# Brise Glace-R
# (ouvrir la voie aux pôles statistiques)

Andrew Robinson[1]    Arnaud Schloesing[2]

août-septembre 2008

1. Depatment of Mathematics and Statistics, University of Melbourne, Parkville, Vic. 3010.
2. Centre Intégré de Bioinformatique, Université Lille II

# Table des matières


**1 Introduction** 7

1.1 **R** . . . 7

1.2 Hello World . . . 7

1.3 Pourquoi **R** ? . . . 8

1.4 Pourquoi pas **R** ? . . . 8

1.5 L'idéal du logiciel libre . . . 8

1.6 Utilisation de ce document . . . 9

1.7 Soyez organisés . . . 9

1.8 Démarrer . . . 9

1.9 Stopper . . . 9

1.10 Répertoire de travail . . . 9

1.11 Conventions . . . 10

**2 Démarrage Rapide** 11

**3 Étude de cas** 13

3.1 À vos marques . . . 13

3.2 Entrée des données . . . 13

3.3 Structure des données . . . 13

3.3.1 Manipulation . . . 13

3.3.2 Récapitulatifs des données . . . 14

3.3.3 Tableaux Pivots . . . 14

3.4 Modèles . . . 15

3.5 Fonctions . . . 15

3.6 Traitement au niveau parcelle . . . 16

**4 Infrastructure** 19

4.1 Obtenir de l'aide . . . 19

4.1.1 En local . . . 19

4.1.2 En déporté . . . 20

4.2 Écrire des scripts . . . 20

4.3 Espaces de travail . . . 21

4.4 Historique . . . 21

4.5 Extensibilité . . . 22

**5 Interface** **25**
5.1 Importer et exporter des données . . . 25
5.1.1 Importer . . . 25
5.1.2 Exporter . . . 27

**6 Données** **29**
6.1 Données : objets et classes . . . 29
6.2 Classes des données . . . 30
6.2.1 Numeric . . . 30
6.2.2 String . . . 30
6.2.3 Factor . . . 30
6.2.4 Logical . . . 31
6.2.5 Données manquantes . . . 31
6.3 Structures des données . . . 32
6.3.1 Vector . . . 32
6.3.2 Data-frame . . . 33
6.3.3 Matrice (Tableau) . . . 35
6.3.4 Liste . . . 36
6.4 Fusion des données . . . 37
6.5 Réadaptation des données . . . 38
6.6 Tri des données . . . 39

**7 Programmation** **41**
7.1 Fonctions . . . 41
7.2 Contrôle de flux . . . 42
7.2.1 Étude de cas . . . 42
7.2.2 Boucles et autres choses . . . 43
7.3 Portée . . . 44
7.4 Déverminage . . . 44
7.5 Classes S3 et objets . . . 45
7.6 Autres langages . . . 48
7.6.1 Écriture . . . 50
7.6.2 Compilation . . . 50
7.6.3 Attachement . . . 50
7.6.4 Appel . . . 51
7.6.5 Bénéfice ! . . . 51

**8 Simples descriptions** **53**
8.1 Une seule variable . . . 53
8.1.1 Quantitative . . . 53
8.1.2 Qualitative . . . 55
8.2 Plusieurs variables . . . 56
8.2.1 Quantitative/quantitative . . . 57
8.2.2 Quantitative/qualitative . . . 57
8.2.3 Qualitative/qualitative . . . 57

**9 Graphiques** 59
9.1 Paramètres d'organisation 60
9.2 Agrémentation d'un graphique 61
9.3 Permanence 62
9.4 Mises à jour 63
9.5 Défis courants 64
9.5.1 Barres d'erreur 64
9.5.2 Graphiques colorés par groupe 64
9.6 Contributions 65
9.6.1 Treillis 65
9.7 Grammaire des graphiques 68

