedes.lm1* est identique à *aedes.lm* sans l'interaction, et *aedes.lm2* contient seulement l'intercept. On peut comparer les modèles avec la fonction `anova` :

```
17 anova(aedes.lm, aedes.lm1, aedes.lm2)
```

```
Analysis of Variance Table

Model 1: LWL ~ Cage/Femelle
Model 2: LWL ~ Cage
Model 3: LWL ~ 1
  Res.Df      RSS Df Sum of Sq      F    Pr(>F)
1     12    15.62
2     21  1736.30 -9  -1720.68 146.88 6.981e-11 ***
3     23  2401.97 -2   -665.68 255.70 1.452e-10 ***
```

A comparer avec le résultat de

```
18 anova(aedes.lm)
```

```
Analysis of Variance Table

Response: LWL
             Df  Sum Sq Mean Sq F value    Pr(>F)
Cage          2  665.68  332.84  255.70 1.452e-10 ***
Cage:Femelle  9 1720.68  191.19  146.88 6.981e-11 ***
Residuals    12   15.62    1.30
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Vu que seuls des effets fixes sont en jeu, la fonction `anova` a utilisé un test F pour comparer les modèles. On aurait pu la forcer à faire un χ^2 sur modèles emboîtés en ajoutant l'argument `test="Chisq"`.

On peut procéder de la sorte pour tester les effets aléatoires d'un modèle `lme` :

```
19 aedes.lme1<-update(aedes.lme, random=~1|Cage)
```

On compare les modèles :

```
20 anova(aedes.lme, aedes.lme1, aedes.lm2)
```

```
           Model df      AIC      BIC    logLik   Test  L.Ratio p-value
aedes.lme      1  4 138.5076 143.0495 -65.25378
aedes.lme1     2  3 178.7793 182.1858 -86.38964 1 vs 2 42.27172  <.0001
aedes.lm2      3  2 179.3659 181.6369 -87.68296 2 vs 3  2.58663  0.1078
```

Le facteur "*Cage*" n'est pas significatif, alors que "*Femelle*" oui. Les conclusions sont donc identiques à celles obtenues par `aov`, mais les tests sont différents : en effet `lme` ne permettant pas de faire un test F, c'est un test de χ^2 sur modèles emboîtés qui est effectué.

4.7 Autres modèles

4.7.1 Modèle linéaire généralisé

R permet également d'ajuster des modèles linéaires généralisés, *via* la fonction `glm`. Cette fonction est identique à `lm`, mais prend en plus en argument un terme spécifiant la 'famille' considérée, à savoir la distribution des résidus. Il est également possible de faire des modèles dits 'quasi-likelihood' sans homoscédasticité (se reporter à la doc et au tutoriel de *R*).

Nous présentons ici le cas de la régression dite "binomiale". Reprenons le jeu de données "*becs*", et effectuons la régression de la survie en fonction de la taille du bec.

```
1 becs<-read.table(file="data/beak_size.csv", header=TRUE, sep="\t", dec=",")
```

La survie est codée par '0' ou '1', sa distribution est une distribution binomiale. On utilisera donc la fonction `glm` avec l'argument **`family="binomial"`** :

```
plot(survival~beak_size, data=becs)
becs.glm<-glm(survival~beak_size, family="binomial", data=becs)
summary(becs.glm)
```

```
Call:
glm(formula = survival ~ beak_size, family = "binomial", data = becs)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.1443  -0.5406  -0.4651  -0.3867   2.7951

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -6.6597     1.1544  -5.769 7.98e-09 ***
beak_size     0.4742     0.1147   4.133 3.58e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 550.64  on 750  degrees of freedom
Residual deviance: 532.50  on 749  degrees of freedom
AIC: 536.5

Number of Fisher Scoring iterations: 5
```

On a un effet positif significatif de la taille du bec sur la survie. On pourra se reporter à la section §B.2.3 pour plus de détails sur les GLM.

4.7.2 Maximum de vraisemblance

De la même manière, *R* permet d'ajuster des modèles selon le principe du maximum de vraisemblance. On utilisera pour se faire la fonction `nlm`.

Annexe A

Quelques fonctions graphiques

A.1 Modifier les paramètres graphiques

La commande `par` permet de récupérer et de modifier de nombreux paramètres graphiques. On citera notamment :

`bg` La couleur de fond.

`cex`, `cex.axis`, `cex.lab`, `cex.main`, `cex.sub` La taille des caractères et des symboles : généraux, pour les axes, pour les étiquettes, pour le titre et pour le sous-titre.

`col`, `col.axis`, `col.lab`, `col.main`, `col.sub` Les couleurs par défaut : générales, pour les axes, pour les étiquettes, pour le titre et pour le sous-titre.

`family` La famille de police de caractère à utiliser.

`font`, `font.axis`, `font.lab`, `font.main`, `font.sub` 1 pour normal (par défaut), 2 pour gras, 3 pour italique, 4 pour gras et italique.

`las` Etiquettes : 0 (défaut) toujours parallèles aux axes, 1 toujours horizontales, 2 toujours perpendiculaires aux axes, 3 toujours verticales.

`lty` Type de ligne, une valeur ou une chaîne de caractères parmi : 0=blank, 1=solid (par défaut), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash

`lwd` Largeur des lignes (1 par défaut).

`oma`, `omi`, `mar`, *etc* Marges.

`pch` Type de symbole à utiliser pour les points, peut être un entier ou un caractère.

`xlog`, `ylog` Échelle logarithmique (FALSE par défaut).

etc Beaucoup d'autres options disponibles, voir `help(par)` pour une liste complète.

Certaines de ces options peuvent être directement passées aux fonctions graphiques (**`lty`**, **`pch`**, **`lwd`**, *etc*), nous en avons déjà vu des exemples. Si elles sont spécifiées par la commande `par`, elles seront effectives pour tous les graphes.

A.2 Graphes composés

On peut afficher plusieurs graphiques sur une même fenêtre. Il existe principalement 2 méthodes. Nous en avons vu une lors de l'affichage des graphes de validation du modèle linéaire. Cette première méthode permet de diviser une fenêtre graphique en une grille de sous-fenêtres. Les graphes seront ensuite tracés dans chaque sous-fenêtre, les uns à la suite des autres. On utilise pour cela 2 arguments de la fonction `par`, **`mfrow`** et **`mfcol`**, qui prennent tous les 2 un vecteur de taille 2, contenant le nombre de lignes et le nombre de colonnes. La différence entre ces deux commandes est qu'avec la première les graphes seront tracés ligne par ligne, alors qu'il seront tracés colonne par colonne avec la seconde.

La seconde méthode utilise la fonction `split.screen`, qui prend en argument un vecteur contenant le nombre de lignes et de colonnes, de manière similaire à la méthode précédente. La différence est que l'on peut choisir dans quelle sous-fenêtre effectuer son graphe en utilisant la commande `screen`. Exemple :

