



Ifremer

Formation “Ifremer ”

Emmanuel Paradis



Institut de recherche
pour le développement

Sète, 2–6 février 2009

Programme

- I Introduction sur R et statistique
- II Structure et manipulation des données
- III Graphiques
- IV Modèles linéaires
- V Maximum de vraisemblance
- VI Modèles linéaires généralisés
- VII Modèles linéaires mixtes
- VIII Régression non-linéaire
- IX Modèles de “lissage”
- X Séries temporelles
- XI Analyses spatiales

I Introduction sur R et statistique

Des données aléatoires pour commencer (et se rappeler quelques concepts statistiques) :

```
> rnorm(10)
 [1]  0.7945464  1.2347852 -0.6727558 -0.1248683 -1.0731775
 [6]  0.8186927  1.2732294 -0.3175405 -1.0760822  0.5352055
> a <- rnorm(10)
> a
 [1] -1.2238241 -0.1101836 -0.7159118  0.2910358  0.7198640
 [6] -0.3708338  1.7233652 -0.8137307 -0.4111521 -0.5303543
> b <- rnorm(10)
> mean(a); mean(b)
 [1] -0.1441725
 [1]  0.3051921
```

Cette différence est-elle statistiquement significative ?

```
> t.test(a, b)
```

```
Welch Two Sample t-test
```

```
data: a and b
```

```
t = -1.0257, df = 17.084, p-value = 0.3193
```

```
alternative hypothesis: true difference in means is not equal to 0
```

```
95 percent confidence interval:
```

```
-1.3733489  0.4746197
```

```
sample estimates:
```

```
mean of x  mean of y
```

```
-0.1441725  0.3051921
```

Et si le test était unilatéral (*one-tailed*) ?

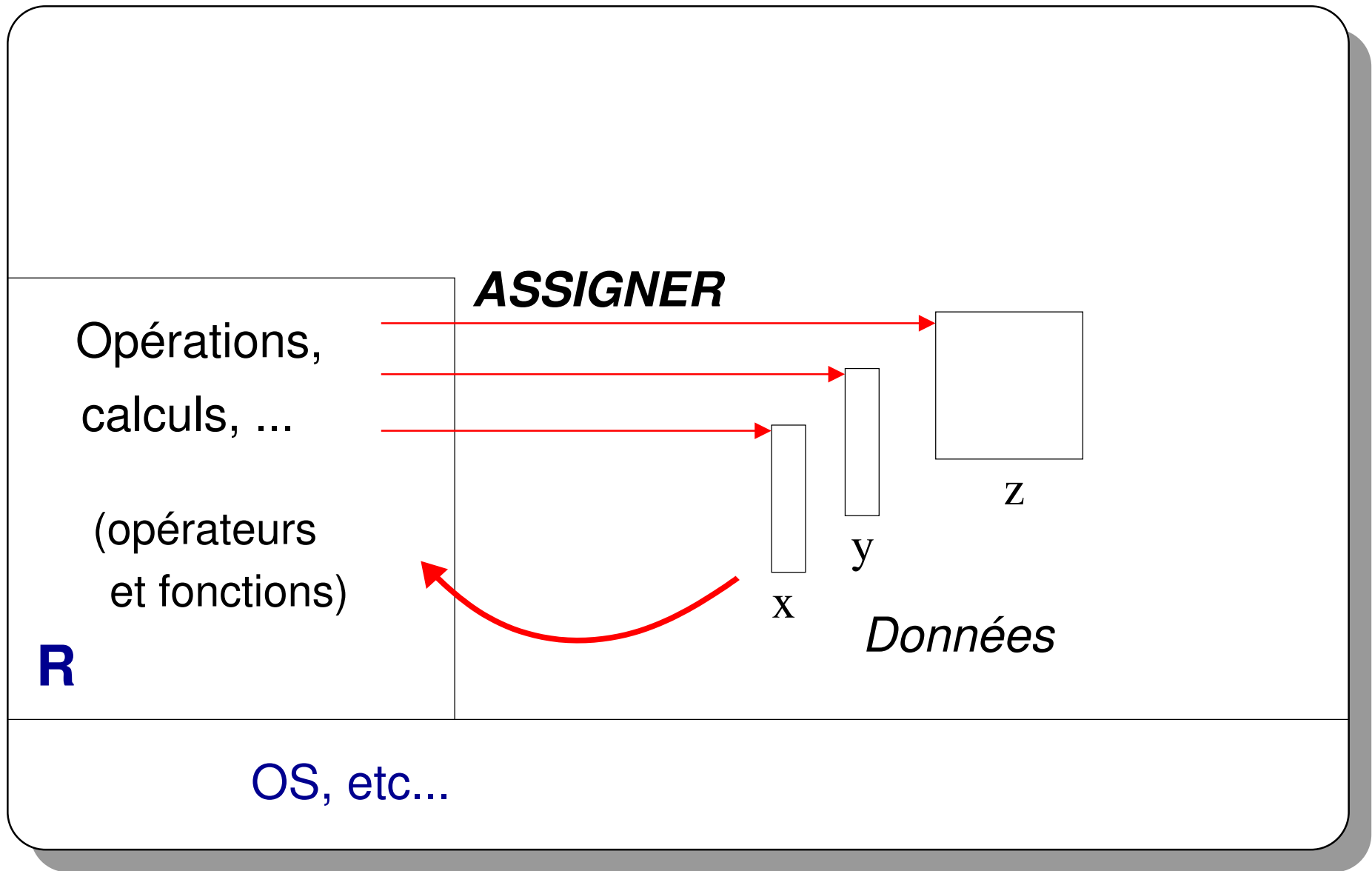
```
> t.test(a, b, alternative = "less")
```

```
Welch Two Sample t-test
```

```
data: a and b
t = -1.0257, df = 17.084, p-value = 0.1597
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
    -Inf 0.3125583
sample estimates:
 mean of x  mean of y
-0.1441725  0.3051921
```

Ce second test était-il nécessaire ?

```
> curve(dt(x, df = 17.084), from = -3, to = 3)
> abline(v = -1.0257, col = "blue")
```



Mémoire vive (RAM)

L'analyse statistique cherche à quantifier la variation des données en :

- variation systématique → **explicative**
- variation aléatoire → **résiduelle** (“nuisance”)

Dans l'exemple ci-dessus on cherche à expliquer la variation d'une variable normale en fonction de l'appartenance à un groupe :

```
> summary(aov(c(a, b) ~ gl(2, 10)))
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
gl(2, 10)	1	1.0096	1.0096	1.052	0.3186
Residuals	18	17.2746	0.9597		

Notre premier programme en R : évaluer le risque de première espèce (*type I error rate*) du test de *t*.

```
p <- numeric(1000)
for (i in 1:1000) {
  x <- rnorm(20)
  y <- rnorm(20)
  p[i] <- t.test(x, y)$p.value
}
hist(p)
sum(p < 0.05)
```

Exercice

1. Comment évaluer le risque de seconde espèce (*type II error rate*) ?

Corrigé

1. Les variables seront simulées avec des moyennes différentes, par exemple :

```
x <- rnorm(20)
```



```
y <- rnorm(20, 1)
```

Le risque de seconde espèce (= 1 – puissance du test) ne sera valable que pour cette différence, la taille d'échantillon et la variance utilisés. À signaler la fonction `power.t.test` qui permet de faire ces calculs de puissance directement.

Quels noms pour les objets ?

Un nom d'objet doit commencer par une lettre.

Les caractères autorisés sont : A-Z a-z 0-9 . _

Tous les noms sont permis mais "quotés".

```
> "a+a" <- 8
```

```
> a+a
```

```
Erreur : objet "a" non trouvé
```

```
> get("a+a")
```

```
[1] 8
```

De même pour les 15 mots réservés du langage : for, in, if, else, while, next, break, repeat, function, NULL, NA, NaN, Inf, TRUE **et** FALSE.

```
> Inf <- 1
```

```
Erreur dans Inf <- 1 : membre gauche de l'assignation (do_set)
```

```
> "Inf" <- 1
> ls()
[1] "a+a" "Inf"
> Inf
[1] Inf
> get("Inf")
[1] 1
```

T et F peuvent être utilisés.

L'usage des lettres accentuées dépend du "locale", donc à éviter.

Le nom d'un objet est limité à 256 caractères, alors n'hésitez pas à utiliser des noms explicites.

Les noms des fonctions peuvent être utilisés et cela pose généralement peu de problème (ex : df) :

```
> log <- 10.2
> log[1]
[1] 10.2
> log(1)
[1] 0
> log <- function(x) paste("x =", x)
> log(1)
[1] "x = 1"
> base::log(1)
[1] 0
```

Comparer deux fonctions de deux packages différents :

```
library(mgcv)
library(gam)
mgcv::gam(. . . .
gam::gam(. . . .
```

Gérer ses scripts de commandes

Un bon éditeur est utile dès qu'on débute avec R. Il permet notamment de :

1. colorer la syntaxe,
2. "allumer" les parenthèses, crochets et accolades,
3. ajouter et éditer des commentaires,
4. éventuellement envoyer des lignes ou des blocs de commandes directement vers R.

Le principal intérêt d'un script de commandes est de pouvoir répéter les analyses (intérêt pratique mais aussi fondamental). Un autre intérêt, mais non moins négligeable, est d'ajouter des commentaires.

Pour répéter des analyses à partir d'un script :

1. copier/coller vers la console ;
2. envoyer les commandes vers R (si possible) ;
3. `source("script_Aedes_morpho_Dakar.R")`
4. `R CMD BATCH script_Aedes_morpho_Dakar.R`

L'avantage de `R CMD BATCH` est que les commandes et les résultats sont dans le même fichier ('`script_Aedes_morpho_Dakar.Rout`').

Avec `source` ou `R CMD BATCH`, chaque ligne de commentaire doit être précédée par `#`. Une alternative, si le bloc est syntactiquement correct :

```
if (FALSE) {  
.....  
... bloc exclu de l'exécution  
.....  
}
```

Ajouter des commandes dans un script

1. copier/coller depuis la console ;
2. `savehistory("R_script_today.R")` sauvegarde toutes les commandes de la session en cours sur le disque. Si aucun nom de fichier n'est précisé, il sera nommé '`.Rhistory`' et sera peut-être caché (et pas forcément associé avec votre éditeur de scripts R).

 Il est vivement conseillé “d’aérer” les opérateurs :

```
x > -1           x>-1
x < -1           x<-1 # :(
```

Ajouter des messages dans un script

Le plus simple : `message("Debut des calculs...")`

Plus intéressant : `cat("n =", n, "\n")`

II Structure et manipulation des données

Vecteur

1
2
3
4
5

length (= 5)

mode (= "numeric")

"Homo"
"Pan"
"Gorilla"

length (= 3)

mode (= "character")

} **Attributs**

length : nombre d'éléments **mode** : numeric, character, logical (complex, raw)

Comment construire un vecteur ?

1. Séries régulières : ':' `seq(from, to, by)` `rep` :

```
rep(1:2, 10)
```

```
rep(1:2, each = 10)
```

```
rep(1:2, each = 2, length.out = 20)
```

2. Séries aléatoires : `rloi(n, ...)`

3. Vecteurs "par défaut" : `numeric(2)` `logical(10)` `character(5)`

4. Concaténer des vecteurs : `c(x, y)`, `c(x, y, z)`, ...

5. Entrer direct au clavier avec `scan()` (numérique) ou `scan(what = "")`

6. Lecture de fichiers

Quelque soit le mode, une valeur manquante est indiquée `NA` (*not available*) mais est stockée de façon appropriée.

```
> x <- c("NA", NA)
```

```
> x
```

```
[1] "NA" NA
```

```
> is.na(x)
[1] FALSE TRUE
```

Les valeurs numériques infinies sont indiquées `Inf` ; `NaN` signifie “not a number”.

```
> -5/0
[1] -Inf
> exp(-5/0)
[1] 0
> Inf + Inf
[1] Inf
> Inf - Inf
[1] NaN
```

Une matrice (***matrix***) est un vecteur arrangé de façon rectangulaire.

Comment construire une matrice ?

1. Avec la fonction `matrix(NA, nrow, ncol, byrow = FALSE)`
2. À partir d'un vecteur : `dim(x) <- c(nr, nc)`, si `length(x) == nr*nc!`
3. En joignant des vecteurs avec `rbind` ou `cbind`.

Facteur

Un facteur (**factor**) est un vecteur de mode numérique codant une variable qualitative (couleur, ...). L'attribut "levels" spécifie les noms des niveaux. Certains niveaux peuvent ne pas être présents.

Un facteur ordonné (**ordered**) a une hiérarchie dans ses niveaux (ex : TB, B, m, M, TM).

Comment construire un facteur ?

1. Séries régulières : `gl(n, k, n*k)` (*generate levels*)
2. Avec la fonction `factor(x, levels =)`
3. À partir d'un vecteur numérique `x` : `cut(x, breaks)` (*cf. ?cut pour les détails*)
4. Voir la fonction `stack` plus loin (à partir d'un tableau).
5. Lecture de fichiers



Les facteurs ne peuvent pas être concaténés avec `c(x, y)`

Tableau de données et liste

Un tableau de données (***data frame***) est un ensemble de vecteurs et/ou de facteurs tous de la même longueur.

Comment construire un tableau de données ?

1. Avec la fonction `data.frame`
2. Lecture de fichiers

Une liste (***list***) est un ensemble d'objets quelconques.

Comment construire une liste ?

1. Avec la fonction `list`
2. Lecture de fichiers (avec `scan`)

Le système d'indexation des vecteurs : []

Numérique : positif (extraire, modifier et/ou 'allonger') OU négatif (extraire uniquement).

Logique : le vecteur d'indices logiques est éventuellement recyclé (sans avertissement) pour extraire, modifier et/ou allonger.

Avec les noms (*names* = vecteur de mode character) ; pour extraire ou modifier.

Extraction et “subsetting” des matrices, tableaux et listes

1. `[,]` (les 3 systèmes) pour matrices et tableaux mais :
 - allongement impossible,
 - `drop = TRUE` par défaut.

Il n’y a pas de `names` mais `colnames` et/ou `rownames` (obligatoires pour les tableaux).

2. Extraction à partir d’un tableau ou d’une liste : `$` (avec noms) `[[` (numérique ou avec noms).
3. Subsetting à partir d’un tableau ou d’une liste : `[` (les 3 systèmes).
`subset` est une fonction qui permet de faire le même genre d’opération de sélection de lignes et/ou colonnes d’une matrice ou d’un tableau.

Les conversions

R a 118 fonctions `as.XXX` pour convertir les objets. Cela peut concerner le **mode** (`as.numeric`, `as.logical`, ...), le **type de données** (`as.data.frame`, `as.vector`, ...), ou même d'autres objets (`as.formula`, `as.expression`, ...).

 R effectue parfois des conversions implicites (***coercions***) :

```
> "0" == 0
[1] TRUE
> "0" == FALSE
[1] FALSE
> 0 == FALSE
[1] TRUE
```

Manipulation des vecteurs logiques

Les vecteurs logiques sont le plus souvent générés avec les opérateurs de comparaison : `==` `!=` `<` `>` `<=` `>=`.

L'opérateur `!` inverse les valeurs d'un vecteur logique (avec d'éventuelles coercions : `!0` `!1`).

L'opérateur `&` compare deux vecteurs logiques élément par élément et retourne un vecteur logique avec `TRUE` si les deux éléments le sont aussi, `FALSE` sinon.


L'opérateur `|` fait la même opération mais retourne `TRUE` si au moins un des éléments l'est aussi.

La fonction `xor` fait la même opération mais retourne `TRUE` si un seul des éléments l'est aussi.

R a 55 fonctions de la forme `is.XXX` (`is.numeric`, `is.logical`, `is.factor`, `is.na`, `is.data.frame`, ...)

Trois fonctions utiles pour manipuler les vecteurs logiques :

- `which` retourne les indices des valeurs TRUE (ex : `which(x == 0)`)
- `any` retourne TRUE s'il y a au moins une valeur TRUE
- `all` retourne TRUE si elles le sont toutes

 L'opérateur `==` n'est pas toujours approprié pour comparer des valeurs numériques (sensibilité à la précision numérique) : utiliser plutôt la fonction `all.equal`.