**10 Régression linéaire** 69
10.1 Préparation 69
10.2 Ajustement 70
10.3 Diagnostics 71
10.4 Autres outils 73
10.5 Examiner le modèle 73
10.6 Autres angles 73
10.7 Autres modèles 74
10.8 Pondération 74
10.9 Transformations 75
10.9.1 Une étude de cas 76
10.10 Tester des effets spécifiques 80
10.11 Contributions 81
10.12 Autres moyens d'ajustement 81

**11 Modèles hiérarchiques** 83
11.1 Introduction 83
11.1.1 Méthodologique 83
11.1.2 Général 84
11.2 De la théorie 84
11.2.1 Les effets 84
11.2.2 Construction du modèle 85
11.2.3 Dilemme 87
11.2.4 Décomposition 88
11.3 Un exemple simple 89
11.3.1 La toute fin 94
11.3.2 Maximum de vraisemblance 94
11.3.3 Maximum de vraisemblance restreint 95
11.4 Étude de cas 95
11.4.1 Données stage 95
11.4.2 Extensions au modèle 112
11.5 Le modèle 113
11.5.1 Z 114
11.5.2 b 114

11.5.3 D . . . 114
11.6 Disputes . . . 114
11.6.1 Contrôler . . . 115
11.6.2 Toucher à tout . . . 115
11.6.3 Modifier . . . 115
11.6.4 Compromis . . . 115
11.7 Annexe : les diagnostics Leave-One-Out . . . 116

**12 Modèle non linéaire** **119**
12.1 Les modèles non linéaires . . . 119
12.1.1 Un seul arbre . . . 121
12.1.2 Plusieurs arbres . . . 123
12.2 Splines . . . 128
12.2.1 Splines cubiques . . . 128
12.2.2 Modèle additif . . . 129
12.3 Hiérarchique . . . 130

# Chapitre 1

# Introduction

## 1.1 R

**R** est un langage de programmation pour l'analyse et la modélisation des données. **R** peut être utilisé comme un langage orienté objet tout comme un environnement statistique dans lequel des listes d'instructions peuvent être exécutées en séquence sans l'intervention de l'utilisateur.

On peut interagir avec **R** en tapant ou collant des instructions sur la ligne de commande de la console ou en les saisissant dans un éditeur de texte [1] et les faisant lire par la console avec la commande `source()`.

L'utilisation exclusive de l'interface ligne de commande (CLI) rend notre temps d'apprentissage plus long. Cependant, cela nous permet aussi de stocker une collection de commandes et de les exécuter sans autre intervention. Cela simplifie beaucoup d'opérations et de tâches, par exemple pour communiquer une analyse à une autre personne ou faire des représentations graphiques ou des comptes rendus d'utilisation fréquente. Un avantage principal de la CLI est qu'elle simplifie le développement et l'utilisation de scripts. Cela nous permet de conserver un texte enregistré permanent et documenté des étapes que nous avons suivies.

Une interaction avec **R** peut être effectuée au travers de menus pour certains systèmes d'exploitation, mais cette interaction est principalement administrative. Des utilisateurs ont écrit des interfaces utilisateurs graphiques (GUI) [2], mais elles ne seront pas traitées ici. On peut trouver et télécharger librement les exécutables et le code source à :

http://www.r-project.org.

## 1.2 Hello World

La fonction imprimant « Hello World » est à la base de toute présentation de langage de programmation.
En **R** cela donne :

```
> hi.there <- function() cat("Hello World!\n")
> hi.there()

Hello World!
```

L'inclusion de `Hello World` dans ce document est, au moins partiellement, ironique, car ce programme n'a que bien peu de ressemblance avec les fonctionnalités réelles de **R** [3] Sans doute le suivant est-il plus approprié.

```
> 1 + 1

[1] 2
```

1. Bluefish sous Linux, TinnR sous Windows, Emacs sous les deux SE.
2. On en saura plus à ce sujet sur http://www.r-project.org/GUI
3. C'est aussi vrai pour la plupart des langages, aussi suis-je discret.