```
1 split.screen(c(2,3))
2 screen(2)
3 hist(rnorm(1000), col="red", main="Normale")
```

```
screen(4)
hist(rgamma(1000, shape=1), col="blue", main="Gamma")
screen(6)
hist(rexp(1000), col="green", main="Exponentielle")
```

On peut fermer les sous-fenêtres en tapant :

```
close.screen(all=TRUE)
```

Un autre avantage de `split.screen` est qu'on peut l'utiliser sur une sous-fenêtre afin de la subdiviser. Exemple :

```
split.screen(c(1,2))
screen(1)
contour(volcano, color = terrain.colors)
screen(2)
split.screen(c(2,2))
screen(3)
hist(rnorm(1000, sd=0.1), col="red")
screen(4)
hist(rnorm(1000, sd=0.4), col="blue")
screen(5)
hist(rnorm(1000, sd=1.0), col="green")
screen(6)
hist(rnorm(1000, sd=2.0), col="yellow")
close.screen(all=TRUE)
```

A.3 Fonctions de dessin

R possède des fonctions qui permettent d'ajouter des dessins à un graphique. Nous avons déjà vu la fonction `lines` qui ajoute des points reliés. On lui passe en argument un vecteur d'abscisses et un vecteur d'ordonnées :

```
x<-seq(from=-5,to=5,by=0.01)
plot(x,x^2-2*x-1, type="l", col="red")
lines(x,x^2-3*x-1, col="blue", lty=2)
lines(x,x^3+x^2-5*x-1, col="blue", lty=4)
```

Plusieurs fonctions similaires existent :

```
x<-seq(from=-5,to=5,by=0.5)
points(x,x^3-4*x^2-x+15, col="green", pch=2)
```

```
x0<- -2
x1<- 2
y0<-seq(from=0,to=20,by=5)
y1<-y0+2
segments(x0,y0,x1,y1)
arrows(x0-2,y0+2,x1-2,y1+2,col="purple")
```

On dispose aussi des fonctions `rect` pour tracer des rectangles et `polygon` pour des polygones. La fonction `text` permet d'ajouter du texte à un graphique. Finalement, la fonction `abline` permet de tracer une droite dont on connaît l'ordonnée à l'origine et le coefficient directeur. Plus généralement, la fonction `curve` permet de tracer un courbe dont on précise l'équation. Un de ces intérêts est qu'elle fonctionne directement avec des objets `lm` et permet de tracer une droite de régression, comme nous l'avons vu au chapitre 3.

A.4 Le paquetage *lattice*

R possède également une bibliothèque de fonction de dessin avancée, la bibliothèque `lattice`. On la charge en tapant :

```
library(lattice)
```

Les fonctions de graphisme de cette bibliothèque ne sont pas compatibles avec les fonctions de base de R, on ne peut donc pas les combiner. Il existe cependant un équivalent pour (presque) chaque fonction graphique. On se reportera au mémento pour obtenir le nom des fonctions lattices équivalentes à celles utilisées dans ce TP, et on se reportera à la documentation de R pour leur utilisation.

A.5 Sauvegarder un graphe dans un fichier

Plusieurs fonctions permettent de sauvegarder un graphe sous forme de fichier image. La plus générique est `dev.print`, qui copie le contenu d'une fenêtre graphique dans un fichier. Elle prend plusieurs arguments :

- **`device`** en gros : le format du fichier, à choisir parmi :

  jpeg Le fameux format photo.

  png Mieux que jpeg, avec gestion de la transparence. Attention cependant, certains windows font des caprices pour lire ces fichiers.

  bitmap Fichier .bmp de windows, non compressés, à proscrire donc.

  pdf Format Adobe.

  postscript Avantage : fichiers vectoriels, très petits. Inconvénients : non lisible sous windows à moins d'installer les logiciels adéquats, difficilement importable dans des documents de type Office (MS ou Open).

  pictex Format LaTeX.

  xfig Fichier .fig, pour les amateurs...

  NB : lors de la spécification du format, on ne met pas de quotes : `png` et non `"png"`.
- **`file`** Le chemin vers le fichier, avec des guillemets.
- **`width`** La largeur du fichier en pixels (400 est une bonne valeur).
- **`height`** La hauteur du fichier en pixels (400 est une bonne valeur).
- **`quality`** (pour les jpeg) la valeur de compression (en pourcentage). 75 par défaut. Une valeur de 100 ne compresse pas les fichiers.

Exemple :