```
> 1.2 - 0.8 == 0.4  
[1] FALSE
```

Manipulation des facteurs

Un facteur est stocké sous forme d'entiers avec un attribut qui spécifie les noms des niveaux (*levels*) :

```
> f <- factor(c("a", "b"))
> f
[1] a b
Levels: a b
> str(f)
Factor w/ 2 levels "a","b": 1 2
```

Donc si on traite les facteurs comme des vecteurs ordinaires, on manipule les codes numériques.

```
> c(f) # == as.integer(f)
[1] 1 2
> as.vector(f) # == as.character(f)
```

```
[1] "a" "b"
```

Les niveaux peuvent être extraits ou modifiés avec la fonction `levels` :

```
> levels(f) <- c(levels(f), "c")
```

```
> f
```

```
[1] a b
```

```
Levels: a b c
```

```
> table(f)
```

```
f
```

```
a b c
```

```
1 1 0
```

```
> g <- factor(c("a", "b", "c", "d"))
```

```
> levels(f) <- levels(g)
```

```
> f
```

```
[1] a b
```

```
Levels: a b c d
```

Pour supprimer les niveaux absents :

```
> factor(f)
[1] a b
Levels: a b
```

Pour concaténer des facteurs :

```
> h <- factor(c(as.vector(f), as.vector(g)))
> h
[1] a b a b c d
Levels: a b c d
```

Note : l'indexation d'un facteur préserve les niveaux :

```
> h[1]
[1] a
Levels: a b c d
```

Exercice

1. Construire un vecteur `x` avec 30 valeurs selon $\mathcal{N}(0, 1)$. Sélectionner les éléments d'indices impairs. Trouver une solution qui marche quelque soit la longueur de `x`. Extraire les valeurs de `x` positives.
2. Créer un vecteur avec trois noms de taxons de votre choix. Afficher le mode de ce vecteur. Créer un vecteur numérique avec les tailles (approximatives) de ces taxons. Trouver comment extraire une taille avec un nom de taxon.
3. Créer un tableau de données avec trois observations (lignes) et deux variables numériques de votre choix. Extraire les noms des rangées de ce tableau avec `rownames`. Qu'observez-vous ? Modifier ces noms avec les noms de taxons choisis ci-dessus.
4. Extraire la première colonne de ce tableau avec `[` puis avec `[[`. Comparer les résultats.
5. Effacer cette première colonne.

Corrigé

1. `x <- rnorm(30)`. Il existe deux solutions, avec l'indexation numérique : `x[seq(1, length(x), 2)]`, ou l'indexation logique : `x[c(TRUE, FALSE)]`, ce qui met bien en exergue la recyclage des indices logiques et pas numériques. Pour extraire les valeurs positives : `x[x > 0]` (`x[x >= 0]` si l'on veut les valeurs positives ou nulles).
2.

```
> taxa <- c("Homo", "Pan", "Gorilla")
> taille <- c(70, 80, 100)
> names(taille) <- taxa
> taille["Pan"]
Pan
  80
```
3.

```
> DF <- data.frame(x = 1:3, y = rnorm(3))
> rownames(DF)
[1] "1" "2" "3"
> rownames(DF) <- taxa
```

4. `> DF[1]`

```
      x  
Homo  1  
Pan   2  
Gorilla 3
```

`> DF[[1]]`

```
[1] 1 2 3
```

`DF[1]` retourne un sous-ensemble du tableau, alors que `DF[[1]]` extrait un vecteur.

5. `DF[1] <- NULL`

Transformation de variables

Les fonctions suivantes transforment un vecteur numérique et retournent un vecteur de même longueur :

`sqrt abs sign log exp [a]sin[h] [a]cos[h] [a]tan[h]`

`sign(x) == x/abs(x)`

Pas d'option sauf `log(x, base = exp(1))` ; `log10(x)` est un raccourci pour `log(x, base = 10)`.

Deux opérateurs spéciaux :

`x %% y` : x modulo y

`x %/% y` : combien de fois *entière* y dans x

Des fonctions spécialisées dont (liste exhaustive dans `?Special`) :

gamma (x)	$\Gamma(x)$
choose (n, k)	C_n^k (souvent noté $\binom{k}{n}$)
factorial (x)	$\prod_{i=1}^x i = \Gamma(x + 1) \equiv x!$ pour x entier ; équivalent à <code>prod(1:x)</code>
lfactorial (x)	$\sum_{i=1}^x \ln i$; équivalent à <code>sum(log(1:x))</code> mais comparer :

```
> lfactorial(1e10)
```

```
[1] 220258509312
```

```
> sum(log(1:1e10))
```

```
Erreur dans 1:1e+10 : le résultat serait un vecteur trop long
```


Exercice

1. On sait que $\sqrt[n]{x} = x^{1/n}$. Comment tester, pour $n = 2$, si R retourne le même résultat pour ces deux opérations ?

Corrigé

```
1. > x <- runif(1e6, 1, 1000)
   > all.equal(sqrt(x), x^0.5)
   [1] TRUE
```

Arrondis

 Il est important de distinguer la précision d'un nombre tel qu'il est stocké en mémoire (et utilisé dans les calculs) et la précision affichée à l'écran (souvent tronquée).

```
> pi
[1] 3.141593
> print(pi, digits = 1)
[1] 3
> print(pi, digits = 16)
[1] 3.141592653589793
```

`ceiling(x)` : arrondit à l'entier supérieur ou égal

`floor(x)` : arrondit à l'entier inférieur ou égal

`trunc(x)` : supprime les décimales ; identique à `floor(x)` si $x \geq 0$

`round(x, digits = 0)` : **arrondit à digits décimales**

`signif(x, digits = 6)` : **arrondit à digits chiffres ; comparer :**

```
> print(round(pi*1000, 6), digits = 18)
```

```
[1] 3141.592654
```

```
> print(signif(pi*1000, 6), digits = 18)
```

```
[1] 3141.59
```

```
> # mais:
```

```
> print(round(pi/10, 6), digits = 18)
```

```
[1] 0.314159
```

```
> print(signif(pi/10, 6), digits = 18)
```

```
[1] 0.314159
```

`zapsmall(x, digits = getOption("digits"))` : **appelle** `round(x, 7 - log10(max(x)))`

Tri de variables

`rev(x)` inverse :

- les éléments de `x` si c'est un vecteur ou une matrice (qui sera convertie en vecteur)
- les colonnes de `x` si c'est un tableau
- les éléments de `x` si c'est une liste

`sort(x)` trie les éléments de `x` et retourne le vecteur trié. L'option `decreasing = TRUE` permet de trier dans le sens décroissant (plus efficace que `rev(sort(x))`).

`order` réalise un tri multiple "hiérarchisé" sur un ensemble de vecteurs : un tri est fait sur le 1er vecteur ; s'il y a des égalités de rang elles sont résolues avec le 2nd vecteur, et ainsi de suite. Cette fonction retourne les indices ainsi triés.

```
> order(c(1, 2, 1, 2))
```

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

```
> order(c(1, 2, 1, 2), 4:1)
```

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

Il est possible de mélanger les modes :

```
> order(c(1, 2, 1, 2), c("b", "b", "a", "a"))
[1] 3 1 4 2
```

```
x <- 10:1
sort(x)
order(x)
all(sort(x) == x[order(x)])
```

Exemple de tri des lignes d'un tableau :

```
o <- order(df$x, df$y, df$z)
df[o, ]
```

`order` a aussi l'option `decreasing = FALSE` (par défaut).

Avec `sort` ou `order` il n'y a pas d'ex-aequo (*tie*).

`rank` retourne les rangs d'un vecteur ; il y a plusieurs méthodes pour résoudre les *ties* (*cf.* `?rank` pour les détails).

```
> x <- rep(1, 4)
> rank(x)
[1] 2.5 2.5 2.5 2.5
> order(x)
[1] 1 2 3 4
```

Résumé et valeurs manquantes

Nous avons vu jusqu'ici des fonctions qui agissent sur chaque élément d'un vecteur.

```
> x <- c(1, NA)
```

```
> log(x)
```

```
[1] 0 NA
```

Mais quelle est le résultat de l'addition de 1 avec une valeur manquante ?

```
> sum(x)
```

```
[1] NA
```

`sum`, comme de nombreuses fonctions du même type, a une option `na.rm = TRUE`.

```
> sum(x, na.rm = TRUE)
```

```
[1] 1
```


mean, var, median, quantile, max, min, range, prod

cumsum et cumprod n'ont pas cette option mais, logiquement, retourne NA dès qu'une valeur manquante est rencontrée.

summary affiche, pour les vecteurs numériques, un résumé (minimum, maximum, quartiles, moyenne) avec le nombre de valeurs manquantes.

Si l'objet est une matrice ou un tableau, le résumé est fait pour chaque colonne.

Lecture/écriture de fichiers

`read.table` lit des données sous forme tabulaire depuis un fichier texte et retourne un tableau (beaucoup d'options).

`scan` lit tout type de données dans un fichier texte.

`write.table` écrit un tableau sous forme tabulaire dans un fichier.

`write` écrit un vecteur simple.

`cat` écrit des chaînes de caractères.

Le package `foreign` pour lire divers formats de fichier.

... et des packages spécialisés sur CRAN.

III Graphiques

Petit rappel historique :

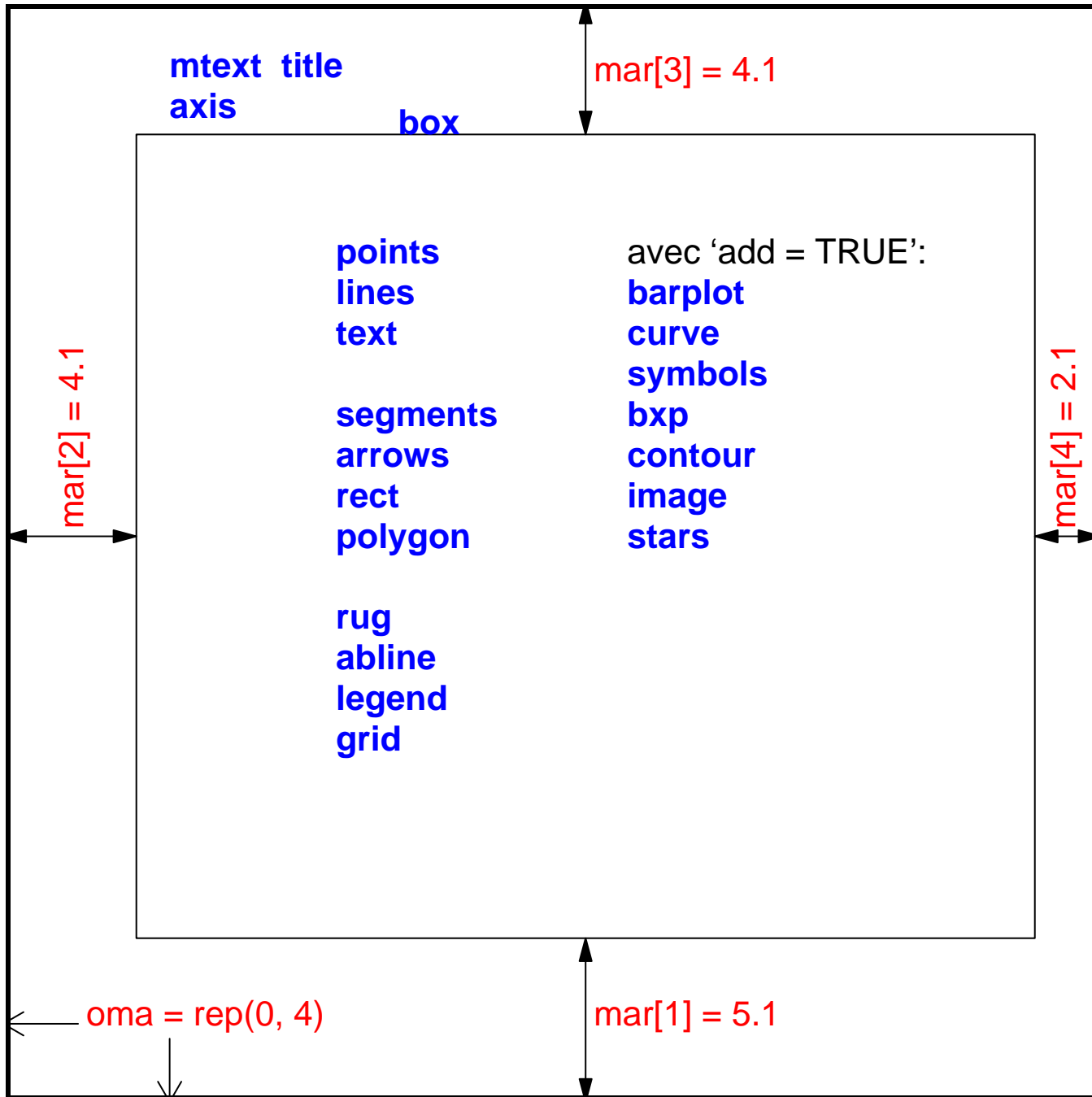
```
> ?anscombe
```

```
> example(anscombe)
```

L'analyse graphique est ***fondamentale*** dans le traitement des données.

Pour les fonctions graphiques primaires : *cf.* “R pour les débutants”.

Vue d'ensemble des fonctions graphiques secondaires :



Représenter les points avec des noms :

```
plot(x, y, "n")  
text(x, y, as.character(g))
```

Application typique :

```
plot(DF$V1, DF$V2, "n")  
text(DF$V1, DF$V2, rownames(DF))
```

Exemple : polygon

```
> args(polygon)  
function (x, y = NULL, density = NULL, angle = 45,  
         border = NULL, col = NA, lty = par("lty"), ...)  
> plot(rnorm(100))  
> a <- c(1, 50, 100, 50)  
> b <- c(0, -2, 0, 2)  
> polygon(a, b)
```

```
> polygon(a, b, col = "white")  
> polygon(a, b, col = "yellow", border = NA) # border="yellow")
```

`mtext` (*marginal text*) :

```
> par(oma = rep(1, 4))  
> plot(1)  
> mtext("Texte marginal")  
> mtext("Texte marginal \"externe\"", outer = TRUE)
```

Le second appel à `mtext` n'a aucun effet si `oma` a ses valeurs par défaut.

oma (*outer margin*) est utile en conjonction avec layout :

```
> par(oma = c(3, 3, 0, 0))
> layout(matrix(1:4, 2, 2))
> layout.show()
> par(mar = rep(2, 4))
> for (i in 1:4) plot(runif(50), xlab="", ylab="")
> mtext("index", 1, outer = TRUE, line = 1)
> mtext("runif(50)", 2, outer = TRUE, line = 1)
```

Le paramètre `mar` peut-être fixé individuellement pour chaque graphe.

Produire une zone de « dessin » 100×100 :

```
par(mar = rep(0, 4))
plot(0, type = "n", xlim = c(0, 100), ylim = c(0, 100),
     xlab = "", ylab = "", xaxt = "n", yaxt = "n", xaxs = "i",
     yaxs = "i", bty = "n")
```

« **clipping** » avec `par(xpd = TRUE)` :

```
> x11()  
> plot(0)  
> points(0.8, 1.25)  
> par(xpd = TRUE)  
> points(0.8, 1.25)
```

Utile pour placer une légende.

Un script typique avec iris :

```
## png("iris.png") # pour page Web
## postscript("iris.eps", width = 8, height = 6)
x <- iris[, 1]; y <- iris[, 2]; g <- iris$Species
co <- c("blue", "red", "yellow")
## co <- rep("black", 3)
psym <- c(19, 19, 19)
## psym <- c(1, 2, 3)
par(bg = "lightslategrey")
plot(x, y, col = co[g], pch = psym[g], xlab = "Sepal.Length",
      ylab = "Sepal.Width")
par(xpd = TRUE)
legend(5, 4.7, levels(g), pch = psym, col = co,
      bty = "n", horiz = TRUE)
## dev.copy2eps(file = "iris.eps") # respecte dim du "device"
## dev.off()
```

... plus sophistiqué :

.....