## 1.3 Pourquoi R ?

1. **R** fonctionne sous Windows, Mac-OS et les variantes Unix (BSD et les différents Linux).
2. **R** procure un grand nombre d'outils statistiques utiles, la plupart d'entre eux ont été testés avec rigueur.
3. **R** fournit des graphiques de qualité publiable dans de nombreux formats, jpeg, postscript, eps, pdf et bmp, avec une interface souple et facilement extensible.
4. **R** s'adapte bien à LaTeXavec le paquetage `Sweave`.
5. **R** s'adapte bien au FORTRAN, au C et aux scripts shell.
6. **R** s'adapte aussi bien aux petits qu'aux grands projets.

**Anecdote :** J'étais en télécommunication de New Haven (CT) à Moscou (ID). J'avais développé et testé le code d'une simulation sur mon micro portable, effectué *ssh-ed* sur un serveur FreeBSD à Moscou et lancé le code complet à l'intérieur d'une session screen [4]. Chaque fois que je voulais en contrôler le progrès je pouvais le faire en me connectant à la session à distance de n'importe où. Quand j'ai enfin terminé, **R** m'envoie par mail le compte rendu.

## 1.4 Pourquoi pas R ?

1. **R** ne peut pas tout faire.
2. **R** ne vous tient pas la main.
3. La documentation peut être opaque.
4. **R** peut vous énerver ou vous vieillir prématurément.
5. Les paquetages des contributeurs ont été soumis à des degrés variés de test et d'analyse. Certains ne sont peut-être pas fiables.
6. Il n'y a pas de garantie de meilleur si l'on paye pour.
7. **R** n'est pas un clicodrome.

**Anecdote :** Je développais une analyse de données à relativement grande échelle, requérant de nombreuses étapes. L'une d'elles consistait à appliquer 180 fois un modèle de croissance d'une forêt (ORGANON), chacune avec une configuration différente, montrer les résultats et les assimiler. J'avais écrit le code **R** pour faire tout ça : produit 180 fichiers de configuration, exécuté un préprocesseur 180 fois, sauvegardé les résultats, exécuté le modèle 180 fois, importé les simulations et les ai manipulées comme des data-frames. Comme cela devait être exécuté de nombreuses fois et requérait des fichiers temporaires identifiables, je décidais d'y inclure des étapes de mise au propre, comprenant la suppression de certains fichiers. Si l'on exécute ce code en le recopiant sur le document, le collant comme un groupe de commandes à la console et qu'une commande échoue, l'interpréteur attaquera néanmoins la suivante. La commande ayant échoué était un changement de répertoire. Mon script a continué à se faire des appels à lui-même et ses scripts compagnons. C'était un cas classique où la puissance d'un outil est supplantée par l'incompétence de son utilisateur. **R** est très puissant.

## 1.5 L'idéal du logiciel libre

**R** est libre, c'est-à-dire que l'on peut télécharger les exécutables et le code source sans rien payer. Cependant, ce n'est pas le volet le plus important de la liberté à l'égard du logiciel. Il y a aussi la liberté due au manque de contraintes.

La phrase la plus souvent utilisée pour décrire ce principe particulier de la liberté est : « Pensez à un libre discours plutôt qu'à une tournée gratuite. » On peut apporter toute les modifications désirées avec une seule obligation : toute distribution future doit être sous la même licence et inclure le code source. Mais l'on peut toujours payer ou faire payer un coût raisonnable pour sa distribution.

Il y a de nombreuses présentations pour l'« Open Source », qui sont généralement regroupées, par erreur. Le libellé fait référence à la licence sous laquelle le code est fourni oui ou non au domaine public. Il y a au moins trois représentations distinctes de licence pour les produits Open Source : GNU, BSD et OSI.

**R** est fourni sous licence GNU qui, en bref, dit que le source est librement disponible et qu'on peut le modifier comme on veut, mais que si on le *distribue*, toute modification doit alors être accompagnée du code source modifié et distribué sous la même licence.

Cela signifie aussi que si l'on apporte des changements et qu'on ne les distribue pas, l'on n'est pas obligé de partager le code source avec qui que ce soit.