```
hist(rnorm(1000), col=rainbow(15))
dev.print(png, file="LoiNormale.png", width=400, height=400)
```

Annexe B

Que faire quand l'hypothèse de normalité n'est pas vérifiée ?

B.1 Cas des tests « simples »

Exemple : `t.test`, `var.test`, `cor.test`, *etc*. Dans ce cas, il existe (presque) toujours un test non-paramétrique ayant la même hypothèse nulle. Il est également possible de construire votre propre test, en utilisant par exemple des permutations, comme vu lors du premier TD.

B.2 Cas du modèle linéaire

(régression simple ou multiple, ANOVA à 1 ou plusieurs facteurs, ANCOVA, modèles plus complexes).

On rappelle que dans ce cas, c'est la normalité des résidus qui importe. Les résidus d'un modèle peuvent être récupérés avec la commande `resid`.

B.2.1 Solution n° 1 : tests non-paramétriques

Si vous êtes dans le cas d'un modèle simple, vous pouvez soit utiliser un test de corrélation non paramétrique (Spearman ou Kendall), bien que les hypothèses ne soient pas strictement identiques à celles du modèle linéaire. Dans le cas d'une ANOVA à un facteur, vous pouvez utiliser le test de Kruskal-Wallis.

B.2.2 Solution n° 2 : normaliser les données

Dans certains cas, il est possible de normaliser les données en les transformant à l'aide d'une fonction qui conserve leur rang (fonction *monotone*). Les fonctions typiquement utilisées sont les fonctions `log`, `sqrt` (racine carré), `exp`, $1/x$ ou `arcsin`. Attention cependant, ces fonctions ne sont pas toujours définies : `log` par exemple ne prendra que des valeurs positives.

Il existe une approche plus élégante et plus puissante qui fait appel au paquet MASS, qui fournit deux types de transformations.

Transformation $\log(y+\alpha)$ La transformation testée ici est du type

$$\log(y+\alpha) \sim x_1 + x_2 + \cdots \tag{B.1}$$

et on cherche le meilleur α qui permettrait de normaliser les données. On utilise pour cela la fonction `logtrans`, qui prend en argument un objet `lm`.

```
library(MASS)
m<-lm(Days ~ Age*Sex*Eth*Lrn, data = quine)
shapiro.test(resid(m))
```

```
> shapiro.test(resid(m))

        Shapiro-Wilk normality test

data:  resid(m)
W = 0.9612, p-value = 0.0003913
```

On utilise alors la fonction `logtrans` :

```
logtrans(m, alpha = seq(0.75, 6.5, len=20))
```

En lui précisant les valeurs de α à essayer (ici, 20 valeurs de 0,75 à 6,5). La détermination de la valeur se fait graphiquement. On peut également la déterminer en utilisant :

```
lt<-logtrans(m, alpha = seq(0.75, 6.5, len=20))
alpha<-lt$x[which.max(lt$y)]
alpha
```

Ce qui nous donne la valeur 2,492. On peut ensuite faire un nouveau modèle :

```
m2<-lm(log(Days+alpha) ~ Age*Sex*Eth*Lrn, data = quine)
shapiro.test(resid(m2))
```

```
> shapiro.test(resid(m2))

        Shapiro-Wilk normality test

data:  resid(m2)
W = 0.9863, p-value = 0.1586
```

Transformation Box-Cox La transformation Box-Cox est un peu plus tordue, mais le principe reste le même. La transformation testée est ici

$$\begin{cases} \dfrac{y^\lambda - 1}{\lambda} & \text{si } \lambda \neq 0 \\ \log(y) & \text{si } \lambda = 0 \end{cases} \sim x_1 + x_2 + \cdots \tag{B.2}$$

et la fonction à utiliser s'intitule `boxcox`. Son utilisation est similaire à `logtrans`. Nous travaillons ici sur un modèle similaire :

```
library(MASS)
m<-lm(Days+1 ~ Age*Sex*Eth*Lrn, data = quine)
shapiro.test(resid(m))
```

```
> shapiro.test(resid(m))

        Shapiro-Wilk normality test

data:  resid(m)
W = 0.9612, p-value = 0.0003913
```

```
bc<-boxcox(m,lambda = seq(-0.05, 0.45, len = 20))
lambda<-bc$x[which.max(bc$y)]
lambda
```

ce qui nous donne la valeur de 0,213. On crée alors un nouveau modèle :

```
m2<-lm( ((Days+1)^lambda)/lambda ~ Age*Sex*Eth*Lrn, data = quine)
shapiro.test(resid(m2))
```

```
> shapiro.test(resid(m2))

        Shapiro-Wilk normality test

data:  resid(m2)
W = 0.9891, p-value = 0.3133
```

B.2.3 Solution n° 3 : utilisation d'un modèle linéaire généralisé (GLM)

Comme son nom l'indique, il s'agit d'une généralisation du modèle linéaire qui autorise une variable non normale, mais également l'hétéroscédasticité.

On a typiquement besoin des GLM lorsque la variable à expliquer n'est pas définie sur $]-\infty,\infty[$. C'est le cas par exemple de données de fréquence ($[0,1]$) ou de comptages (nombres entiers positifs). On utilise alors une fonction qui permet de transformer cette variable en une variable définie sur l'intervalle $]-\infty,\infty[$. Cette fonction ne modifie pas l'ordre des valeurs (elle est *monotone*). Par exemple, pour des fréquences, on peut utiliser

$$g : p \to log\left(\frac{p}{1-p}\right), \tag{B.3}$$

une fonction que l'on nomme *logit*. Pour une variable définie seulement de manière positive (tels des comptages), la fonction `log` permet d'obtenir une variable transformée définie sur $]-\infty,\infty[$. On appelle *fonction de lien* de telles fonctions g. Un modèle linéaire généralisé s'écrit donc

$$g(\hat{y}_i) = a.x_i + b + \epsilon_i \tag{B.4}$$

dans le cas d'une régression linéaire simple par exemple. Les ϵ dans ce cas suivent une distribution particulière, qui dépend de ce qu'on appelle la *famille* du modèle linéaire généralisé. Un modèle linéaire est ainsi un cas particulier de modèle linéaire généralisé avec une erreur normale (famille gaussienne) et un lien identité.

Voici quelques cas usuels :

Famille	Fonction de lien	Formule	Nom usuel
Gaussienne	Identité	$g(x) = x$	Modèle linéaire
Poisson	Log	$g(x) = \log(x)$	Régression de Poisson (comptages)
Binomiale	Logit	$g(x) = \log\left(\frac{x}{1-x}\right)$	Régression logistique
Multinomiale	Logit	$g(x) = \log\left(\frac{x}{1-x}\right)$	
Exponentielle	Inverse	$g(x) = \frac{1}{x}$	
Gamma	Inverse	$g(x) = \frac{1}{x}$	

Un modèle linéaire généralisé s'effectue avec la fonction `glm`, qui suit la même syntaxe que `lm`, mais prend un argument supplémentaire intitulé **`family`**. Celui-ci est un objet de type *family* qui décrit le type de GLM, notamment la fonction de lien. *R* possède différents objets *family* prédéfinis, correspondant aux cas les plus courants (voir la documentation pour « family »). Exemple :