```
COLOR <- TRUE # FALSE (ou 1 # 0)
if (COLOR) {
  par(bg = "lightslategrey")
  co <- c("blue", "red", "yellow")
  psym <- c(19, 19, 19)
} else {
  par(bg = "transparent") # au cas où...
  co <- rep("black", 3)
  psym <- c(1, 2, 3)
}
```

.....

plotrix (par Jim Lemon) est un package spécialisé dans les fonctions graphiques de haut niveau (avec de nombreux exemples) :

<code>axis.break</code>	<code>axis.mult</code>
<code>barp</code>	<code>color2D.matplot</code>
<code>count.overplot</code>	<code>gantt.chart</code>
<code>pie3D</code>	<code>plotCI</code>
<code>polar.plot</code>	<code>radial.plot</code>
<code>triax.plot</code>	

La plupart des packages spécialisés fournissent des fonctions graphiques appropriées (`ade4`, `ape`, `maps`, ...) utilisant notamment la fonction générique `plot`.

Lattice et Grid

```
> library(lattice)
> xyplot(y ~ x | g)
> xyplot(Sepal.Width ~ Sepal.Length | Species, data = iris)
> histogram(~ Sepal.Length | Species, data = iris)

> m <- sample(1:5, size = 1e3, replace = TRUE)
> v <- sample(1:5, size = 1e3, replace = TRUE)
> x <- rnorm(1e3, m, v)
> table(m, v)
> hist(x) # != histogram(~x)
> histogram(~x | m * v)
> histogram(~x | v * m)
> m <- factor(m, labels = paste("mean =", 1:5))
> v <- factor(v, labels = paste("var =", 1:5))
> histogram(~x | m * v)
```

Principales caractéristiques de lattice :

- les fonctions graphiques « traditionnelles » ne peuvent pas être utilisées
- l'argument principal est une formule
- il est possible d'utiliser un argument `data` pour localiser les variables
- les objets graphiques sont modifiables

☺ Les graphes conditionnés sont des outils puissants pour l'exploration des données.

Les fonctionnalités de lattice sont vastes : histogrammes, graphes 3D, ...

☹ Les fonctions ayant de nombreuses options il est difficile de s'y retrouver.

Les paramètres graphiques sont difficiles à trouver et à modifier.

Les fonctions `panel*` sont difficiles à utiliser, bien que très puissantes.

```
> trellis.par.get()
> show.settings()
> trellis.par.set(list(strip.background=list(alpha=1,
+                               col="grey")))
```

Les fonctions graphiques secondaires sont dans le package grid :

```
> library(grid)
> apropos("^grid\\.")
```

Pour démarrer, il est préférable d'utiliser `lattice` avec les paramètres par défaut. De plus le package est encore à un stade de développement (ver. 0.17 avec R 2.8.1).

IV Modèles linéaires

Le modèle : $E(y) = \beta_1 x_1 + \beta_2 x_2 + \dots + \alpha$

y : réponse

x_1, x_2, \dots : prédicteurs

$\beta_1, \beta_2, \dots, \alpha$: paramètres

Les observations : $y_i = \beta_1 x_{1i} + \beta_2 x_{2i} + \dots + \alpha + \epsilon_i$

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

Une régression « simple » :

```
> x <- 1:50  
> y <- x + 3 + rnorm(50, 0, 10)  
> mod <- lm(y ~ x)  
> summary(mod)
```

Call:

```
lm(formula = y ~ x)
```

Residuals:

Min	1Q	Median	3Q	Max
-17.1684	-4.6348	0.8496	6.7277	16.4951

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	3.81285	2.46452	1.547	0.128
x	0.93191	0.08411	11.079	7.95e-15

Residual standard error: 8.583 on 48 degrees of freedom

Multiple R-Squared: 0.7189, Adjusted R-squared: 0.713

F-statistic: 122.8 on 1 and 48 DF, p-value: 7.952e-15

```
> plot(x, y)
```

```
> abline(mod)
```

```
> abline(b = 1, a = 3, lty = 2)
```



```
> legend("topleft", legend = c("Modèle estimé",  
+ "Modèle simulé"), lty = 1:2)
```

Les variables qualitatives sont codées avec les **contrastes**.

Cas avec deux classes : $z = \{R, B\}$, z est substituée par x_z :

$$z = R \rightarrow x_z = 0 \quad z = B \rightarrow x_z = 1$$

Cas avec trois classes : $z = \{R, B, V\}$, z est substituée par x_{z_1} et x_{z_2} :

	x_{z_1}	x_{z_2}
R	0	0
B	1	0
V	0	1

Pour une variable avec n catégories, $n - 1$ variables 0/1 sont créées ; il y a donc $n - 1$ paramètres supplémentaires associés à l'effet de cette variable.

Une analyse de variance à un facteur à trois niveaux :

```
> yb <- rnorm(60, 1:3)
> z <- gl(3, 1, 60)
> mod.b <- lm(yb ~ z)
> summary(mod.b)
```

Call:

```
lm(formula = yb ~ z)
```

Residuals:

Min	1Q	Median	3Q	Max
-2.00118	-0.52484	0.07318	0.78672	1.93827

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.3646	0.2210	6.174	7.44e-08
z2	0.4141	0.3126	1.325	0.190
z3	2.1182	0.3126	6.777	7.48e-09

Residual standard error: 0.9884 on 57 degrees of freedom
Multiple R-Squared: 0.4752, Adjusted R-squared: 0.4567
F-statistic: 25.8 on 2 and 57 DF, p-value: 1.049e-08

```
> plot(z, yb)
> plot.default(z, yb)
> library(lattice)
> histogram(~ yb | z, layout = c(1, 3))
```

Exercice

1. Refaire l'ajustement de `mod.b` mais en remplaçant `z` par `zb <- rep(1:3, 20)`. Comment faire pour obtenir les mêmes résultats ?

Corrigé

1. `zb` est un vecteur numérique et sera donc traité comme un prédicteur continu.
Pour obtenir le même ajustement : `lm(yb ~ factor(zb))`.
-

Note : il y a plusieurs type de contrastes :

```
> apropos ("^contr\\.")  
[1] "contr.SAS"          "contr.helmert"    "contr.poly"  
[4] "contr.sum"         "contr.treatment"
```

Les tests statistiques (des effets) ne sont pas affectés par le choix du type de contrastes, mais les paramètres estimés seront différents :

```
> contr.treatment(g1(3, 1)) # défaut dans R, pas dans S-PLUS
  2 3
1 0 0
2 1 0
3 0 1
> contr.SAS(g1(3, 1))
  1 2
1 1 0
2 0 1
3 0 0
```

Cela peut avoir des conséquences pour comparer les résultats de différents logiciels.

Formulation générale des modèles linéaires

$E(y) = \beta x$	Régression linéaire
$E(y) = \beta z$	Analyse de variance (ANOVA)
$E(y) = \beta_1 x + \beta_2 z$	Analyse de covariance (ANCOVA)

Dans tous les cas les erreurs sont normalement distribuées autour de la moyenne : ces modèles sont ajustés par la méthode des moindres carrés (minimiser $\sum (y_i - \hat{y}_i)^2$).

Dans R, `aov` et `lm` produisent le même ajustement ; la différence est dans l’affichage des résultats.

```
> summary(aov(yb ~ z))
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
z	2	50.412	25.206	25.802	1.049e-08
Residuals	57	55.683	0.977		

Pour la syntaxe des formules : “R pour les débutants”, p. 62.

Les interactions

Pour coder une interaction, de nouvelles variables sont créées par le produit des variables ou de leurs codages numériques. Si une variable qualitative a plus de deux niveaux, de nouvelles variables sont créées avec le produit de toutes les combinaisons 2 à 2 possibles entre les deux variables.

Pour deux variables avec n_1 et n_2 catégories, respectivement, $(n_1 - 1)(n_2 - 1)$ nouvelles variables 0/1 sont créées.

Pour les interactions d'ordre supérieur (entre trois variables ou plus), les combinaisons 3 à 3, 4 à 4, etc, sont utilisées.

`model.matrix` permet de visualiser le modèle numérique créé par une formule (notez la première colonne de 1) :

```
> model.matrix(yb ~ z)
> data.frame(model.matrix(yb ~ z), z)
```

Les diagnostics de régression

```
> plot(y, fitted(mod))
> abline(a = 0, b = 1, lty = 3)
> hist(residuals(mod))
> par(mfcol = c(2, 2))
> plot(mod)
```

1. Valeurs prédites par le modèle \hat{y}_i (en x) et résidus r_i (en y); idem que `plot(fitted(mod), residuals(mod))`.
2. Valeurs prédites (en x) et racine carrée des résidus standardisés, pour la $i^{\text{ème}}$ observation :

$$e_i = r_i / \left(\hat{\sigma} \sqrt{1 - h_{ii}} \right)$$

Les résidus r_i ne sont en fait pas indépendants et de variance homogène. La matrice de variance-covariance H est calculée avec $H = X(X^T X)^{-1} X^T$ (dans R : `H <- x %*% solve(t(x) %*% x) %*% t(x)`), en prenant

éventuellement `x <- cbind(1, x)`, ou `x <- model.matrix(mod)`; les h_{ii} sont les éléments sur la diagonale de cette matrice (`H[i, i]`) qui peuvent être aussi calculés par `hatvalues(mod)`.

3. Vu que $e_i \sim \mathcal{N}(0, 1)$, le graphe des valeurs de distribution prédites par cette loi et celles observées doit donner $x = y$.
4. *leverage* = h_{ii} , mesure de l'influence (effet de levier) de chaque observation sur la régression.

Pour apprécier la différence entre résidus et influence :

```
> mod.new <- lm(c(y, 75) ~ c(x, 100))  
> plot(mod.new)
```

Les tests d'hypothèse

```
> LIFEHIST <- read.delim("Mammal_lifehistories.txt")
> m1 <- lm(litter.size ~ mass.g., data = LIFEHIST)
> summary(m1)
```

Call:

```
lm(formula = litter.size ~ mass.g., data = LIFEHIST)
```

Residuals:

Min	1Q	Median	3Q	Max
-1.8053	-1.7253	-0.3053	1.1947	11.3747

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	2.805e+00	4.949e-02	56.68	< 2e-16
mass.g.	-2.605e-08	9.238e-09	-2.82	0.00488

Residual standard error: 1.769 on 1284 degrees of freedom

(154 observations deleted due to missingness)
Multiple R-Squared: 0.006156, Adjusted R-squared: 0.005382
F-statistic: 7.953 on 1 and 1284 DF, p-value: 0.004875

```
> m2 <- lm(litter.size ~ order, data = LIFEHIST)
> # m2 <- update(m1, formula = litter.size ~ order)
> summary(m2)
```

Call:

```
lm(formula = litter.size ~ order, data = LIFEHIST)
```

Residuals:

Min	1Q	Median	3Q	Max
-3.0338	-0.7268	-0.1844	0.5615	10.5390

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.27192	0.11261	11.295	<2e-16
orderCarnivora	1.49249	0.15108	9.879	<2e-16

```
orderCetacea      -0.25953      0.23598     -1.100      0.2716
[...]
```

```
Residual standard error: 1.406 on 1339 degrees of freedom
(84 observations deleted due to missingness)
```

```
Multiple R-Squared: 0.3791,      Adjusted R-squared: 0.3716
```

```
F-statistic: 51.09 on 16 and 1339 DF,  p-value: < 2.2e-16
```

```
> anova(m2) # == summary(aov(litter.size ~ order ...
```

```
Analysis of Variance Table
```

```
Response: litter.size
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
order	16	1616.98	101.06	51.087	< 2.2e-16
Residuals	1339	2648.85	1.98		

```
> m3 <- lm(litter.size ~ mass.g.*order, data = LIFEHIST)
```

```
> summary(m3)
```

```
Call:
```

```
lm(formula = litter.size ~ mass.g. * order, data = LIFEHIST)
```

```
Residuals:
```

```
      Min       1Q   Median       3Q      Max
-3.1418 -0.6929 -0.1807  0.5588 10.4874
```

```
Coefficients: (2 not defined because of singularities)
```

```
[...]
```

```
Residual standard error: 1.381 on 1254 degrees of freedom
(154 observations deleted due to missingness)
```

```
Multiple R-Squared: 0.4086,      Adjusted R-squared: 0.394
```

```
F-statistic: 27.95 on 31 and 1254 DF,  p-value: < 2.2e-16
```

```
> anova(m3)
```

```
Analysis of Variance Table
```

```
Response: litter.size
```

```
      Df  Sum Sq Mean Sq F value    Pr(>F)
```

```

mass.g.          1    24.90    24.90  13.0521  0.0003149
order           16  1548.57    96.79  50.7338 < 2.2e-16
mass.g.:order   14    79.29     5.66   2.9688  0.0001725
Residuals      1254  2392.27     1.91

```

```
> anova(lm(litter.size ~ order*mass.g., data = LIFEHIST))
```

Analysis of Variance Table

Response: litter.size

```

              Df  Sum Sq Mean Sq F value    Pr(>F)
order         16 1573.46   98.34  51.5493 < 2.2e-16
mass.g.        1    0.01    0.01  0.0044  0.9470630
order:mass.g. 14    79.29    5.66   2.9688  0.0001725
Residuals    1254  2392.27    1.91

```

```
> drop1(m3, test = "F")
```

Single term deletions

Model:

```
litter.size ~ mass.g. * order
```

```

              Df Sum of Sq      RSS      AIC F value    Pr(F)

```

```

<none>                2392.27   862.23
mass.g.:order 14      79.29 2471.56   876.16   2.9688 0.0001725
> drop1(lm(litter.size~order+mass.g., data=LIFEHIST), test="F")
Single term deletions

```

Model:

```
litter.size ~ order + mass.g.
```

	Df	Sum of Sq	RSS	AIC	F value	Pr(F)
<none>			2471.6	876.2		
order	16	1548.6	4020.1	1469.8	49.6544	<2e-16
mass.g.	1	0.008413	2471.6	874.2	0.0043	0.9476

```
> library(lattice)
```

```
> xyplot(litter.size ~ log(mass.g.) | order, data = LIFEHIST)
```

Les modèles ne peuvent être comparés que s'ils ont été ajustés au même vecteur de réponses :

- $y \sim x$ et $\log(y) \sim x$ ne peuvent **pas** être comparés !
- $y \sim x$ et $y \sim x + z$ ne seront pas ajustés aux mêmes données si x et z ont des NA à des observations différentes.