4. Screen est multiplexeur de terminal open-source très utile.

## 1.6 Utilisation de ce document

Ce document est au format pdf. L'Acrobat Reader donne la possibilité de copier tout texte dans le presse-papiers, en choisissant l'outil Select text, en sélectionnant le texte voulu et en frappant les touches Ctrl et C simultanément. On peut alors le coller sur la console **R** : sous Windows, avec Ctrl + V ou en cliquant sur le bouton droit de la souris et choisissant l'option coller, sous Linux, en collant directement la sélection avec le bouton central de souris. Cela évite beaucoup de frappe pénible. Les commandes **R** sont écrites en incliné dans une police de machine à écrire. Cela peut être un peu déroutant, par exemple, quand une barre verticale | est utilisée et apparaît comme un slash. Copier et coller est plus sûr que de saisir soi-même. Aussi une commande qui occupe plus d'une seule ligne est-elle reliée par le signe +, qu'il n'est pas nécessaire de taper en transcription du code ; cela crée des erreurs.

Aussi tout texte de ce document peut-il être copié et collé dans des documents textes.

Si vous choisissez de saisir les commandes à la console, notez quand-même que j'ai été plutôt paresseux vis-à-vis d'une importante convention : finissez toujours votre saisie en utilisant la touche Entrée.

Les jeux de données utilisés dans ce document peuvent être obtenus à l'adresse :

http://www.ms.unimelb.edu.au/~andrewpr/r-users/

## 1.7 Soyez organisés

La structure des fichiers est une affaire de goût et de préférence. Je préfère vouer un seul répertoire pour chaque projet. Chacun de ces répertoires contient plusieurs des sous-répertoires suivants : `data`, `documents`, `graphiques`, `images`, `scripts`, `src` et `sweave`. L'avantage est que **R** permet le libellé relatif des répertoires, aussi sais-je que du répertoire de script, les données sont toujours dans `../data`, les graphiques dans `../graphiques` et les images toujours dans `../images`. Cette structure rend l'écriture des comptes rendus aussi facile que possible.

## 1.8 Démarrer

Les différents systèmes d'exploitation interagissent avec **R** de façons différentes. Certains requièrent un double-clic sur l'icone approprié, certain que l'on frappe sur R et sur Entrée.

**R** possède un très grand nombre de paquetages à ajouter pour avoir différents outils statistiques et graphiques. Quelques uns d'entre eux sont chargés en mémoire au démarrage. Nous approfondirons cette question en section 4.5, mais pour le moment, il suffit de savoir qu'une quantité raisonnable de fonctionnalités est disponible, et que quand il en faut plus, on l'obtient facilement.

## 1.9 Stopper

Utilisez Ctrl+C ou Échap pour stopper le processus. Cela fonctionnera la plupart du temps. On n'aura guère à appeler le système d'exploitation pour y parvenir. Quitter **R** en même temps en tapant `q()`.

## 1.10 Répertoire de travail

Le répertoire de travail est la place par défaut dans et de laquelle **R** écrit et lit des fichiers. Si l'on veut lire ou écrire à un autre endroit, on doit le dire explicitement. La vie est de ce fait plus facile si toutes les données et les scripts sont gardés au même endroit (fréquemment sauvegardé !).

Sous Windows, il existe un item de menu qui permet de choisir le répertoire de travail aussi bien qu'en ligne de commande. Dans les versions CLI de **R**, comme sous Linux, on ne peut utiliser que les instructions de ligne de commande :

```
> getwd()               # Quel est le répertoire de travail ?
> setwd("~/R/Travail")  # À fixer comme répertoire de travail.
```

Les slashes sont utilisés quel que soit le système d'exploitation sous-jacent. Cela peut sembler curieux sous Windows, mais c'est en fait une propriété, un niveau supplémentaire de portabilité du code.

## 1.11 Conventions

Il y a plusieurs façons de faire les choses en **R**. Il n'y a pas de conventions officielles sur la façon dont doit-être utilisé le langage, mais les indications suivantes peuvent se trouver utiles dans la communication avec les grands utilisateurs de **R**.