```
1 m<-glm(y~x, data, family=binomial(link="logit"))
2 plot(m)
3 summary(m)
```

La plupart des fonctions qui marchent avec `lm` fonctionnent également avec `glm`.

Pour information, on peut également spécifier la variance comme une fonction des valeurs prédites, ce qui permet dans certains cas de régler les problèmes d'hétéroscédasticité.

Annexe C

Programmer avec R : la dérive génétique

Ce chapitre aborde quelques aspects 'avancés' de l'utilisation de R. **Consigne importante : ce chapitre est linéaire, ne pas passer à la suite tant qu'un exercice n'est pas terminé (et réussi)**.

C.1 Le sujet…

On considère une population d'organismes haploïdes, et on étudie la fréquence d'un gène de résistance. Pour les locus considérés, on a deux allèles : "R" pour résistant et "S" pour sensible. Chaque individu dans la population émet le même nombre de gamètes dans le milieu, et ces gamètes ont la même probabilité de donner lieu à un individu qui arrivera à maturité (pas de sélection). Qui plus est, on considérera que la taille de la population est constante au cours du temps. On étudie **l'évolution de la fréquence des allèles résistants au cours du temps**.

La seule source de variabilité est le hasard lié à la reproduction, à l'échantillonnage aléatoire des gamètes pour former la génération suivante (figure C.1). Cela revient en fait à choisir aléatoirement dans la population à t les individus qui donneront un descendant à la génération $t+1$, sachant qu'un individu peut-être choisi plusieurs fois, et un autre aucune fois. La fréquence à $t+1$ peut donc être différente de la fréquence à t, par le seul effet de l'échantillonnage aléatoire.

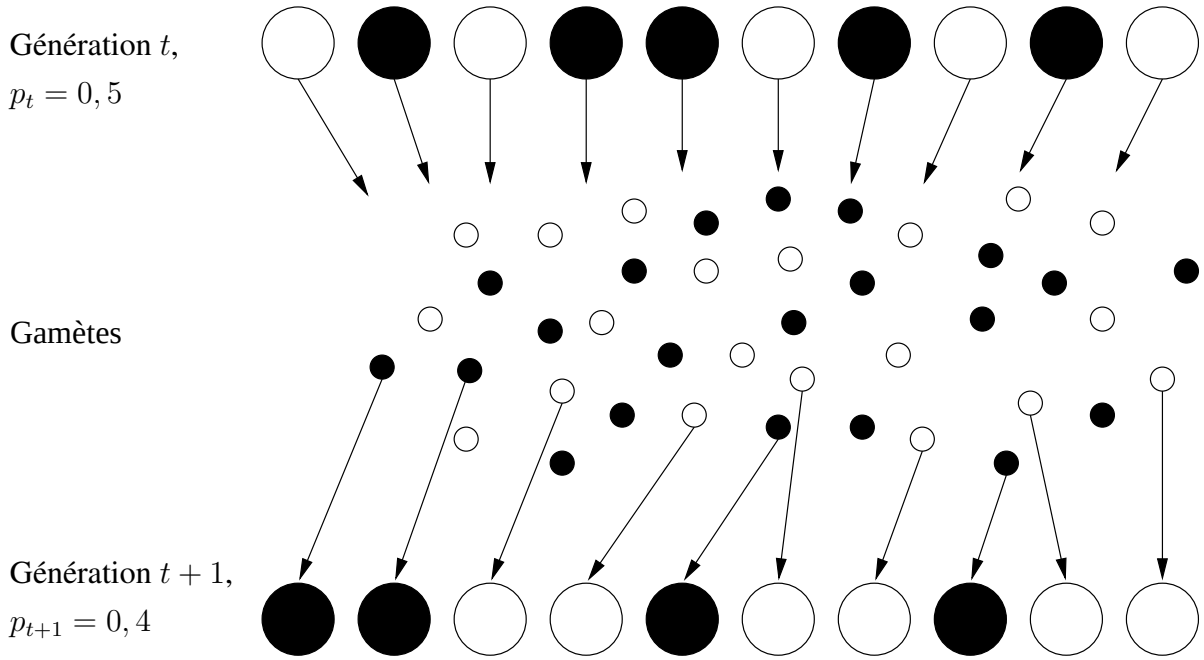


FIG. C.1 – Effet de la fécondation aléatoire sur la fréquence d'un gène de résistance dans une population de taille 10.

Exercice 18 – *Créer une population de taille 10, contenant 50% de résistants. Il s'agira d'un vecteur de "R" (résistant) et de "S" (sensibles), nommé pop.*
– *Créer un autre vecteur comprenant 10 descendants choisis au hasard parmi les individus de pop. Aide : regarder la documentation de la fonction* `runif` *qui permet de générer des nombres aléatoires selon une loi uniforme.*
– *Calculer la fréquence des allèles résistants à la nouvelle génération.*
– *Recommencer les deux dernières étapes plusieurs fois, faire un histogramme des fréquences obtenues et calculer leur moyenne et variance.*

C.2 Évolution de la population après plusieurs générations

Créer ses propres fonctions avec R Les fonctions, dans R, sont des objets de classe `function` (!). La syntaxe est la suivante :

nom_de_fonction `<-` `function(` **param1**, **param2**, **param3**, etc `)` *corps_de_fonction*

– **param1**, **param2**, **param3** sont autant de paramètres à passer à la fonction.
– *corps_de_fonction* contient ce que fait la fonction. Pour les fonctions simples (1 ligne), il s'agit généralement d'une formule mathématique. Ex :