```
res.lm <- lm(...  
res.aov <- aov(...
```

1. `summary(res.aov)` : tableau d'ANOVA (= tests sur les effets $\sim F$)
`summary(res.lm)` : tests sur paramètres ($\sim t$)

2. `anova`

Si un modèle : tableau d'ANOVA en incluant les effets dans l'ordre de la formule.

Si plusieurs modèles : tableau d'ANOVA entre les modèles.

- (a) `anova(res.lm)` et `summary(res.aov)` sont identiques.
- (b) L'ordre des termes dans la formule est important s'il y a plusieurs prédicteurs qualitatifs et que les effectifs retournés par `table` sont inégaux (*unbalanced design*).

3. `drop1` : teste les effets individuels vs. le modèle complet.

(a) Le principe de marginalité est respecté.

(b) `drop1(res.lm)` et `summary(res.lm)` sont identiques si tous les prédicteurs sont continus ou avec deux catégories (car chaque effet a 1 ddl) et qu'il n'y a pas d'interaction dans le modèle.

4. `add1` teste l'ajout d'un ou plusieurs effets.

Ex. : si le modèle initial n'inclut pas d'interaction : `add1(res, ~.^2)` testera l'addition de chaque interaction individuellement.

5. `predict` calcule les valeurs prédites par le modèle. L'option `newdata` sert à spécifier de nouvelles valeurs pour les prédicteurs ; ceux-ci doivent être nommés de la même façon que dans la formule :

```
> predict(m1, newdata = 1e9)
```

```
Error in eval(predvars, data, env) :
```

```
not that many frames on the stack
```


```
> predict(m1, newdata = data.frame(mass.g. = 1e9))
```

```
[1] -23.24696
```

```
> ndf <- data.frame(mass.g. = 1e9)
```

```
> predict(m1, newdata = ndf)
[1] -23.24696
```

anova, drop1, add1 et predict sont des fonctions génériques.

 Les 'méthodes' sont documentées séparément : ?anova donne peu d'information ; c'est ?anova.lm (ou ?anova.glm, ...) qu'il faut généralement consulter.

Mais où sont les “type III SS” dans R ?

“Nowhere, it seems to me, is this *SAS coup d'état* more evident than in the way Analysis of Variance concepts are handled” — W.N. Venables

Les “type III SS” testent une hypothèse qui n’a, généralement, pas de sens.



Il ne peut pas y avoir d’interaction sans effets principaux, puisque, par définition, l’interaction implique un “contraste”. C’est le ***principe de marginalité***.

$y \sim x * A$ avec $A = \{A_1, A_2\}$

$$E[y] = \beta_1 x + \beta_2 x_A + \beta_3 x x_A + \alpha$$

$$A_1 \rightarrow E[y] = \beta_1 x + \alpha$$

$$A_2 \rightarrow E[y] = (\beta_1 + \beta_3)x + \beta_2 + \alpha$$

$H_0 : \beta_1 = 0$ (hypothèse testée par les “type III SS”)

$$A_1 \rightarrow E[y] = \alpha$$

$$A_2 \rightarrow E[y] = \beta_3 x + \beta_2 + \alpha$$

$$y \sim A * B \quad A = \{A_1, A_2\}, B = \{B_1, B_2\}$$

$$E[y] = \beta_1 x_A + \beta_2 x_B + \beta_3 x_A x_B + \alpha$$

$$A_1, B_1 \rightarrow E[y] = \alpha$$

$$A_1, B_2 \rightarrow E[y] = \beta_2 + \alpha$$

$$A_2, B_1 \rightarrow E[y] = \beta_1 + \alpha$$

$$A_2, B_2 \rightarrow E[y] = \beta_1 + \beta_2 + \beta_3 + \alpha$$

$$H_0 : \beta_1 = 0$$

$$A_1, B_1 \rightarrow E[y] = \alpha$$

$$A_1, B_2 \rightarrow E[y] = \beta_2 + \alpha$$

$$A_2, B_1 \rightarrow E[y] = \alpha$$

$$A_2, B_2 \rightarrow E[y] = \beta_2 + \beta_3 + \alpha$$

```
drop1(mod, test = "F")
drop1(mod, .~., test = "F") # type III SS
```

`model.matrix` peut être utile pour “vérifier” un modèle.

```
> x <- 1:10
> z <- gl(2, 5)
> y <- rnorm(10)
> drop1(lm(y ~ x*z), test = "F")
```

Single term deletions

Model:

```
y ~ x * z
```

	Df	Sum of Sq	RSS	AIC	F value	Pr (F)
<none>			4.9085	0.8838		
x:z	1	0.1807	5.0892	-0.7546	0.2209	0.655

```
> drop1(lm(y ~ x*z), .~., test = "F")
```

Single term deletions

Model:

$y \sim x * z$

	Df	Sum of Sq	RSS	AIC	F value	Pr(F)
<none>			4.9085	0.8838		
x	1	0.4441	5.3526	-0.2500	0.5429	0.4890
z	1	0.1633	5.0718	-0.7888	0.1997	0.6707
x:z	1	0.1807	5.0892	-0.7546	0.2209	0.6549

> model.matrix(~ x*z)

	(Intercept)	x	z2	x:z2
1	1	1	0	0
2	1	2	0	0
3	1	3	0	0
4	1	4	0	0
5	1	5	0	0
6	1	6	1	6
7	1	7	1	7
8	1	8	1	8
9	1	9	1	9

```
10          1 10  1  10
attr(,"assign")
[1] 0 1 2 3
attr(,"contrasts")
attr(,"contrasts")$z
[1] "contr.treatment"
```


V Maximum de vraisemblance

L'idée de vraisemblance est assez intuitive : on calcule la probabilité d'obtenir les observations.

Mais on ne peut pas interpréter cette quantité comme une probabilité car nous ne pouvons pas définir un ensemble d'événements dont la somme des probabilités serait 1. En d'autres termes, elle n'a aucune valeur prédictive.

$$L = \prod_{i=1}^n f_{\theta}(x_i)$$

```
> x <- rnorm(10)
> prod(dnorm(x))
[1] 4.370395e-06
> prod(dnorm(x, -1))
[1] 4.491077e-09
> prod(dnorm(x, 1))
[1] 1.930839e-07
```

Exercice

1. Essayez avec 600 valeurs.

Corrigé

```
1. > prod(dnorm(rnorm(600)))  
[1] 0
```

$$\ln L = \sum_{i=1}^n \ln(f_{\theta}(x_i))$$

```
> sum(dnorm(x, log = TRUE))  
[1] -12.34066  
> sum(dnorm(x, -1, log = TRUE))  
[1] -19.22117  
> sum(dnorm(x, 1, log = TRUE))  
[1] -15.46014
```

Quelle est l'estimation par le maximum de vraisemblance de cette moyenne ?

```
> m <- seq(-1, 1, 0.1)
> ## l <- numeric(length(m))
> ## for (i in 1:length(l))
> ##   l[i] <- sum(dnorm(x, m[i], log = TRUE))
> l <- sapply(m,
+ function(y) sum(dnorm(x, y, log = TRUE)))
> plot(m, l)
```

On cherche donc à maximiser la fonction... :

```
function(y) sum(dnorm(x, y, log = TRUE))
```

... ou à minimiser :

```
function(y) -2*sum(dnorm(x, y, log = TRUE))
```

$$\text{Deviance} = -2 \times \ln L$$

La fonction `nlm` du package `stats` permet de minimiser une fonction sans en connaître les dérivées.

```
> fd <- function(p) -2*sum(dnorm(x, p, log = TRUE))
> nlm(fd, 1)
$minimum
[1] 24.32768

$estimate
[1] 0.1880516

$gradient
[1] 1.024958e-05

$code
[1] 1
```

```
$iterations
```

```
[1] 1
```

```
> mean(x) # rassurant, non ?
```

```
[1] 0.1880516
```

La théorie nous dit que :

$$SE(\hat{\theta}) = \left(\left[-\frac{\partial^2 \ln L}{\partial \theta^2} \right]_{\hat{\theta}} \right)^{-\frac{1}{2}}$$

Comment obtenir cette valeur de la dérivée seconde au maximum ?

```
> out <- nlm(fd, 1, hessian = TRUE)
```

```
> se <- solve(diag(out$hessian))
```

```
> ## intervalle de confiance a 95%:
```

```
> out$estimate - 1.96*se
```

```
      [,1]
[1,] 0.09005162
> out$estimate + 1.96*se
      [,1]
[1,] 0.2860516
```

Je peux estimer les paramètres d'un modèle donné, mais comment je choisis ce modèle ?

Si deux modèles sont emboîtés (l'un est un cas particulier de l'autre), alors le quotient de leur vraisemblance multiplié par 2 suit une loi du χ^2 avec un nombre de degrés de liberté égal à la différence du nombre de paramètres entre les deux modèles. Plus simplement :

$$Dev_1 - Dev_2 \sim \chi^2 \quad df = k_2 - k_1$$

Ce test (*likelihood ratio test*, LRT) n'a de sens que pour un jeu de données considéré.

```
> x1 <- rnorm(10)
> x2 <- rnorm(10, 1)
> ## modèle nul (k = 1):
> x <- c(x1, x2)
> nlm(fd, 1)
$minimum
[1] 61.29482

$estimate
[1] 0.9590089
.....
> ## modèle alternatif (k = 2):
> x <- x1
> nlm(fd, 1)
$minimum
[1] 23.92164

$estimate
[1] 0.2389253
```

```
.....  
> x <- x2  
> nlm(fd, 1)  
$minimum  
[1] 27.00276  
  
$estimate  
[1] 1.679092  
.....  
> (chi <- 61.29482 - (23.92164 + 27.00276))  
[1] 10.37042  
> 1 - pchisq(chi, 2)  
[1] 0.001280504
```

→ LRT : $\chi_1^2 = 10.37, P = 0.001$

Si les modèles ne sont pas emboîtés, ils peuvent être comparés par le critère d'information d'Akaike $AIC = Dev + 2 \times k$.

Le modèle avec la plus petite valeur d'AIC doit être préféré.

```
> 61.29482 + 2*1
```

```
[1] 63.29482
```

```
> 23.92164 + 27.00276 + 2*2
```

```
[1] 54.9244
```

Ce critère n'a aucune signification absolue et ne doit servir qu'à comparer des modèles ajustés au même jeu de données.

Exercice

1. Dans l'exemple ci-dessus, on sait que la loi normale a deux paramètres (μ, σ^2) : pourquoi avoir pris $k = 1$? Modifier `fd` pour avoir $k = 2$.
2. On sait que pour un risque constant λ les temps de survie t suivent une loi exponentielle dont la densité $f(t) = \lambda e^{-\lambda t}$ peut être calculée avec `deexp`. Écrire un programme en R qui calculera, pour un échantillon, l'estimation par maximum de vraisemblance de λ ($\hat{\lambda}$).

Corrigé

1. L'estimation n'est faite que sur la moyenne μ , la variance étant fixée $\sigma^2 = 1$.
Pour estimer les deux paramètres simultanément :

```
fd <- fonction(p) -2*sum(dnorm(x, p[1], p[2], log = TRUE))
```

2. Il y a deux approches : la première est similaire à celle utilisée plus haut pour la loi normale :

```
fd <- fonction(p) -2*sum(dexp(x, p, log = TRUE))
```

La seconde approche est d'écrire la log-vraisemblance : $\ln L = n \ln \lambda - \lambda \sum t_i$ ($i = 1, \dots, n$). On trouve facilement la dérivée en fonction de λ : $\partial \ln L / \partial \lambda = n/\lambda - \sum t_i$. On pose cette dérivée égale à zéro pour trouver $\hat{\lambda} = n / \sum t_i$, soit dans R `1/mean(x)`.

Écriture probabiliste d'un modèle linéaire simple : $y_i \sim \mathcal{N}(\beta x_i + \alpha, \sigma^2)$ pour x_i donné (distribution conditionnelle). Donc :

$$L = \prod_{i=1}^n f_{\theta}(y_i) \quad \theta = \{\beta, \alpha, \sigma^2\}$$

```
fd <- function(p) {  
  m <- p[1]*x + p[2]  
  -2*sum(dnorm(y, m, p[3], log = TRUE))  
}
```

avec $p[1] = \beta$ $p[2] = \alpha$ $p[3] = \sigma^2$

Note : la fonction passée à `nlm` doit être définie avec un seul argument (le vecteur de paramètres).

```
> x <- 1:50  
> y <- 1.5*x + 8 + rnorm(50, 0, 10)  
> nlm(fd, c(1, 1, 1))
```

```
$minimum
```

```
[1] 379.3798
```

```
$estimate
```

```
[1] 1.301263 14.872885 10.749500
```

```
$gradient
```

```
[1] 1.380391e-05 1.429409e-06 -1.681586e-06
```

```
.....
```

```
> lm(y ~ x)$coeff
```

```
(Intercept)          x  
 14.872842    1.301265
```

```
> summary(lm(y ~ x))$sigma
```

```
[1] 10.97117
```

```
> AIC(lm(y ~ x))
```

```
[1] 385.3798
```

VI Modèles linéaires généralisés

Deux caractéristiques du modèle linéaire classique sont limitantes : (i) la moyenne prédite est distribuée entre $-\infty$ et $+\infty$, (ii) l'hypothèse de normalité des résidus.

Les transformations de la réponse font perdre la nature des données analysées (fécondité, effectifs, durée de vie, ...).

L'idée des modèles linéaires généralisés (MLG ou *GLM* pour *generalized linear models*) est de reprendre le modèle linéaire $E(y) = \beta x$ et de l'étendre à d'autres distributions. Pour prendre en compte que la moyenne peut être distribuée sur un intervalle fini, on considérera une fonction de celle-ci :

$$g[E(y)] = \beta x$$

telle que $-\infty < g[E(y)] < +\infty$. g est appelée une fonction de **lien** (*link*).

 Les données originales ne sont pas transformées $g[E(y)] \neq g(y)$.

Quatre distributions sont couramment utilisées :

Gaussienne : réponse continue entre $-\infty$ et $+\infty$

Poisson : réponse entière entre 0 et $+\infty$

Binomiale : réponse entière entre 0 et n (nb de cas)

Gamma : réponse continue entre 0 et $+\infty$

La distribution des données pouvant être asymétrique, la méthode des moindres carrés n'est pas applicable : les MLG sont ajustés par maximum de vraisemblance.

```
> gm1 <- glm(litter.size ~ mass.g., data = LIFEHIST)
> summary(gm1)
```

Call:

```
glm(formula = litter.size ~ mass.g., data = LIFEHIST)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-1.8053	-1.7253	-0.3053	1.1947	11.3747

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	2.805e+00	4.949e-02	56.68	< 2e-16
mass.g.	-2.605e-08	9.238e-09	-2.82	0.00488

(Dispersion parameter for gaussian family taken to be 3.130944)

Null deviance: 4045.0 on 1285 degrees of freedom
Residual deviance: 4020.1 on 1284 degrees of freedom
(154 observations deleted due to missingness)
AIC: 5121.3

Number of Fisher Scoring iterations: 2

Pour une distribution gaussienne, moindres carrés et maximum de vraisemblance sont équivalents.

```
> anova(gml, test = "Chisq")
```

```
Analysis of Deviance Table
```

```
Model: gaussian, link: identity
```

```
Response: litter.size
```

```
Terms added sequentially (first to last)
```

	Df	Deviance	Resid. Df	Resid. Dev	P(> Chi)
NULL			1285	4045.0	
mass.g.	1	24.9	1284	4020.1	0.004801

```
> 1 - pchisq(24.9/3.13, 1)
```

```
[1] 0.004794751
```

AIC peut être utilisé à la place du LRT :

1. modèles non-emboîtés ;

2. grand nombre de modèles considérés a priori (évite tests 2 à 2) ;
3. l'objectif est de sélectionner un modèle à but prédictif (car AIC est moins conservateur que LRT).