1. Bien que le signe égal « `=` » fonctionne pour une affectation, il est aussi utilisé pour d'autres choses, par exemple dans le passage d'arguments. La flèche « `<-` » n'est, elle, utilisé que pour l'affectation. Utilisez-la s'il vous plaît.
2. Les espaces ne coûtent pas cher. Utilisez-les librement entre les arguments et entre objets et opérateurs arithmétiques.
3. Donnez des noms utiles aux objets. N'appelez pas votre modèle `modele` ou votre data-frame `data`.
4. Vous pouvez terminer vos lignes par un point-vigule,mais ne le faites pas.

Mauvais :

```
> constante=3.2808399;
> x=x*constante;
```

Bon :

```
> pied-par-metre <- 3.2808399
> hauteur.m <- hauteur.p * pied-par-metre
```

# Chapitre 2

# Démarrage Rapide

**Interaction :** Si vous apprenez **R** à partir d'un fichier pdf comme celui-ci, vous pouvez exécuter très facilement les commandes en sélectionnant du texte (autant que vous voulez), le copiant dans le presse-papiers et le collant sur la console avec les raccourcis clavier ou les boutons de souris.

**Saisie des données :** Obtenir vos données dans **R** depuis un fichier est relativement facile. Vous devez faire deux choses :

1. soyez sûrs que **R** sait où trouver les données. Le faire en stockant les données dans le *répertoire de travail* (cf. Section 1.10) ou en stockant le fichier quelque part où il est facile à trouver, comme votre lecteur de base, et en faire soit votre répertoire de travail, soit de dire à **R** où le trouver (cf. Section 5.1.1).
2. soyez sûrs que **R** connaît le type de vos données, en choisissant la fonction appropriée pour les lire – utilisez `read.csv()` pour des fichiers à séparateurs virgule (cf. Section 5.1.1).

**Structure des données :** Utiliser la commande `str()` pour connaître la structure de l'objet données.

**Tracés :** Des diagrammes de dispersion des données peuvent être construits avec la fonction `plot()` (cf. Chapitre 9).

# Chapitre 3

# Étude de cas

Le but de ce chapitre est de faire un tour de chauffe sur les possibilités de **R**, en suivant un chemin relativement cohérent. Copiez les lignes de code rencontrées dans la console **R** et regardez ce qui se passe à l'exécution. Essayez de faire un compte rendu de ce que vous voyez au cours de vos analyses journalières.

*Ne vous inquiétez pas trop pour le pas-à-pas de l'exécution.* Une partie de ce que vous voyez vous est familière, une autre pas. Plus tard, nous répéterons la plupart de ces étapes dans différents contextes.

## 3.1 À vos marques

Assurez-vous que le jeu de données `ufc.csv` est placé dans un répertoire nommé `data`. Ensuite, démarrez **R** de la manière appropriée à votre système d'exploitation. Enfin, faire de ce répertoire votre répertoire de travail. Alors, le code qui suit devrait fonctionner sans autre intervention.

## 3.2 Entrée des données

D'abord, nous lisons les données sous la forme d'un fichier à séparateur virgule. Les données sont celles de situation des plantations sur une parcelle de 300 ha « Experimental Forest » de l'University of Idaho, nommée plantation d'Upper Flat Creek. Le plan d'échantillonnage était celui d'un simple échantillon systématique d'une variable CBS (Coefficient de Biotope par Surface, BAF en anglais) de 7 $m^2/ha$ ; le diamètre est mesuré pour chaque arbre et la hauteur pour chaque lot. Dans le jeu de données, certaines plantations sont absentes. Elles sont notées par des arbres sans diamètre mesuré et sans spécification d'espèce.

```
> ufc <- read.csv("../data/ufc.csv")
> ufc.baf <- 7
> ufc.surface <- 300
```

## 3.3 Structure des données

Nous allons examiner la `str()`ucture des données, contrôler les premières lignes et compter les données manquantes par colonne.

```
> str(ufc)
> head(ufc)
> colSums(is.na(ufc))
```