```
f <- function(x) x^2
```

est la fonction 'carré', `f(2)` renvoie 4. Si la fonction est plus complexe (plusieurs lignes), il faut entourer le corps de la fonction avec des accolades. Ex :

```
centre.et.reduit <- function(x) {
  moyenne     <- mean(x)
  ecart.type <- sd(x)
  y <- (x - moyenne) / ecart.type
  return(y)
}
```

Cette fonction centre et réduit (retranche la moyenne et divise par l'écart-type) la variable x et renvoie le résultat. La variable centrée réduite a une moyenne de 0 et un écart-type de 1 (tester la fonction et le vérifier !). Plusieurs remarques :
– Il faut spécifier la valeur à retourner avec la commande `return`.
– Les variables *moyenne*, *variance*, *y*, ainsi que les paramètres sont des variables **locales**, c'est à dire qu'elles n'existent que dans la fonction (elles n'apparaîtront pas en tapant `ls()` pr exemple). Lorsqu'on passe les valeurs aux paramètres lors de l'appel de la fonction, de même que lorsqu'on renvoie le résultat, les valeurs sont recopiées (on parle de passage par valeur). Regarder l'exemple suivant :

```
f <- function(x) {
  x <- 3*x^2-5
  y <- x/4
  return(y)
  }
x <- 0.5
ls()
```

```
[1] "f" "x"
```

```
f(x)
```

```
[1] -1.0625
```

```
x
```

```
[1] 0.5
```

```
y
```

```
Error: Object "y" not found
```

La fonction a modifié *localement* la valeur de la variable *x*, cependant la variable **globale** *x* a toujours la valeur 0.5. De même, la variable *y*, créée localement, n'est pas visible en dehors de la fonction.

Exercice 19 *Créer les deux fonctions suivantes :*
- `freq` *qui prend en argument un vecteur population et renvoie la fréquence des résistants,*
- `popgen` *qui crée une nouvelle population en tirant au hasard les descendants d'une population 'mère' spécifiée en argument.*
- *tester vos fonctions ! Donnent-elles les bons résultats ?*

Créer des boucles dans R Comme la plupart des langages de programmation, R permet de structurer le code dans des **boucles**. Voici comment créer une boucle de type **for** :

`for(` *variable* `in` *vecteur* `)` *corps_de_la_boucle*

- *variable* est une variable compteur qui prendra la valeur de chaque élément de *vecteur* à chaque itération. Au premier passage, elle est égale à *vecteur[1]*, puis au deuxième à *vecteur[2]*, jusqu'à *vecteur[length(vecteur)]*. Le corps de la boucle est exécuté entièrement à chaque itération. Lorsque tout le vecteur est parcouru, la boucle s'arrête.
- *corps_de_la_boucle* peut être une seule ligne, ou bien un bloc entouré d'accolades. NB : comme pour les fonctions, toutes les variables définies dans la boucle, de même que la variable compteur sont **locales** à la boucle, et ne seront donc pas accessibles en dehors de celle-ci.

Exemple :

```
for(i in 1:10) cat(i,"\n")
```

La fonction `cat` écrit une liste d'objets dans la console. Ici, elle écrit le contenu de la variable *i* et revient à la ligne. Autre exemple :

```
v <- -5:5
for(i in v) {
  s <- sin(i*pi)
  cat("Sin(", i, "*pi) = ", s, "\n",sep="")
}
```

Il est bien sûr possible d'utiliser des boucles dans des fonctions et *vice versa*. on peut également *emboîter* plusieurs boucles :

```
for (i in 1:10) {
  cat("Table de ",i,":\n")
  for(j in 1:10) {
    cat(i,"*",j,"=",i*j,"\n")
  }
}
```

C.3 Suivi en temps réel

On s'intéresse ensuite au devenir de la population après plusieurs générations.

Exercice 20
- *Construire une fonction* `evolve` *qui prend en argument une population initiale, la fait évoluer pendant un nombre* **`n`** *de générations (passé en argument à la fonction) et renvoie le résultat.*
- *Faire évoluer la population pop pendant 3, 5, 10, etc générations. Que se passe-t-il ?*

Il est également intéressant de regarder l'évolution pas à pas :

Exercice 21
- *Créer une fonction* `evolve2` *qui fonctionne de manière similaire à* `evolve`*, mais renvoie un vecteur contenant la fréquence de résistants à chaque génération au lieu de renvoyer la population finale.*
- *Faire un graphe de la fréquence en fonction des générations pour 10 simulations et 20 générations.*

C.4 Évolution moyenne

Exercice 22
- *En utilisant une boucle* ***for****, créer une fonction* `get.freqs` *qui :*
- *fait évoluer 200 fois une population* **`population`** *pendant* **`nb.gen`** *générations,* **`population`** *et* **`nb.gen`** *étant des paramètres de la fonction,*

  - *calcule la fréquence des résistants dans chaque cas,*
  - *renvoie le vecteur des fréquences obtenues.*
- *appliquer la fonction* `get.freqs` *à la population pop, pendant 1, 2, 3, 10, 15, 30 et 100 générations, et tracer l'histogramme des fréquences dans chaque cas. Que se passe-t-il ?*

On procède maintenant à une approche plus quantitative. On étudie la moyenne et la variance des fréquences de résistants au cours du temps.

Exercice 23 *Tracer l'évolution de la moyenne et de la variance des fréquences de 200 populations évoluant à partir de pop en fonction de nombre de générations. On utilisera la fonction* `get.freqs` *définie précédemment, et on regardera les générations de 1 à 100, toutes les 5 générations. Conclusion ?*

C.5 Influence de la taille de la population

Exercice 24
- *Créer une fonction* `create.pop` *qui renvoie un vecteur population de taille* **`size`** *avec une proportion* **`freq`** *de résistants,* **`size`** *et* **`freq`** *étant des paramètres de la fonction.*
- *Créer une population de taille 100 avec une proportion 0.5 de résistants.*
- *Faire un suivi en temps réel d'une dizaine de simulations, et comparer les résultat avec ceux obtenus précédemment pour une population de taille 10.*

Créer des boucles conditionnelles dans R R permet également de faire d'autre types de boucles que les boucles **for**. Voici comment créer une boucle de type **while** :

`while(` *test* `)` *corps_de_la_boucle*

- *test* : tant que la condition est vraie, continue la boucle.
- *corps_de_la_boucle* : idem que pour la boucle **for**.

Exemple :