Le second argument de `glm` est `family` qui est une fonction et par défaut `family = gaussian` :

```
> args(gaussian)
function (link = "identity")
```

Par défaut `glm` ajuste le même modèle que `lm`.

Trois liens sont possibles pour `gaussian` : "identity", "log" et "inverse".

Essayons le lien inverse :

```
> gmlinv <- glm(litter.size ~ mass.g., gaussian("inverse"),
+              data = LIFEHIST)
```

```
> summary(gmlinv)
```

```
Call:
```

```
glm(formula = litter.size ~ mass.g., family = gaussian("inverse"  
  data = LIFEHIST)
```

```
Deviance Residuals:
```

Min	1Q	Median	3Q	Max
-2.5995	-1.0753	0.1331	0.9656	10.5914

```
Coefficients:
```

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	2.774e-01	4.676e-03	59.315	<2e-16
mass.g.	2.882e-05	3.218e-06	8.957	<2e-16

```
(Dispersion parameter for gaussian family taken to be 2.418827)
```

```
Null deviance: 4045.0 on 1285 degrees of freedom  
Residual deviance: 3105.3 on 1284 degrees of freedom
```

(154 observations deleted due to missingness)

AIC: 4789.2

Number of Fisher Scoring iterations: 23

Les deux modèles sont :

$$\begin{aligned} \text{gm1} : \quad E(y) &= -2.60 \times 10^{-8} \text{ mass} + 2.80 \quad \hat{\sigma}^2 = 3.13 \\ \text{gm1inv} : \quad \frac{1}{E(y)} &= 2.88 \times 10^{-5} \text{ mass} + 0.28 \quad \hat{\sigma}^2 = 2.42 \end{aligned}$$

L'ajustement est sensiblement amélioré par le lien inverse (mais n'oublions pas les données sont hétérogènes).

Les diagnostics de régression sont identiques à ceux du modèle linéaire, avec la différence que, par défaut, les résidus de déviance sont calculés.

Distribution de Poisson

Les liens permis sont "log" (le défaut), "identity" et "sqrt".

```
> x <- runif(50, 1, 50)
> y <- rpois(50, x)
> plot(x, y)
> gm4 <- glm(y ~ x)
> gm5 <- glm(y ~ x, poisson)
> summary(gm4)
```

Call:

```
glm(formula = y ~ x)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-7.8011	-2.0088	-0.5986	1.8911	11.3623

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-1.70545	0.95811	-1.78	0.0814
x	1.11443	0.03506	31.79	<2e-16

(Dispersion parameter for gaussian family taken to be 13.40717)

Null deviance: 14191.68 on 49 degrees of freedom
Residual deviance: 643.54 on 48 degrees of freedom
AIC: 275.64

Number of Fisher Scoring iterations: 2

```
> summary(gm5)
```

Call:

```
glm(formula = y ~ x, family = poisson)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-----	----	--------	----	-----

-3.73525 -0.79706 -0.02801 0.79571 1.84332

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	1.830827	0.075914	24.12	<2e-16
x	0.047641	0.002117	22.51	<2e-16

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 630.755 on 49 degrees of freedom
Residual deviance: 66.642 on 48 degrees of freedom
AIC: 302.63

Number of Fisher Scoring iterations: 4

La prise en compte de la nature de la distribution de la réponse améliore très nettement la dispersion des résidus.

On notera le paramètre de dispersion égal à 1 (car la variance est contrainte à être égale à la moyenne dans la loi de Poisson).

Distribution binomiale

La réponse est spécifiée dans `glm` avec :

1. une matrice à deux colonnes où la première est le nombre de « succès » et la seconde le nombre d'« échecs »
2. un vecteur avec des 0 et des 1 (ou `FALSE` et `TRUE`)
3. un facteur où le premier niveau (codé 1) est pris comme « échec » et les autres comme « succès »

```
> y <- sample(0:1, size = 20, replace = TRUE)
```

```
> x <- 1:20
```

```
> gmb1 <- glm(y ~ x, binomial)
```

```
> gmb2 <- glm(cbind(y, 1 - y) ~ x, binomial)
```

```
> gmb3 <- glm(factor(y) ~ x, binomial)
```

```
> AIC(gmb1); AIC(gmb2); AIC(gmb3)
```

```
[1] 23.67469
```

```
[1] 23.67469
```

```
[1] 23.67469
```

Dans les deux premières situations il est aisé d'inverser les succès et les échecs :

```
> glm(!y ~ x, binomial)
> glm(cbind(1 - y, y) ~ x, binomial)
```


Les coefficients seront de signe opposé. Si on a préparé une matrice au préalable :

```
> Y <- cbind(y, 1 - y)
> glm(Y ~ x, binomial)
> glm(Y[, 2:1] ~ x, binomial)
```

Par défaut le lien est la fonction logit. Les choix possibles sont :

"logit"	$\ln \frac{p}{1-p}$ (régression logistique)
"probit"	$F_{\mathcal{N}}^{-1}(p)$
"cauchit"	$F_{\mathcal{C}}^{-1}(p)$
"log"	$\ln p$
"cloglog"	$\ln(-\ln(1-p))$

$p = E(y)$; F^{-1} est l'inverse de la fonction de densité de probabilité cumulée.

 Le choix d'un lien n'est pas trivial. Il est recommandé, pour débiter, d'utiliser le lien par défaut.

```
> library(MASS)
> data(Aids2)
> ?Aids2
```

Le genre a-t-il un effet sur la « survie » ?

```
> aids.m1 <- glm(status ~ sex, binomial, data = Aids2)
> anova(aids.m1, test = "Chisq")
```

Analysis of Deviance Table

Model: binomial, link: logit

Response: status

Terms added sequentially (first to last)

	Df	Deviance	Resid. Df	Resid. Dev	P(> Chi)
NULL			2842	3777.5	
sex	1	0.2	2841	3777.3	0.6

L'âge a-t-il un effet sur la « survie » ?

```
> aids.m2 <- glm(status ~ age, binomial, data = Aids2)
> anova(aids.m2, test = "Chisq")
```

Analysis of Deviance Table

Model: binomial, link: logit

Response: status

Terms added sequentially (first to last)

	Df	Deviance	Resid. Df	Resid. Dev	P(> Chi)
NULL			2842	3777.5	
age	1	6.8	2841	3770.7	0.009191

Une fois l'effet de l'âge pris en compte, le genre a-t-il un effet ? On prend soin de mettre `sex` après `age` dans la formule, sinon, il faudra tester les effets avec `drop1` (mais sans interaction).

```
> aids.m3 <- glm(status ~ age*sex, binomial, data = Aids2)
```

```
> anova(aids.m3, test = "Chisq")
```

Analysis of Deviance Table

Model: binomial, link: logit

Response: status

Terms added sequentially (first to last)

	Df	Deviance	Resid. Df	Resid. Dev	P(> Chi)
NULL			2842	3777.5	
age	1	6.8	2841	3770.7	0.009191
sex	1	0.2	2840	3770.5	0.6
age:sex	1	3.8	2839	3766.6	0.1

```
> summary(aids.m2)
```

Call:

```
glm(formula = status ~ age, family = binomial, data = Aids2)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-1.5696	-1.3663	0.9450	0.9917	1.1303

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	0.111819	0.149300	0.749	0.4539
age	0.010065	0.003881	2.593	0.0095

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 3777.5 on 2842 degrees of freedom
Residual deviance: 3770.7 on 2841 degrees of freedom
AIC: 3774.7

Number of Fisher Scoring iterations: 4


Le modèle sélectionné est donc (p : probabilité de décès) :

$$\ln\left(\frac{p}{1-p}\right) = 0.01 \text{ age} + 0.11$$

```
> range(Aids2$age)
[1] 0 82
> a <- seq(0, 82, 0.1)
> risk <- predict(aids.m2, newdata = data.frame(age = a),
+               type = "response")
> plot(a, risk, type = "l", xlab = "Âge (années)",
+      ylab = "Probabilité de décès", ylim = 0:1)
> title("Risque de mortalité prédit par le modèle \"aids.m2\"")
```

La page d'aide de `predict.glm` inclut des informations très intéressantes sur comment calculer un intervalle de prédiction.

Loi gamma

 Gamma () donne la loi gamma pour un MLG ;
gamma (x) calcule la fonction $\Gamma(x) = 1 \times 2 \times \dots \times (x - 1)$ pour x entier.

Les liens permis sont "inverse" (le défaut), "identity" et "log".

```
> Aids2$surv <- Aids2$death - Aids2$diag
> surv.m1 <- glm(surv ~ age, Gamma, data = Aids2,
+               subset = surv > 0)
> anova(surv.m1, test = "Chisq")
```

Analysis of Deviance Table

Model: Gamma, link: inverse

Response: surv

Terms added sequentially (first to last)

```
      Df Deviance Resid. Df Resid. Dev P(>|Chi|)
NULL          2813      3219.5
age         1      27.5      2812      3192.0 3.945e-09
> summary(surv.m1)
```

Call:

```
glm(formula = surv ~ age, family = Gamma, data = Aids2, subset =
      0)
```

Deviance Residuals:

```
      Min       1Q   Median       3Q      Max
-3.2565  -0.9426  -0.2280   0.3740   3.0442
```

Coefficients:

```
      Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.539e-03 1.543e-04  9.969 < 2e-16
age         2.471e-05 4.199e-06  5.884 4.47e-09
```


(Dispersion parameter for Gamma family taken to be 0.7935563)

Null deviance: 3219.5 on 2813 degrees of freedom
Residual deviance: 3192.0 on 2812 degrees of freedom
AIC: 39502

Number of Fisher Scoring iterations: 6

Formulation générale des MLG

$$g(E[y]) = \beta X \quad \text{Var}(y_i) = \phi \mathcal{V}(E[y_i])$$

ϕ : paramètre de dispersion ; \mathcal{V} : fonction de variance

Réponse	g^*	ϕ	\mathcal{V}	$\text{Var}(y_i)$
normale	identité	σ^2	.	σ^2
gamma	inverse	$1/\nu$	μ^2	μ^2/ν
Poisson	log	1	μ	μ
binomiale	logit	1	$\mu(1 - \mu)$	$\mu(1 - \mu)n_i$

*fonction de lien par défaut dans R

L'utilisateur choisit la distribution de la réponse et la fonction de lien.
Le reste découle de ces choix.

Sur- (sous-) dispersion avec les lois binomiale et de Poisson

Avec ces deux lois de distribution, la variance est contrainte par la moyenne. La méthode de quasivraisemblance permet de lever cette contrainte.

Seules la moyenne et la variance des observations sont considérées : il n'est donc pas possible de calculer la vraisemblance (pas d'AIC). La comparaison avec un MLG standard par LRT n'est donc pas possible.

En pratique, on considère que si $\phi > 4$ (ou $\phi < 0.25$), il y a sur- (sous) dispersion. Ces phénomènes sont généralement dûs à une dépendance entre observations.

```
> summary(glm(status ~ age, quasibinomial, data = Aids2))
```

Call:

```
glm(formula = status ~ age, family=quasibinomial, data=Aids2)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-----	----	--------	----	-----

-1.5696 -1.3663 0.9450 0.9917 1.1303

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	0.111819	0.149342	0.749	0.45407
age	0.010065	0.003882	2.593	0.00957

(Dispersion parameter for quasibinomial family
taken to be 1.000563)

Null deviance: 3777.5 on 2842 degrees of freedom
Residual deviance: 3770.7 on 2841 degrees of freedom
AIC: NA

Number of Fisher Scoring iterations: 4

La fonction `quasi` permet de construire une famille avec lien et fonction de variance spécifiques (?`quasi` pour tous les détails : toutes les familles de distributions sont documentées sur cette page).

Exercice

1. Reprendre le modèle `aids.m2` et tracer l'intervalle de confiance à 95% du risque prédit. Le graphe sera légendé.
2. Charger les données `eagles` du package `MASS`.
 - (a) Reprendre le modèle `eagles.glm` dans les exemples et analyser le avec `anova` et `summary`. Commenter.
 - (b) Ré-ajuster ces modèles avec `family = quasibinomial`. Que constatez-vous ? (Comparer notamment les résultats avec ou sans la variable `v`.) Commenter sur la récolte des données.

Corrigé

1. On utilise l'option `se.fit=TRUE` de `predict.glm` :

```
risk <- predict(aids.m2, newdata = data.frame(age = a),
               type = "response", se.fit = TRUE)
plot(a, risk$fit, type = "l", xlab = "Âge (années)",
     ylab = "Probabilité de décès", ylim = 0:1)
lines(a, risk$fit - 1.96 * risk$se.fit, lty = 2)
```

```
lines(a, risk$fit + 1.96 * risk$se.fit, lty = 2)
legend("topleft", c("Risque estimé",
                    "Intervalle de confiance à 95%"), lty = 1:2)
```

```
2. (a) > anova(eagles.glm, test = "Chisq")
```

```
.....
```

	Df	Deviance	Resid.	Df	Resid.	Dev	P(> Chi)
NULL				7		128.267	
P	1	48.994		6		79.273	2.567e-12
A	1	16.042		5		63.232	6.196e-05
V	1	56.275		4		6.956	6.300e-14
P:A	1	6.624		3		0.333	0.010

```
> summary(eagles.glm)
```

```
.....
```

```
Coefficients:
```

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	0.8977	0.4480	2.004	0.04507
PS	-3.4605	1.1287	-3.066	0.00217
AI	-0.3590	0.5986	-0.600	0.54870
VS	5.4324	1.3602	3.994	6.5e-05

```
PS:AI          -3.6614      1.6279   -2.249   0.02450
.....
```

La significativité de A est différente selon que cette variables testée en l'absence (anova) ou en présence (summary) de V.

```
(b) > summary(glm(cbind(y, n - y) ~ P*A + V, quasibinomial,
+ data = eagles))
```

```
.....
```

```
Coefficients:
```

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	0.8977	0.1139	7.882	0.004256
PS	-3.4605	0.2870	-12.059	0.001227
AI	-0.3590	0.1522	-2.359	0.099518
VS	5.4324	0.3458	15.708	0.000561
PS:AI	-3.6614	0.4139	-8.846	0.003045

```
(Dispersion parameter for quasibinomial family taken
to be 0.06464582)
```

```
.....
```

```
> summary(glm(cbind(y, n - y) ~ P*A, quasibinomial,  
+ data = eagles))
```

.....

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.8827	1.3879	1.357	0.246
PS	-1.5950	1.9061	-0.837	0.450
AI	-0.5744	1.8474	-0.311	0.771
PS:AI	-3.1473	4.1466	-0.759	0.490

(Dispersion parameter for quasibinomial family taken
to be 11.70334)

.....

Les effets de P et de A sont dépendants de V . On peut suspecter qu'un même individu est entré plusieurs fois en tant que victime dans l'étude.