### 3.3.1 Manipulation

Ah, quelques diamètres manquent. Ils correspondent aux plantations absentes. De même, plusieurs hauteurs manquent ; l'attribut hauteur a du être sous-échantillonné. Ensuite faut-il obtenir des unités familières pour nos mesures d'arbre (cm pour diamètre et m pour la hauteur, respectivement).

```
> ufc$haut.m <- ufc$height/10
> ufc$diam.cm <- ufc$dbh/10
> ufc$especes <- ufc$species
```

À quoi cela ressemble t'il ?

```
> str(ufc)
```

### 3.3.2 Récapitulatifs des données

Maintenant on peut obtenir des instantanés utiles des données.

```
> range(ufc$diam.cm)
> range(ufc$haut.m, na.rm = TRUE)
```

Les hauteurs nulles posent un problème. Oublions-les.

```
> ufc$haut.m[ufc$haut.m < 0.1] <- NA
> range(ufc$haut.m, na.rm = TRUE)
```

Comment les espèces se présentent-elles ?

```
> table(ufc$especes)
```

Les F(ir) et FG( ?) sont probablement des GF (Grand Fir). Effectuons-en le changement.

```
> ufc$especes[ufc$especes %in% c("F","FG")] <- "GF"
> ufc$especes <- factor(ufc$especes)
> table(ufc$especes)
```

Combien de hauteurs manquantes avons-nous pour chaque espèce ?

```
> table(ufc$especes[is.na(ufc$haut.m)]
```

Ces récapitulatifs utilisent les outils graphiques standard pour l'analyse exploratoire.

```
> boxplot(diam.cm ~ especes, data = ufc)
> boxplot(haut.m ~ especes, data = ufc)
> scatter.smooth(ufc$diam.cm, ufc$haut.m)
```

On peut aussi utiliser des outils graphiques plus développés.

```
> require(MASS)
> truehist(ufc$diam.cm, h = 2.5, col = "darkseagreen3")

> require(lattice)
> xyplot(haut.m ~ diam.cm | especes, data = ufc)
> histogram(~ diam.cm | especes, data = ufc)
```

### 3.3.3 Tableaux Pivots

Ce ne sont pas des « Tirés-Lachés » mais, ici encore, ils sont faciles à construire et la sortie plus simple à manipuler.

```
> tapply(ufc$diam.cm, ufc$especes, length)
> tapply(ufc$diam.cm, ufc$especes, mean)
```

## 3.4 Modèles

Examinons attentivement la relation entre hauteur et diamètre, en utilisant de simples outils de modélisation. Commençons avec la régression linéaire.

```
> hd.lm.1 <- lm(haut.m ~ diam.cm, data = ufc)
> summary(hd.lm.1)
> scatter.smooth(ufc$diam.cm, ufc$haut.m)
> abline(hd.lm.1, col = "red")
```

Ajustons par espèce et comparons les régressions au modèle précédent.

```
> hd.lm.2 <- lm(haut.m ~ diam.cm * especes, data = ufc)
> summary(hd.lm.2)
> xyplot(haut.m ~ diam.cm | especes, panel = function(x, y, ...) {
+     panel.xyplot(x, y)
+     panel.abline(lm(y ~ x), col = "blue")
+     panel.abline(hd.lm.1, col = "darkgreen")
+     if (sum(!is.na(y)) > 2) {
+         panel.loess(x, y, span = 1, col = "red")
+     }
+ }, subset = especes != "", data = ufc)
```

Peut-être devrions-nous renseigner ces hauteurs manquantes. Nous pourrions utiliser un modèle pour le faire ; utilisons un simple modèle d'attribution à effet mixtes [17].

```
> require(nlme)
> hd.lme <- lme(I(log(haut.m)) ~ I(log(diam.cm)) * especes,
+               random = ~ I(log(diam.cm)) | plot,
+               na.action = na.exclude,
+               data = ufc)
> predicted.log.hauts <- predict(hd.lme, na.action = na.exclude,
+                                newdata = ufc)
```