```
a <- 0
b <- 1
c <- a + b
while(c < 100) {
  cat(c,"\n")
  a <- b
  b <- c
  c <- a + b
}
```

(Suite de Fibonacci) Ici toutes les variables sont **globales**.

Exercice 25
- *Créer une fonction* `tps.fix` *qui fait évoluer une population jusqu'à la fixation d'un des deux allèles, puis renvoie le temps de fixation (nombre de générations écoulées jusqu'à la fixation).*
- *Étudier la distribution du temps de fixation pour une population de taille 10, une population de taille 100.*
- *Faites le graphe du temps moyen de fixation d'une population en fonction de sa taille (de 10 à 500, attention au nombre de points !).*
- *Ajuster un modèle permettant de prédire le temps moyen de fixation en fonction de la taille de la population.*

Annexe D

Représentation 3D et introduction à RGL : randonnée dans la vallée de l'Ubaye.

Ce chapitre supplémentaire est focalisé sur la manipulation de données 3D. Il comporte une introduction au paquetage `rgl`, qui permet d'effectuer des graphiques OpenGL dans R. Les exercices de ce chapitre ne sont pas difficiles, mais il est impératif d'avoir effectué tous les autres chapitres auparavant, y compris le premier chapitre supplémentaire (il y a un peu de programmation à faire !).

D.1 Introduction

Les jeux de données étudiées ici concernent le trajet d'une randonnée pédestre dans la vallée de l'Ubaye (Alpes du Sud), voir figure D.1.

Le fichier `SoleilBoeuf.csv` contient les coordonnées du trajet, dans trois colonnes "x", "y" et "z".

Exercice 26 *Mise en jambe.*
- *Calculez les altitudes minimales et maximales du parcours, ainsi que le dénivelé.*
- *Tracez le trajet tel qu'il apparaît sur la carte, i.e. seulement les x et les y.*

D.2 Calcul du profile

On va maintenant tracer le profile du parcours, *i.e.* l'altitude en fonction de la distance parcourue.

Exercice 27 *Distance parcourue.*
- *Les points 1 à 23 correspondent à la portion du trajet qui est effectuée à l'aller et au retour. Dupliquez cette portion afin qu'elle apparaissent aussi à la fin du trajet.*
- *Pour chaque point, calculez la distance parcourue depuis le début. On pourra stocker les résultats dans une nouvelle colonne nommée "d".*
- *Tracez le profile.*

Un autre calcul intéressant à faire est celui des pentes en chaque point.

Exercice 28 *Calcul des pentes.*
- *Pour chaque point, calculez la pente (NB : la pente est la dérivée du profile en un point). On pourra stocker le résultat dans une colonne nommée "dz".*
- *Représentez la pente en fonction de la distance.*
- *Juxtaposez le profile et sa dérivée sur un même graphe (Note : consultez l'annexe A).*

D.3 Représentation 3D

Le fichier `SoleilBoeuf_grid_z_.csv` contient une matrice avec des altitudes relevées selon une grille régulière de 20×20 points. Les fichiers `SoleilBoeuf_grid_x_.csv` et `SoleilBoeuf_grid_y_.csv` contiennent les coordonnées en abscisse et en ordonnée des points de la grille.

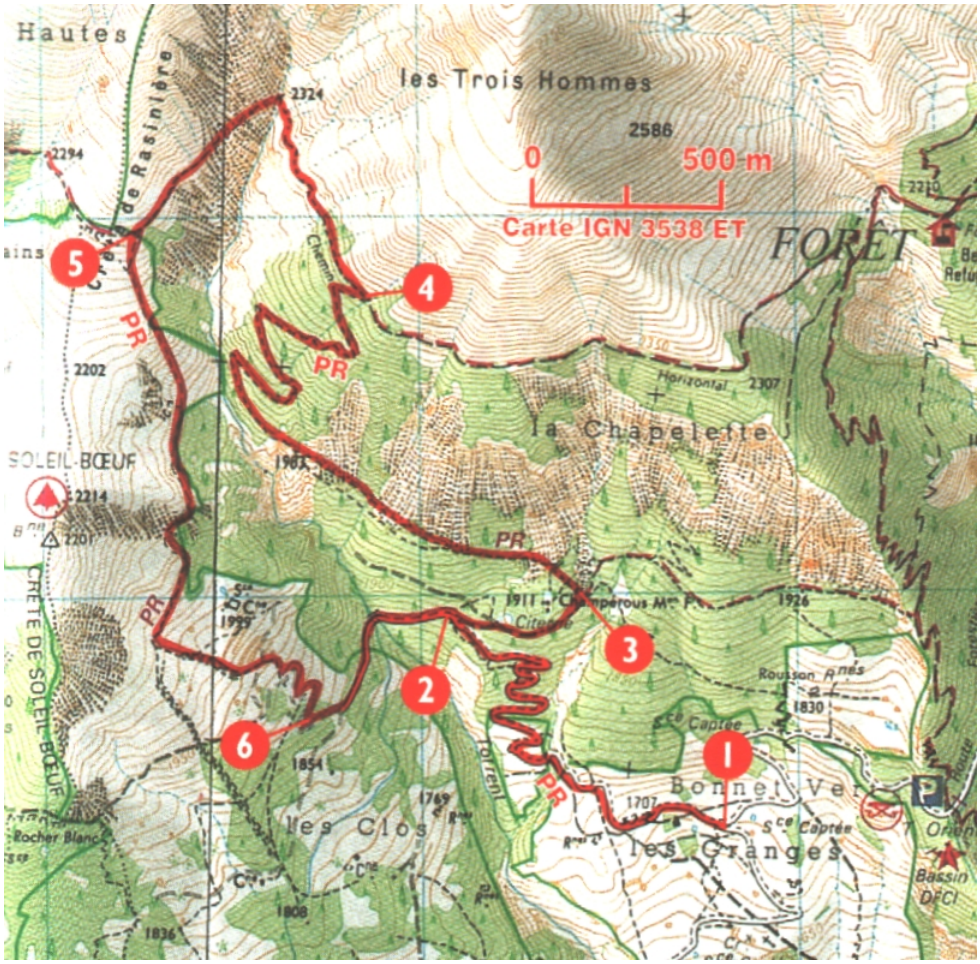


FIG. D.1 – Randonnée dans l'Ubaye : la crête de Soleil Boeuf (source : *La vallée de l'Ubaye... à pied*, Topo-Guide, FFRP).

Exercice 29 *Topographie.*

- *Utilisez la fonction* `filled.contour` *pour représenter la topographie des lieux. R possède plusieurs jeux de couleurs prédéfinis, dont un nommé* `terrain.colors` *pour les représentations topographiques. Se reporter à l'aide de cette fonction pour plus de détails.*
- *Représentez la topographie en lignes de niveau (cf.* `contour`*), et ajoutez le trajet sur la carte.*

D.4 Utilisation de RGL

La paquetage `rgl` permet de représenter des objets en 3 dimensions grâce à OpenGL. On le charge en tapant