Double MLG et variance structurée

Idée : modéliser la dispersion en fonction d'un prédicteur (Aitkin, 1987, Modelling variance heterogeneity in normal regression using GLIM, *Appl. Stat.*).

$$y_i = \beta x_i + \beta_0 + \epsilon_i \quad \text{"mean model"}$$

$$\ln(\epsilon_i^2) = \lambda z_i + \lambda_0 + \nu_i \quad \nu_i \sim \Gamma \quad \text{"variance model"}$$

L'algorithme d'Aitkin est codé dans la fonction `VarDisp`.

Exercice

1. Étudier le code de la fonction `VarDisp`. En déduire l'algorithme d'Aitkin.
2. Simuler des données avec un modèle linéaire dont la variance est hétérogène ou pas. Tester la fonction `VarDisp` avec ces données.

Corrigé

1. Un modèle `lm(y ~ x)` est ajusté, puis un `glm(res ~ z, Gamma)` où `res` sont les résidus du premier modèle. La procédure est répétée en utilisant

l'inverse des valeurs prédites par le second modèle en guise de poids pour le premier, et itérée jusqu'à convergence de la déviance.

```
2. > x <- 1:100
> z <- rep(1:25, 4)
> y <- 0.1*x + 1 + rnorm(100, 0, exp(0.1*z))
> yb <- .1*x + 1 + rnorm(100)
> VarDisp(x, y, z)
.....
x          0.103293    0.009401    10.987    <2e-16
.....
z          0.18711    0.02007     9.324    3.56e-15
.....
> VarDisp(x, yb, z)
.....
x          0.103119    0.003151    32.722    < 2e-16
.....
z         -0.01360    0.01710    -0.795     0.428
.....
```

Pour une solution plus générale : package nlme.

Les modèles linéaires généralisés mixtes (GLMM) : pas de solution satisfaisante pour le moment.

Pour les amateurs :

`glmmPQL` dans **MASS** : résultats très similaires à ceux de `glm`; pas de déviance car quasivraisemblance (pénalisée).

`lmer` dans **lme4** : interface différente de `lme`, notamment pas d'argument `random`.

Autres packages sur CRAN : `glmmAK`, `glmmML`, `glmmBUGS`, `mlmmm`

`repeated` implémente les "HGLM" (*hierarchical GLM*)

`popgen.unimass.nl/~jlindsey`

VII Modèles linéaires mixtes

```
> library(lattice)
> histogram(~ litter.size | order, data = LIFEHIST)
> xyplot(litter.size ~ log(mass.g.) | order, data = LIFEHIST)
> LIFEHIST$LNmass <- log(LIFEHIST$mass.g.)
```

Si l'ordonnée à l'origine (*intercept*) varie aléatoirement entre chaque ordre au lieu de varier systématiquement (modèles m_2 et m_3) ?

```
> library(nlme)
> m.e.i <- lme(litter.size ~ LNmass, random = ~ 1 | order,
+ data = LIFEHIST, na.action = na.omit)
```

Dans une formule, le terme ~ 1 désigne l'ordonnée à l'origine. Pour rappel, $\text{lm}(y \sim x - 1)$ fait la régression par l'origine.

Le défaut de `na.action` pour `lme` n'est pas le même que pour `lm`.

```
> summary(m.e.i)
```

```
Linear mixed-effects model fit by REML
```

```
Data: LIFEHIST
```

```
          AIC      BIC    logLik
4468.979 4489.61 -2230.490
```

```
Random effects:
```

```
Formula: ~1 | order
```

```
(Intercept) Residual
```

```
StdDev:    0.7127022 1.350317
```

```
Fixed effects: litter.size ~ LNmass
```

	Value	Std.Error	DF	t-value	p-value
(Intercept)	3.879222	0.27719053	1268	13.994786	0
LNmass	-0.220054	0.02203408	1268	-9.986987	0

```
Correlation:
```

```
(Intr)
```

LNmass -0.677

Standardized Within-Group Residuals:

Min	Q1	Med	Q3	Max
-2.33112046	-0.58003277	-0.09435008	0.35037399	7.80802878

Number of Observations: 1286

Number of Groups: 17

$BIC = Dev + k \times \ln(n)$ *Bayesian information criterion*

Si la pente (β) varie aléatoirement entre chaque ordre au lieu de varier systématiquement (modèle m3) ?

```
> m.e.b <- update(m.e.i, random = ~ LNmass - 1 | order)
```

```
> summary(m.e.b)
```

Linear mixed-effects model fit by REML

Data: LIFEHIST

AIC	BIC	logLik
4501.452	4522.083	-2246.726

Random effects:

Formula: ~LNmass - 1 | order

LNmass	Residual
--------	----------

StdDev: 0.09773808	1.365796
--------------------	----------

Fixed effects: litter.size ~ LNmass

	Value	Std.Error	DF	t-value	p-value
(Intercept)	4.594547	0.13671942	1268	33.60567	0
LNmass	-0.300379	0.03222036	1268	-9.32264	0

Correlation:

(Intr)

LNmass	-0.528
--------	--------

Standardized Within-Group Residuals:

Min	Q1	Med	Q3	Max
-2.2540418	-0.6298213	-0.1156598	0.3808878	7.6210840

```
Number of Observations: 1286
```

```
Number of Groups: 17
```

Et les deux ?

```
> update(m.e.i, random = ~ LNmass | order)
```

Équivalent à ..., random = ~ LNmass + 1 | order

Comment supprimer les ordres qui ont des effectifs faibles ?

```
> Norder <- table(LIFEHIST$order)
```

```
> names(Norder)[Norder > 55]
```

```
[1] "Artiodactyla" "Carnivora"      "Insectivora"
```

```
[4] "Primates"     "Rodentia"
```

```
> X <- subset(LIFEHIST, order %in% names(Norder)[Norder > 55])
```



```
> #==LIFEHIST[LIFEHIST$order %in% names(Norder)[Norder > 55], ]
> m.e.bi <- update(m.e.i, random = ~ LNmass | order, data = X)
> summary(m.e.bi)
```

Linear mixed-effects model fit by REML

Data: X

	AIC	BIC	logLik
	4011.91	4042.132	-1999.955

Random effects:

Formula: ~LNmass | order

Structure: General positive-definite, Log-Cholesky parametrization

	StdDev	Corr
(Intercept)	1.8902398	(Intr)
LNmass	0.1291065	-0.989
Residual	1.3820060	

Fixed effects: litter.size ~ LNmass

	Value	Std.Error	DF	t-value	p-value
(Intercept)	3.925094	0.8775381	1134	4.472847	0.0000

```
LNmass      -0.204805  0.0641022  1134  -3.194973   0.0014
Correlation:
(Intr)
LNmass -0.97
```

Standardized Within-Group Residuals:

Min	Q1	Med	Q3	Max
-2.2904947	-0.5787237	-0.1302723	0.3899087	7.7528960

Number of Observations: 1140

Number of Groups: 5

REML (*residual maximum likelihood*) corrige le biais dans l'estimation des composants de variance dû au fait que des effets fixes sont estimés (on retrouve cette idée dans l'estimation de la variance d'un échantillon sur $n - 1$ et non n).

La significativité des effets aléatoires peut être testée en comparant avec le même modèle ajusté par l_m , mais le modèle mixte doit être ajusté par ML et non REML. (Ici on prend soin de réajuster m_1 avec le prédicteur transformé.)

```
> m.e.iML <- update(m.e.i, method = "ML")
> m1 <- update(m1, formula = litter.size ~ log(mass.g.))
> anova(m.e.iML, m1)
```

	Model	df	AIC	BIC	logLik	Test
m.e.iML	1	4	4461.805	4482.442	-2226.902	
m1	2	3	4648.245	4663.723	-2321.122	1 vs 2

L.Ratio p-value

m.e.iML		
m1	188.4402	<.0001

Pour s'assurer que les modèles ont été ajustés aux mêmes données :

```
> str(m.e.iML$residuals)
 num [1:1286, 1:2] 0.325 -0.218 -0.545 -0.261 -0.627 ...
- attr(*, "dimnames")=List of 2
 ..$ : chr [1:1286] "1" "2" "3" "4" ...
 ..$ : chr [1:2] "fixed" "order"
- attr(*, "std")= num [1:1286] 1.35 1.35 1.35 1.35 1.35 ...
> str(m1$residuals)
 Named num [1:1286] -0.954 -1.801 -1.804 -1.801 -1.805 ...
```

```
- attr(*, "names")= chr [1:1286] "1" "2" "3" "4" ...
```

Exercice

1. Tester les effets aléatoires du modèle `m.e.b`.
2. Faire de même avec `m.e.bi` (indice : il faudra encore réduire le jeu de données).

Corrigé

```
1. > m.e.bML <- update(m.e.b, method = "ML")
```

```
> anova(m.e.bML, m1)
```

	Model	df	AIC	BIC	logLik	Test
m.e.bML	1	4	4493.900	4514.537	-2242.950	
m1	2	3	4648.245	4663.723	-2321.122	1 vs 2
			L.Ratio	p-value		
m.e.bML						
m1			156.3448	<.0001		

```

2. > Xb<-subset(LIFEHIST, order %in% names(Norder) [Norder>160])
> m.e.biML <- update(m.e.bi, method = "ML", data = Xb)
> anova(m.e.biML, update(m1, data = Xb))

```

	Model	df	AIC	BIC	logLik
m.e.biML	1	6	3289.047	3317.954	-1638.524
update(m1, data = Xb)	2	3	3313.186	3327.639	-1653.593


```


```

	Test	L.Ratio	p-value
m.e.biML			
update(m1, data = Xb)	1 vs 2	30.13885	<.0001

Le modèle statistique derrière `m.e.i` est :

$$\text{litter.size}_i = \beta \text{LNmass}_i + \alpha + \zeta_j + \epsilon_i$$

où j est l'indice de l'ordre, $\zeta_j \sim \mathcal{N}(0, \sigma_O^2)$, $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$.

Les composants de variance estimés sont $\hat{\sigma}_O = 0,71$ et $\hat{\sigma} = 1,35$. ($\hat{\sigma}_O$ et $\hat{\sigma}$ sont indépendants).

L'estimation des effets fixes (le vecteur β) n'est pas biaisée par la présence d'effets aléatoires, mais les erreurs-standards sont sous-estimées si ces derniers sont ignorés. Cela se comprend dans le sens où la variation aléatoire entre les ordres est « captée » par le modèle à effet fixe.

Le modèle derrière `m.e.b` est :

$$\text{litter.size}_i = (\beta + \eta_j)\text{LNmass}_i + \alpha + \epsilon_i \quad \eta_j \sim \mathcal{N}(0, \sigma_\beta^2)$$

Le modèle derrière `m.e.bi` est évidemment :

$$\text{litter.size}_i = (\beta + \eta_j)\text{LNmass}_i + \alpha + \zeta_j + \epsilon_i$$

η et ζ peuvent être corrélés. On observe ici $\text{Corr}(\eta, \zeta) = -0.989$.

La fonction `intervals` calcule les intervalles de confiance à 95% de tous les paramètres du modèles, y compris les (co)variances :

```
> intervals(m.e.bi)
```

```
Approximate 95% confidence intervals
```

```
Fixed effects:
```

	lower	est.	upper
(Intercept)	2.2033132	3.9250940	5.64687480
LNmass	-0.3305769	-0.2048047	-0.07903251

```
attr(,"label")
```

```
[1] "Fixed effects:"
```

```
Random Effects:
```

```
Level: order
```

	lower	est.	upper
sd((Intercept))	0.85741871	1.8902398	4.1671665
sd(LNmass)	0.05190057	0.1291065	0.3211620
cor((Intercept),LNmass)	-0.99997329	-0.9891946	0.3768320

```
Within-group standard error:
```

lower	est.	upper
1.326139	1.382006	1.440226

Coefficients et valeurs prédites

```
> coef(m.e.i)
```


	(Intercept)	LNmass
Artiodactyla	3.682869	-0.2200541
Carnivora	4.720206	-0.2200541
Cetacea	3.976942	-0.2200541
....		

```
> coef(m.e.b)
```

	(Intercept)	LNmass
Artiodactyla	4.594547	-0.3001138
Carnivora	4.594547	-0.2135372
Cetacea	4.594547	-0.2571568
....		

Ces coefficients sont la somme des effets fixes et des effets aléatoires. Ces derniers sont estimés par **BLUP** (*best linear unbiased prediction*).

Les effets fixes et les effets aléatoires peuvent être extraits séparément avec `fixed.effects` et `random.effects`.

 Les effets aléatoires ne sont pas des paramètres du modèle (d'où “prediction” et pas “estimation”).

`predict.lme` a une option `level` qui précise le niveau pour lequel les prédictions doivent être faites.

Exercice

1. Simuler des données avec $y_{ij} = 0.2x_{ij} + \zeta_j + \epsilon_{ij}$, pour $i = 1, \dots, 20$, $j = 1, \dots, 5$, $x_{ij} = 1, \dots, 20$ pour tous les j , $\zeta_j \sim \mathcal{N}(0, 0.5)$, et $\epsilon_{ij} \sim \mathcal{N}(0, 0.3)$.
2. Ajuster le modèle linéaire mixte correspondant. On donnera les “labels” A à E aux niveaux (g). Comparer les valeurs observées de y et celles prédites par le modèle.
3. Faire une prédiction pour $\{x = 30, g = E\}$ et $\{x = 30, g = F\}$.

Corrigé

```
1. x <- rep(1:20, 5)
   b <- rnorm(5, 0, 0.5)
   y <- 0.2*x + rep(b, each = 20) + rnorm(100, 0, 0.3)

2. g <- gl(5, 20, labels = LETTERS[1:5])
   m <- lme(y ~ x, random = ~ 1 | g)
   plot(predict(m), y)

3. > newx <- data.frame(x = c(30, 30), g = c("E", "F"))
   > predict(m, newx)
           E           F
5.783796           NA
attr(,"label")
[1] "Predicted values"
> predict(m, newx, level = 0:1)
   g predict.fixed predict.g
1 E      6.04341    5.783796
2 F      6.04341           NA
```

Les effets aléatoires peuvent être hiérarchiques.

```
> update(m.e.i, random = ~ 1 | order/family)
```

```
Linear mixed-effects model fit by REML
```

```
Data: LIFEHIST
```

```
Log-restricted-likelihood: -2177.54
```

```
Fixed: litter.size ~ LNmass
```

```
(Intercept)          LNmass
```

```
3.2705311   -0.1524292
```

```
Random effects:
```

```
Formula: ~1 | order
```

```
(Intercept)
```

```
StdDev:    0.5839318
```

```
Formula: ~1 | family %in% order
```

```
(Intercept) Residual
```

```
StdDev:    0.6397829 1.256388
```

Number of Observations: 1286

Number of Groups:

```
order family %in% order
      17           99
```

Rappel : `order/family` **et** `order + family %in% order` sont identiques.

Exercice

1. Essayez les effets hiérarchiques `order/family` avec le modèle `m.e.b.`

Corrigé

```
1. > (m.e.b.OF <- update(m.e.b, random=~LNmass-1|order/family))
Linear mixed-effects model fit by REML
Data: LIFEHIST
Log-restricted-likelihood: -2195.294
Fixed: litter.size ~ LNmass
(Intercept)      LNmass
```

4.3759198 -0.2783661

Random effects:

Formula: ~LNmass - 1 | order
LNmass

StdDev: 0.08818959

Formula: ~LNmass - 1 | family %in% order
LNmass Residual

StdDev: 0.07667828 1.273524

Number of Observations: 1286

Number of Groups:

order	family %in%	order
17		99

> AIC(m.e.b)

[1] 4501.452

> AIC(m.e.b.OF)

[1] 4400.589

Dépendance entre observations

`lme` a l'option `correlation` qui prend pour argument une fonction qui spécifie la forme de la dépendance entre observations ; celles fournies avec `nlme` sont documentées à `?corClasses`.