```
library(rgl)
```

Ce paquetage n'est pas installé par défaut avec R, il faut donc l'installer en tapant[1]

```
install.packages("rgl")
```

D.4.1 Tracé du trajet

La fonction `rgl.points` trace des points, dont on passe les coordonnées en **`x`**, **`y`** et **`z`**. Attention cependant, les axes ne sont pas étiquetés de façon intuitive dans les fonctions `rgl` : l'axe y désigne les altitudes et z la profondeur. Celà a pour conséquence que le repère (x, y, z) n'est pas direct, et conduit donc à une représentation inversée selon l'axe y. pour

[1] Vous devez avoir les droits administrateur sur l'ordinateur pour effectuer cette commande.

obtenir la représentation souhaiter, il faut effectuer la correspondance suivante :

Axe :	Axe dans RDG :
x	**z**
y	**x**
z	**y**

La fonction `rgl.lines` est similaire à `rgl.points` et trace de segments. Attention, `rgl.lines` est l'équivalent de `segments` et non de `lines`. Si on veut tracer des lignes, il convient donc de dupliquer les coordonnées :

$$x = \{0, 1, 2, 3, 4\} \longrightarrow x2 = \{0, 1, 1, 2, 2, 3, 3, 4\}$$

Il existe aussi des fonctions permettant de tracer des triangles et des quadrilatères : `rgl.triangles` et `rgl.quads`

Finalement, la fonction `rgl.surface` permet de tracer des surfaces. Elle fonctionne comme la fonction `persp`.

Exercice 30 *Trajet 3D.*

- *Représentez le trajet en 3 dimensions.*
- *Utilisez les données topographiques pour planter le décors, en utilisant `rgl.surface`. Se reporter à la documentation de la fonction pour un exemple d'utilisation des couleurs.*
- *La fonction `rgl.bg` permet de changer le fond de l'image et d'ajouter du brouillard... expression libre.*

Annexe E

Mémento

E.1 Système

help, ?	Obtenir de l'aide
help.start	Aide HTML
library	Charge un paquetage
install.packages	Installer un paquetage depuis le serveur CRAN
update.packages	Actualise des paquetages
data	Charge un jeu de données en mémoire
ls	Liste le contenu des objets en mémoire
rm	Efface un objet en mémoire
setwd	Spécifie le répertoire courant
getwd	Récupère le répertoire courant
source	Lit un script et l'exécute

E.2 Manipulation de vecteurs/facteurs

<-	Affectation : `y<-a*x+b`
vector	Crée un nouveau vecteur
factor	Crée un nouveau facteur
ordered	Crée un nouveau facteur ordonné
c	Concatène : `y<-c(2,3,0,6,10)`
:	Série entière : `5:9` équivaut à `c(5,6,7,8,9)`
seq	Crée une séquence
rep	Crée un vecteur répétition d'un motif donné
length	Donne la longueur d'un vecteur
[]	Sous vecteur : `x[3]`, `x[3:8]`, `x[x > 0]`, `x[c(T,F)]`
+ - /*	Opérateurs mathématiques
unique	Renvoie un vecteur sans éléments répétés
as.vector	Convertit en vecteur
as.factor	Convertit en facteur
as.ordered	Convertit en facteur ordonné
is.vector	TRUE si vecteur
is.factor	TRUE si facteur
is.ordered	TRUE si facteur ordonné
cut	Discrétise un vecteur nuérique en facteur
split	Convertit un vecteur en facteur selon les niveaux donnés par un autre facteur
levels	Récupère les modalités d'un facteur (ordonné ou pas)
sum	Somme
cumsum	Somme cumulée
prod	Produit
cumprod	Produit cumulé

E.3 Statistiques descriptives

summary	Résumé des données
mean	Moyenne
var	Variance
sd	Ecart type
median	Médiane
cov	Covariance
cor	Correlation

E.4 Graphiques (paquettages graphics et lattice)

plot	xyplot	Graphe générique, $y = f(x)$
boxplot	bwplot	Boîtes à moustaches
coplot	xyplot	Graphes conditionnels (en fonction d'un attribut discret)
interaction.plot	—	Graphes d'interaction de deux facteurs
barplot	barchart	Graphiques en barres
dotchart	dotplot	Autres graphiques en barres
stripchart	stripplot	Similaire à boxplot pour un petit nombre de données
hist	histogram	Histogramme
legend	—	Ajoute une légende à un graphe
persp	wireframe	Graphe 3D : perspective
contour	contourplot	Graphe 3D : lignes de niveaux
filled.contour		Graphe 3D : lignes de niveaux + différentes couleurs par niveau
image	levelplot	Graphe 3D : différentes couleurs par niveau

E.5 Dessin :

lines	llines	Relie des points sur un graphe
points	lpoints	Trace des points sur un graphe
segments	lsegments	Relie des segments sur un graphe