La plus simple est `corCompSymm` qui spécifie que toutes les observations sont corrélées de façon identique : il n'y a donc qu'un paramètre à estimer. L'option `form` permet de spécifier, à l'aide d'une formule, les éventuelles strates de cette dépendance.

```
> update(m.e.i, correlation = corCompSymm(form = ~ 1 | order))
Linear mixed-effects model fit by REML
  Data: LIFEHIST
 Log-restricted-likelihood: -2230.490
 Fixed: litter.size ~ LNmass
(Intercept)          LNmass
 3.8792223    -0.2200541
```

Random effects:

Formula: ~1 | order

(Intercept) Residual

StdDev: 0.7127022 1.350317

Correlation Structure: Compound symmetry

Formula: ~1 | order

Parameter estimate(s):

Rho

2.1646e-19

Number of Observations: 1286

Number of Groups: 17

Il est possible de spécifier que les données sont corrélées sans stratification avec `corCompSymm(form = ~ 1)` : les résultats ne sont pas changés.

La fonction `gls` (*generalized least squares*) dans `nlme` permet d'ajuster un modèle avec données corrélées sans effets aléatoires (`lme` en requiert au moins un) :

```
> m.gls <- gls(litter.size ~ LNmass,  
+ correlation = corCompSymm(form = ~ 1 | order),  
+ data = LIFEHIST, na.action = na.omit)
```

Generalized least squares fit by REML

Model: litter.size ~ LNmass

Data: LIFEHIST

Log-restricted-likelihood: -2230.490

Coefficients:

(Intercept)	LNmass
3.8792223	-0.2200541

Correlation Structure: Compound symmetry

Formula: ~1 | order

Parameter estimate(s):

Rho

0.2178804

Degrees of freedom: 1286 total; 1284 residual

Residual standard error: 1.526860

Ignorer la stratification rend la corrélation entre observations nulle :

```
> update(m.gls, correlation = corCompSymm(form = ~ 1))
```

```
Generalized least squares fit by REML
```

```
Model: litter.size ~ LNmass
```

```
Data: LIFEHIST
```

```
Log-restricted-likelihood: -2326.846
```

```
Coefficients:
```

```
(Intercept)      LNmass
```

```
4.8967068  -0.3061011
```

```
Correlation Structure: Compound symmetry
```

```
Formula: ~1
```

```
Parameter estimate(s):
```

```
      Rho
```

```
3.250077e-19
```

```
Degrees of freedom: 1286 total; 1284 residual
```

```
Residual standard error: 1.472197
```

Le même modèle mais avec observations indépendantes :

```
> (m.gls.cor0 <- update(m.gls, correlation = NULL))
```

```
Generalized least squares fit by REML
```

```
Model: litter.size ~ LNmass
```

```
Data: LIFEHIST
```

```
Log-restricted-likelihood: -2326.846
```

```
Coefficients:
```

```
(Intercept)          LNmass
```

```
4.8967068   -0.3061011
```

```
Degrees of freedom: 1286 total; 1284 residual
```

```
Residual standard error: 1.472197
```

```
> AIC(m.gls); AIC(m.gls.cor0)
```

```
[1] 4468.979
```

```
[1] 4659.691
```

On réajuste les modèles par ML pour faire un LRT :

```

> m.gls.ML <- update(m.gls, method = "ML")
> m.gls.cor0.ML <- update(m.gls.cor0, method = "ML")
> anova(m.gls.ML, m.gls.cor0.ML)

```

	Model	df	AIC	BIC	logLik	Test
m.gls.ML	1	4	4461.805	4482.442	-2226.902	
m.gls.cor0.ML	2	3	4648.245	4663.723	-2321.122	1 vs 2
	L.Ratio		p-value			
m.gls.ML						
m.gls.cor0.ML	188.4402		<.0001			

Il y a 10 structures de corrélation disponibles dans `nlme` (*cf.* `?corClasses`, chaque structure est documentée à sa page propre).

`corSymm` : corrélation générale (sans structure)

3 fonctions pour les corrélations temporelles : `corAR1`, `corARMA`, `corCAR1`

5 fonctions pour les corrélations spatiales : `corExp`, `corGaus`, `corLin`, `corRatio`, `corSpher`

Chacune de ces fonctions a les options `form` pour spécifier le(s) niveau(x) de corrélation, et `fixed = FALSE` pour estimer les paramètres (par défaut).

Il est possible de créer ses propres structures de corrélation (ex. le package `ape` corrélations phylogénétiques).

Variance hétérogène

Fonction de variance \mathcal{V} générale : $Var(\epsilon) = \sigma^2 \mathcal{V}^2(\mu, z, \delta)$


μ : moyenne, z : covariable(s), δ : paramètre(s)

Cinq fonctions de variance dans nlme ($j = 1, \dots, G$) :

<code>varFixed(~z)</code>	$\mathcal{V} = \sqrt{z}$	0 paramètre
<code>varIdent(form=~1 g)</code>	δ_j	$G - 1$ (car $\delta_1 = 1$)
<code>varPower(form=~z)</code>	$ z ^\delta$	1
<code>varPower(form=~z g)</code>	$ z ^{\delta_j}$	G
<code>varConstPower(form=~z)</code>	$\delta_1 + z ^{\delta_2}$	2
<code>varConstPower(form=~z g)</code>	$\delta_{1j} + z ^{\delta_{2j}}$	$2G$
<code>varExp(form=~z)</code>	$e^{\delta z}$	1
<code>varExp(form=~z g)</code>	$e^{\delta_j z}$	G

Excepté pour `varFixed`, `form` n'est pas le premier argument. Dans tous les cas il y a une option `fixed`.

`varComb` combine deux ou plus fonctions de variance en les multipliant.

 L'option `weights` de `lme` ou `gls` attend un objet de classe `varFunc`, alors qu'avec `lm` il s'agit de poids utilisés pour ajuster le modèle par WLS (*weighted least squares*).

Exercice

1. Essayer différentes fonctions de variance avec les données simulées pour tester `VarDisp`.

Corrigé

```
1. > summary(gls(y ~ x, weights = varExp(form = ~ z)))
```

```
.....
```

```
          AIC          BIC      logLik
560.0275  570.3673 -276.0137
```

```
Variance function:
```

```
Structure: Exponential of variance covariate
```

```
Formula: ~z
```

```
Parameter estimates:
```

```
      expon
0.09256942
> summary(gls(y ~ x))
.....
      AIC      BIC    logLik
633.3768 641.1317 -313.6884
> summary(gls(yb ~ x, weights = varExp(form = ~ z)))
.....
      AIC      BIC    logLik
283.939 294.2789 -137.9695
```

Variance function:

```
.....
Parameter estimates:
      expon
-0.006661481
> summary(gls(yb ~ x))
.....
      AIC      BIC    logLik
282.3703 290.1252 -138.1851
```

VIII Régression non-linéaire

Le modèle : $E(y) = f(x_1, x_2, \dots, \theta_1, \theta_2, \dots) = f(x, \theta)$

Les observations : $y_i = f(x_i, \theta) + \epsilon_i \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2)$

```
> n <- 50
> x <- 1:n
> y <- 1/(1 + exp(.1*x)) + rnorm(n, 0, .02)
> plot(x, y)
> plot(log(x), y) # plot(x, y, log = "x")
> plot(sqrt(x), y)

> mod.nl <- nls(y ~ b/(1 + exp(a*x)), start = list(a=10, b=10))
Erreur dans nls(y ~ b/(1 + exp(a * x)), start = list(a = 10, b = 10)) :
  gradient singulier
> mod.nl <- nls(y ~ b/(1 + exp(a*x)), start = list(a=2, b=2))
Nonlinear regression model
```



```
model: y ~ b/(1 + exp(a * x))
data: parent.frame()
      a      b
0.09407 0.94963
residual sum-of-squares: 0.01517
```

```
Number of iterations to convergence: 8
Achieved convergence tolerance: 2.857e-07
```

```
> AIC(mod.nl)
[1] -257.1287
> AIC(mod.sqrtx <- lm(y ~ sqrt(x)))
[1] -194.7126
> AIC(lm(y ~ x))
[1] -148.2175
> AIC(glm(y ~ x, family = gaussian(link = "inverse")))
[1] -159.5486

> plot(x, y)
> lines(x, predict(mod.nl))
```

```
> abline(lm(y ~ x), col = "blue", lty = 2)

> x11()
> plot(sqrt(x), y)
> abline(mod.sqrtx, col = "red", lty = 2)

> layout(matrix(1:2, 2))
> plot(mod.sqrtx, which = 1)
> pred <- predict(mod.nl)
> res <- residuals(mod.nl)
> plot(pred, res)
> lines(lowess(pred, res), col = "red")
```

On notera la différence d'échelle des axes des y (liée à la variance résiduelle).

IX Modèles de “lissage”

Expansion non-linéaire des prédicteurs

Problème : faire entrer X de façon non-linéaire dans le modèle sans avoir besoin de construire un modèle non-linéaire.

Les expansions non-linéaires permettent d'utiliser la machinerie du modèle linéaire. ($\log(X)$, \sqrt{X} font partie de ces méthodes.)

```
> x <- 1:50
> y <- rnorm(50)
> summary(lm(y ~ poly(x, 3, raw = TRUE)))$sigma
> # == y ~ x + I(x^2) + I(x^3)
[1] 1.016576
> summary(lm(y ~ poly(x, 3)))$sigma
[1] 1.016576
> library(splines)
> summary(lm(y ~ bs(x, 3)))$sigma
[1] 1.016576
```

Les coefficients ne sont pas les mêmes mais les prédictions sont identiques.

```
> cor(poly(x, 3, raw = TRUE))
      1      2      3
1 1.0000000 0.9694696 0.9186250
2 0.9694696 1.0000000 0.9861705
3 0.9186250 0.9861705 1.0000000
> cor(poly(x, 3))
      1      2      3
1 1.0000000e+00 1.414206e-17 -1.151965e-17
2 1.414206e-17 1.0000000e+00 1.212274e-17
3 -1.151965e-17 1.212274e-17 1.0000000e+00
> cor(bs(x, 3))
      1      2      3
1 1.00000000 0.1259115 -0.7623014
2 0.1259115 1.0000000 0.1641629
3 -0.7623014 0.1641629 1.0000000
```

`bs` génère une spline (prononcez 'splaine') polynomiale par intervalle (*piecewise polynomial splines*).

```
bs(x, df = NULL, knots = NULL, degree = 3, intercept = FALSE,  
    Boundary.knots = range(x))
```

`df` est ignoré si `knots` est spécifié, il est alors déterminé par `df = length(knots) + degree + intercept`.

Par défaut `bs(x)` génère une spline cubique avec zéro nœud (*knot*), d'où les mêmes résultats qu'avec `poly(x, 3)`.

Autre usage possible, régression linéaire par intervalle :

```
> mod <- lm(y ~ bs(x, 5, degree = 1))  
> plot(x, y)  
> lines(x, predict(mod), col = "blue")
```

Donc `lm(y ~ bs(x, degree = 1))` et `lm(y ~ x)` donnent les mêmes prédictions (avec des paramètres différents).

`poly(x, n)` génère une matrice à n colonnes dont la $i^{\text{ème}}$ est un polynôme de x d'ordre i tel que ces colonnes soient non-corrélées (orthogonales) :

```
> pairs(poly(x, 5))  
> pairs(poly(x, 5, raw = TRUE))
```


Tout savoir sur les polynômes orthogonaux :

en.wikipedia.org/wiki/Orthogonal_polynomials

`ns` ajoute une contrainte de linéarité en dehors de la limite des nœuds.

```
> mod.bs <- lm(y ~ bs(x, 3))  
> mod.ns <- lm(y ~ ns(x, 3))  
> newx <- data.frame(x = -20:70)  
> plot(x, y, xlim = c(-20, 70))
```

```
> lines(newx$x, predict(mod.bs, newdata = newx), col = "red")
Warning message:
In bs(x, degree = 3L, knots = numeric(0), Boundary.knots = c(1L,
  some 'x' values beyond boundary knots may cause ill-conditioning)
> lines(newx$x, predict(mod.ns, newdata = newx), col = "blue")
```

 Il est logique de ne pas considérer séparément les paramètres liés à l'expansion non-linéaire d'un prédicteur.

```
> anova(mod.ns)
Analysis of Variance Table

Response: y
          Df Sum Sq Mean Sq F value Pr(>F)
ns(x, 3)   3  2.717   0.906   0.9216 0.4379
Residuals 46 45.214   0.983
```

Pour utiliser les splines dans des graphiques : `xspline`.

`smooth.spline` : uniquement bivarié, compromis entre la qualité de l'ajustement et le degré de lissage :

$$\sum_{i=1}^n [y_i - f(x_i)]^2 + \lambda \int (f''(x))^2 dx$$

λ : pénalité du terme de lissage

Régression locale

Le principe est simple : calculer les valeurs prédites en utilisant les prédicteurs des points voisins.

`loess` ou `lowess` : polynôme local pondéré par la distance d au point :

$$w_i = \left[1 - \left(\frac{d_i}{\max(d)} \right)^3 \right]^3 \quad \text{si } d_i \leq \alpha, \quad w_i = 0 \quad \text{sinon}$$

span : α

Ainsi que d'autres fonctions "historiques" : `smooth`, `ksmooth`, `supsmu`.

Régression non-paramétrique basée sur un **noyau** (*kernel*) : packages `np` et `sm`.

Modèles généralisés additifs

```
> library(mgcv)
> gam1 <- gam(litter.size ~ s(LNmass), data = LIFEHIST)
> summary(gam1)
```

Family: gaussian

Link function: identity

Formula:

`litter.size ~ s(LNmass)`

Parametric coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	2.79435	0.04068	68.69	<2e-16

Approximate significance of smooth terms:

	edf	Ref.df	F	p-value
s(LNmass)	5.492	5.992	103.1	<2e-16

```
R-sq.(adj) = 0.324    Deviance explained = 32.7%  
GCV score = 2.139    Scale est. = 2.1282    n = 1286
```

```
> plot(gam1, TRUE, shade = TRUE, shade.col = "orange")  
> gam2 <- gam(litter.size ~ s(LNmass) + order, data = LIFEHIST)  
> plot(gam2, TRUE, shade = TRUE, shade.col = "orange")
```

Dans le package `mgcv` :

- terme(s) fixe(s) possible(s)
- terme(s) lisse(s) d'une ou plusieurs variables
- plusieurs types de fonctions lisses, avec ou sans pénalités
- le ddl des fonctions lisses peut être fixé ou estimé
- les fonctions lisses peuvent être reliées à des covariables
- plusieurs fonctions lisses peuvent avoir des paramètres identiques
- possible de construire ses propres fonctions lisses

Le degré de lissage est estimé par GCV (*generalized crossed validation*) ou UBRE (*unbiased risk estimator*) :

$$\text{GCV} = \frac{n \times \text{Dev}}{(n - df_e)^2}$$

$$\text{UBRE} = \frac{\text{Dev} + 2\phi df_e}{n} - \phi$$

UBRE est une variante d'AIC.

cf. `?gam` et `?gam.method`

Exercice

1. Explorer le jeu de données `Boston` de MASS avec des GAM.

Corrigé

1. `str(Boston)` permet de visualiser qu'il y a 14 variables : on s'intéressera à la dernière en guise de réponse (*cf.* `?Boston`). Quelques essais montrent que

les colonnes 4 et 9 amènent à une erreur avec `gam`. Ci-dessous un essai avec les autres variables qui sont ajustées successivement et les graphes dessinés sur la même figure.

```
layout(matrix(1:12, 4))
par(mar = c(5, 4, 0, 0))
for (i in c(1:3, 5:8, 10:13))
  plot(gam(Boston[, 14] ~ s(Boston[, i])),
       xlab = colnames(Boston)[i])
```

Le package `gamlss` propose des GAM pour un grand nombre de distribution, y compris une modélisation de la dispersion.

Autres packages sur CRAN : `gam`, `GAMBoost`, `COZIGAM`, `pgam`, `VGAM`

X Séries temporelles

L'analyse des séries temporelles est marquée par :

- des structures de données particulières,
- la dépendance temporelle des observations.

cran.r-project.org/web/views/TimeSeries.html (CRAN → Task Views → TimeSeries) donne une vue d'ensemble intéressante sur la question.

Structures de données temporelles

Classe "ts" pour les séries régulières.

```
> data(sunspots)
```

```
> str(sunspots)
```

```
Time-Series [1:2820] from 1749 to 1984: 58 62.6 70 55.7 85 83.1  
66.3 75.9 75.5 ...
```

```
> tsp(sunspots)
[1] 1749.000 1983.917 12.000

> n <- 120
> x1 <- cos(12 * 1:n) + rnorm(n, 0, 0.2)
> x2 <- rnorm(n)
> X <- ts(cbind(x1, x2), start=2001, end=2010, frequency=12)
> str(X)
mts [1:109, 1:2] 0.9966 0.0473 -0.1193 -0.735 -0.9781 ...
- attr(*, "dimnames")=List of 2
..$ : NULL
..$ : chr [1:2] "x1" "x2"
- attr(*, "tsp")= num [1:3] 2001 2010 12
- attr(*, "class")= chr [1:2] "mts" "ts"
> tsp(X)
[1] 2001 2010 12
> plot(X)
```

Classe "Date" pour les dates.

`as.Date` : transforme une chaîne de caractères en objet "Date".

```
> x <- "2009-02-05"
> d <- as.Date(x, "%Y-%m-%d")
> x; d
[1] "2009-02-05"
[1] "2009-02-05"
> str(x); str(d)
chr "2009-02-05"
Class 'Date' num 14280
> b <- as.Date("05/02/09", "%d/%m/%y")
> b
[1] "2009-02-05"
> str(b)
Class 'Date' num 14280
```

à essayer avec :


```
> x <- c("14071789", "05022009")
> x <- as.Date(x, "%d%m%Y")
> plot(x, c(10, 100)) # voir l'axe des x
> str(x)
Class 'Date'   num [1:2] -65914 14280
> diff(x) # == x[2] - x[1]
Time difference of 80194 days
```

La syntaxe résumée :

- %d jour du mois (01–31)
- %m mois (01–12)
- %Y année (4 chiffres)
- %y année (2 chiffres) à éviter !
- %a nom du jour abrégé*
- %A nom du jour complet*
- %b nom du mois abrégé*
- %B nom du mois complet*

*système-dépendant (*locale*) mais “partial matching”

Les éléments manquants dans l'entrée sont pris dans l'horloge de la machine :

```
> as.Date("", "") == as.Date("17", "%d")
```

```
[1] FALSE
```

```
> as.Date("", "") == as.Date("05", "%d")
```

```
[1] TRUE
```

Plus compliqué pour le temps, mais l'idée se généralise :

```
> d1 <- strptime("05-02-2009 13:30:00", "%d-%m-%Y %H:%M:%S")
```

```
> d2 <- strptime("05-02-2009 13:30:30", "%d-%m-%Y %H:%M:%S")
```

```
> d2 - d1
```

```
Time difference of 30 secs
```

Pour l'opération inverse :

```
> format(x, "%A %d %B %Y")  
[1] "mardi 14 juillet 1789" "jeudi 05 février 2009"
```

Plus de détails :

?strptime # détails sur syntaxe...

?Date

?as.Date

Autocorrélation

```
> acf(sunspots, lag.max = 200)
> pacf(sunspots, lag.max = 200)
> acf(X)
> pacf(X)
```

Sous H_0 : pas d'autocorrélation, la valeur attendue est > 0 en valeur absolue.

```
> acf.X <- acf(X, plot = FALSE)
> plot(acf.X)
```

Autorégression

Le modèle autorégressif le plus simple AR(1) : $x_t = ax_{t-1} + \epsilon_t$ avec x centré et $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$.

```
> ar(sunspots, order.max = 1)
```

Call:

```
ar(x = sunspots, order.max = 1)
```

Coefficients:

```
1  
0.9217
```

```
Order selected 1 sigma^2 estimated as 284.2
```

Modèle auto-régressif général d'ordre p :

$$x_t = \sum_{i=1}^p a_i x_{t-i} + \epsilon_t$$

```
> ar(sunspots)
```

```
Call:
```

```
ar(x = sunspots)
```

```
Coefficients:
```

1	2	3	4	5	6
0.5400	0.0962	0.0789	0.0897	0.0410	0.0487
7	8	9	10	11	12
0.0095	0.0124	0.1073	0.0189	0.0098	0.0253
13	14	15	16	17	18
-0.0136	0.0030	0.0431	-0.0364	0.0000	-0.0602
19	20	21	22	23	24
0.0022	-0.0181	-0.0426	-0.0107	0.0541	-0.0821
25	26	27	28		
0.0701	-0.0039	-0.0345	-0.0286		

```
Order selected 28  sigma^2 estimated as 233.8
```

Un modèle simple à moyenne mobile MA(1) : $x_t = \epsilon_t + b\epsilon_{t-1}$

```
> arima(sunspots, c(0, 0, 1))
```

Call:

```
arima(x = sunspots, order = c(0, 0, 1))
```

Coefficients:

```
          ma1  intercept
          0.7233    51.2700
s.e.      0.0098    0.9415
```

```
sigma^2 estimated as 842:  log likelihood = -13499.11,  aic = 21
```

Modèle général ARMA(p,q) :

$$x_t = \sum_{i=1}^p a_i x_{t-i} + \sum_{j=0}^q b_j \epsilon_{t-j}$$

```
> arima(sunspots, c(0, 0, 0))
```

```
Call:
```

```
arima(x = sunspots, order = c(0, 0, 0))
```

```
Coefficients:
```

```
      intercept
```

```
      51.266
```

```
s.e.      0.818
```

```
sigma^2 estimated as 1887:  log likelihood = -14636.78,  aic = 1887.813
```

```
> mean(sunspots); var(sunspots)
```

```
[1] 51.26596
```

```
[1] 1887.813
```

Modèle ARIMA(p,d,q) : modèle ARMA(p,q) ajusté à x différenciée d'ordre d ($x_t - x_{t-d}$).


```
> args(arima)
```

```
function (x, order = c(0, 0, 0), seasonal = list(order = c(0, 0, 0, 0),  
  period = NA), xreg = NULL, include.mean = TRUE, transform.pars =  
  fixed = NULL, init = NULL, method = c("CSS-ML", "ML", "CSS"),  
  n.cond, optim.control = list(), kappa = 1e+06)
```

Comparaison MA(1) et ARMA(1,1) :

```
> tsdiag(arima(sunspots, c(0, 0, 1)))
```

```
> tsdiag(arima(sunspots, c(1, 0, 1)))
```

```
> plot(stl(sunspots, 1))
```

```
> plot(stl(sunspots, 10))
```

```
> plot(stl(X[, "x1"], 12))
```

```
> plot(stl(X[, "x2"], 12))
```

Exercice

1. Tester `arima` et ses options sur les données simulées X .

Modèles non-linéaires : packages `tseriesChaos` et `tsDyn`

De “R pour les débutants” (modifié) :

```
ricker <- function(nzero, r, K=1, time=100, from=0, to=time)
{
  N <- numeric(time+1)
  N[1] <- nzero
  for (i in 1:time) N[i+1] <- N[i]*exp(r*(1 - N[i]/K))
  N
}

> out <- ricker(0.1, 3, 1, 1e3)
> library(tseriesChaos)
> a <- embedd(out, lags = 1:3)
```

```
> library(scatterplot3d)
> scatterplot3d(a, type = "l", angle = 90)
```

```
> library(tsDyn)
```

```
.....
```

```
> star(out, noRegimes = 2)
```

```
Using default threshold variable: thDelay=0
```

```
Testing linearity... p-Value = 0
```

```
The series is nonlinear. Incremental building procedure:
```

```
Building a 2 regime STAR.
```

```
Using default threshold variable: thDelay=0
```

```
Performing grid search for starting values...
```

```
Starting values fixed: gamma = 10 , th = 0.4561601 ; SSE = 1
```

```
Optimization algorithm converged
```

```
Finished building a MRSTAR with 2 regimes
```

```
Non linear autoregressive model
```

Multiple regime STAR model

Regime 1 :

Linear parameters: 1.4758817, 79916.112111, -0.8849242

Regime 2 :

Linear parameters: -1.4756764, -79916.1121963, 0.8849467

Non-linear parameters:

3.0002236, -2.7626793

XI Analyses spatiales

Les structures de données spatiales sont par essence plus complexes :

```
> library(geoR)
> plot(sic.367, borders = sic.borders, scatter3d = TRUE)
> plot(sic.367)
> names(sic.367)
$coords
[1] "v2" "v3"

$data
[1] "data"

$other
[1] "altitude"

> args(read.geodata)
function (file, header = FALSE, coords.col = 1:2, data.col = 3,
```

```
data.names = NULL, covar.col = NULL, covar.names = "header",  
units.m.col = NULL, realisations = NULL, na.action = c("ifany",  
"ifdata", "ifcovar", "none"), rep.data.action, rep.covar.action,  
rep.units.action, ...)
```

$Z(x)$: 'processus', x : système de coordonnées, $Z(x_i)$: observations

Au contraire d'un processus temporel, ici $Z(x)$ est généralement continu.

Variogramme

Semivariogramme γ :

$$\gamma(x, y) = \frac{1}{2}E([Z(x) - Z(y)]^2) = \frac{1}{2}[C(x, x) + C(y, y)] - C(x, y)$$

avec l'(auto)covariance $C(x, y) = cov(\epsilon(x), \epsilon(y))$ donc $\gamma(x, y) = c(0) - C(x, y)$ donc $\gamma(x, x) = 0$.

Si Z est stationnaire : $\gamma_s(x - y) = \gamma(x, y)$

Si Z est isotrope : $\gamma_i(d(x, y)) = \gamma(x, y)$

Variogramme empirique :

$$\hat{\gamma}(h) = \frac{1}{n_H} \sum_{(x_i, x_j) \in H} [Z(x_i) - Z(x_j)]^2$$

H est un ensemble de points défini par h , n_H : nombre de points dans H

```
> X <- expand.grid(x = 1:10, y = 1:10)
> X$z <- rnorm(100)
> Xgeo <- as.geodata(X)
> plot(evg.X <- variog(Xgeo, uvec = 10))
```

Nombreuses informations retournées : *cf.* `?variog` pour les détails.

La fonction `variogram` du package `spatial` est différente : les données doivent être ajustées à une surface au préalable, et les limites des intervalles ne peuvent pas être définies :

```
> library(spatial)
> X11()
> variogram(surf.ls(2, X), 10, ylim = c(0, 1.2))
```

Variogramme théorique :

geoR modélise l'autocovariance $C(h) = \sigma^2 \rho(h)$

$\rho(h)$: fonction d'autocorrélation

?cov.spatial documente 13 fonctions, plus la possibilité d'emboîter des modèles. (Si vous êtes perdus : `eyefit(evg.X)`.)

Exercice

1. Ajuster différents modèles de variogramme aux données `topo` du package MASS. On tracera les graphiques appropriés.

Corrigé

```
1. library(MASS)
   data(topo)
   X <- as.geodata(topo)
   vgx <- variog(X)
   mod <- c("cubic", "exponential", "spherical", "wave")
   co <- c("blue", "red", "yellow", "black")
   plot(vgx)
```

```
for (i in 1:length(mod))
  lines(variofit(vgx, cov.model = mod[i]), col = co[i])
legend("topleft", mod, lty = 1, col = co)
```

Pour faire les graphes séparés sur la même figure :

```
layout(matrix(1:4, 2))
for (i in 1:length(mod)) {
  plot(vgx, main = mod[i])
  lines(variofit(vgx, cov.model = mod[i]))
}
```

Kriging

kriging : méthode d'interpolation spatiale

$$\hat{Z}(x_0) = \sum_{i=1}^n w_i(x_0) Z(x_i)$$

Les poids $w_i(x_0)$ sont choisis afin de minimiser la variance de kriging :

$$\sigma^2(x_0) = \text{Var}(Z(x_0) - \hat{Z}(x_0))$$

Types de kriging dans geoR :

1. simple "SK" $E(Z(x)) = 0$
2. ordinaire "OK" $E(Z(x)) = \mu$
3. tendance externe "KTE"
4. universel "UK" $E(Z(x)) = \beta f(x) + \epsilon(x)$

`krige.conv` ajuste le kriging

`trend.spatial` définit la matrice de tendance spatiale

Exercice

1. On prendra les données `topo` du package `MASS`. Dessiner une carte en couleur de la “zone” après kriging.

Corrigé

```
1. > vf <- eyefit(vgx)
   variog: computing omnidirectional variogram
> vf
   cov.model          sigmasq  phi  tausq  kappa  kappa2
1 spherical 5313.56825396825 6.24    0  <NA>  <NA>
   practicalRange
1              6.24
> sq <- seq(0, 6, 0.1)
> loc <- expand.grid(x = sq, y = sq)
```

```
> kgctrl <- krige.control(obj.model = vf)
> kg <- krige.conv(X, locations = loc, krige = kgctrl)
krige.conv: model with constant mean
krige.conv: Kriging performed using global neighbourhood
> image(kg)
```

L'objet retourné par `krige.conv` est décrit sur la page d'aide correspondante et sa structure peut être affiché comme pour tout objet :

```
> str(kg)
List of 6
 $ predict      : num [1:3721] 923 924 925 925 924 ...
 $ krige.var    : num [1:3721] 1448 1313 1195 1101 1033 ...
 $ beta.est     : Named num 855
 ..- attr(*, "names")= chr "beta"
 $ distribution: chr "normal"
 $ message      : chr "krige.conv: Kriging performed using gl
 $ call        : language krige.conv(geodata = X, locations
- attr(*, "sp.dim")= chr "2d"
- attr(*, "prediction.locations")= symbol loc
```

- `attr(*, "parent.env")=<environment: R_GlobalEnv>`
 - `attr(*, "data.locations")= language X$coords`
 - `attr(*, "class")= chr "kriging"`
-

Ajustement de surfaces :

Dans `spatial` : `surf.ls` et `surf.gls` ajustent des surfaces par moindres carrés (généralisés).

`geoR` `likfit` ajuste une surface polynomiale par ML.

`spdep` : `lagsarlm`, `spautolm`, ...

Pour la cartographie : `maps`, `mapdata`, `mapproj`, `maptools`.