arrows	larrows	Trace des flèches sur un graphe
text	ltext	Ecrit du texte sur un graphe
abline	—	Ajoute une ou plusieurs lignes droites à un graphe en spécifiant leur équation
curve	—	Trace une courbe sachant son équation
polygon	—	Ajoute un polygone à un graphe
rect	—	Ajoute un ou plusieurs rectangles à un graphe
box	—	Crée une boîte autour du graphique

E.6 Tests statistiques

cor.test	Test de correlation (Pearson, Spearman et Kendall)
chisq.test	Test de chi 2
friedman.test	Test des rangs de Friedman
t.test	Test t de Student
var.test	Test F de comparaison de variances
kruskal.test	Test de la somme des rangs de Kruskal-Wallis
ks.test	Test de Kolmogorov-Smirnov
wilcox.test	Tests de Wilcoxon (Mann-Whitney)

E.7 Formules

~	Sépare les variables "à expliquer" des variables "explicatives"
A+B	Effet de A + effet de B
A-B	Enlever un effet
A:B	Interaction
A*B	= A + B + A:B
A/B	= A + A:B
A|B	A conditionnellement à B
(A+B+C)^2	= A + B + C + A:B + A:C + B:C
I(A^2)	'insulate' : l'expression 'insularisée' a son sens mathématique
Error(C)	C est le terme d'erreur (aov seulement, modèle II)

E.8 Modèles

lm	Modèle linéaire
glm	Modèle linéaire généralisé
aov	Analyse de variance
lme	Modèle linéaire à effets mixtes
nlme	Modèle non-linéaire
nls	Moindre carrés non-linéaires
nlm	Maximum de vraisemblance
summary	Résumé du modèle
coefficients	Récupère les coefficients d'un modèle
fitted.values	Recupère les prédictions d'un modèle
residuals	Recupère les résidus d'un modèle
anova	Calcule une table d'ANOVA
predict	Prédit des valeurs à partir d'un modèle
step	Sélection de modèles par AIC
qqnorm	Graphe Quantile-Quantile normal (droite de Henry)
hmctest	Test de Harrison-Mac-Cabe d'homoscédasticité des résidus
bptest	Test de Breusch-Pagan d'homoscédasticité des résidus
dwtest	Test de Durbin-Watson d'indépendance des résidus

E.9 Matrices

matrix	Crée une nouvelle matrice
cbind	Lie des vecteurs en colonnes
rbind	Lie des vecteurs en lignes
[i]	Elément i dans la matrice (dans l'ordre de lecture)

`[i,j]`	Elément i de la colonne j de la matrice
`[,j]`	Vecteur colonne j de la matrice
`[i,]`	Vecteur ligne i de la matrice
`dim`	Dimensions de la matrice
`eigen`	Décomposition spectrale d'une matrice
`as.matrix`	Convertit en matrice
`is.matrix`	TRUE si matrice
`rownames`	Noms de lignes
`colnames`	Noms de colonnes

E.10 Jeux de données

`data.frame`	Crée un nouveau jeu de données
`read.table`	Crée un jeu de données à partir d'un fichier texte
`write.table`	Ecrit un jeu de données dans un fichier
`edit`	Edite un jeu de données
`attach`	Le jeu de données est attaché au chemin courant
`detach`	Enlève la base de donnée courante du chemin de recherche
`[]`	Sous-jeu de données
`[i,j]`	Elément i de la colonne j du jeu de données
`[,j]`	Vecteur colonne j du jeu de données
`[i,]`	Vecteur ligne i du jeu de données
`names`	Noms de colonnes
`rownames`	Noms de lignes

E.11 Objets

`$`	Champ
`list`	Crée un nouvel objet liste
`names`	Récupère les noms de champs présents dans un objet

E.12 Programmation

`function`	Crée une nouvelle fonction
`if(expr_1) expr_2 else expr_3`	Instruction conditionnelle
`for(name in expr_1) expr_2`	Boucle for
`repeat expr while (condition)`	Boucle repeat
`while (condition) expr`	Boucle while
`==, !=`	Opérateurs logiques d'égalité
`<, >, <=, >=`	Opérateurs logiques d'inégalité
`&`	'Et' logique
`|`	'Ou' logique

E.13 Quelques paquetages utiles

`lattice`	Autres fonctions graphiques
`nlme`	Modèles linéaires et non linéaires mixtes (effet aléatoire + fixes), moidres carrés généralisés.
`lmtest`	Tests pour modèles linéaires.
`car`	Companion to Applied Regression. Permet de faire des moindres carrés de type II et III.
`ade4`	Analyse de données environementales
`ape`	Phylogénie et Evolution (analyse comparative)
`rgl`	Utilisation d'OpenGL dans R

Bibliographie

[1] José C. Pinheiro and Douglas M. Bates. *Mixed-Effects Models in S and S-Plus*. Springer, 2000.

[2] Robert R. Sokal and F. James Rohlf. *Biometry*. W. H. Freeman and co., 3ème edition, 1995.

[3] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, 4ème edition, 2002.

Index