Les étapes suivantes sont, en premier, de copier les hauteurs prédites, et en second, de copier au dessus des hauteurs prédites, les hauteurs observées là où elles existent.

```
> ufc$haut.m.p[!is.na(ufc$diam.cm)] <- exp(predicted.log.hauts)
> ufc$haut.m.p[!is.na(ufc$haut.m)] <- ufc$haut.m[!is.na(ufc$haut.m)]
```

## 3.5 Fonctions

Les fonctions existent pour convertir diamètres et hauteurs en volumes, par espèces [29], pour cette région géographique. Elles semblent complexes, mais sont faciles à utiliser et performantes. Ici nous laissons les conversions en dehors de la fonction, mais elles peuvent être à l'intérieur.

```
> vol.fvs.ni.bdft <- function(spp, dbh.in, ht.ft){
+   bf.params <-
+     data.frame(
+                 especes = c("WP", "WL", "DF", "GF", "WH", "WC", "LP", "ES",
+                   "SF", "PP", "HW"),
+                 b0.small = c(26.729, 29.790, 25.332, 34.127, 37.314, 10.472,
+                   8.059, 11.851, 11.403, 50.340, 37.314),
+                 b1.small = c(0.01189, 0.00997, 0.01003, 0.01293, 0.01203,
+                   0.00878, 0.01208, 0.01149, 0.01011, 0.01201, 0.01203),
+                 b0.large = c(32.516, 85.150, 9.522, 10.603, 50.680, 4.064,
+                   14.111, 1.620, 124.425, 298.784, 50.680),
+                 b1.large = c(0.01181, 0.00841, 0.01011, 0.01218, 0.01306,
+                   0.00799, 0.01103, 0.01158, 0.00694, 0.01595, 0.01306))
+   dimensions <- data.frame(dbh.in = dbh.in,
+                            ht.ft = ht.ft,
```

```
+                             especes = as.character(spp),
+                             this.order = 1:length(spp))
+   dimensions <- merge(y=dimensions, x=bf.params, all.y=TRUE, all.x=FALSE)
+   dimensions <- dimensions[order(dimensions$this.order, decreasing=FALSE),]
+   b0 <- with(dimensions, ifelse(dbh.in <= 20.5, b0.small, b0.large))
+   b1 <- with(dimensions, ifelse(dbh.in <= 20.5, b1.small, b1.large))
+   volumes <- b0 + b1 * dimensions$dbh.in^2 * dimensions$ht.ft
+   return(volumes)
+ }
```

Après sauvegarde de cette fonction on peut l'utiliser dans notre code comme toute autre fonction.

```
> cm.en.pouces <- 1/2.54
> m.en.pieds <- 3.281
> pc.pd.en.m3 <- 0.002359737
> ufc$vol.m3 <- with(ufc, vol.fvs.ni.bdft(especes,
+                                         diam.cm*cm.en.pouces,
+                                         haut.m.p*m.en.pieds) * pc.pd.en.m3)
> ufc$sect.m2 <- ufc$diam.cm^2*pi/40000
> ufc$facteur.arbre <- ufc.baf / ufc$sect.m2
> ufc$vol.m3.ha <- ufc$vol.m3 * ufc$facteur.arbre
```

Nous avons maintenant une estimation du volume en m$^3$ par ha représenté par chaque arbre.

```
> str(ufc)
```

## 3.6 Traitement au niveau parcelle

D'abord, nous construisons un data-frame constitué des situations (connues) des parcelles.

```
> ufc.parc <- as.data.frame(cbind(c(1:144), rep(c(12:1),12),
+                                                rep(c(1:12), rep(12,12))))
> names(ufc.parc) = c("parc","nord.n","est.n")
> ufc.parc$nord = (ufc.parc$nord.n - 0.5) * 134.11
> ufc.parc$est = (ufc.parc$est.n - 0.5) * 167.64
```

Ensuite nous pouvons calculer les valeurs au niveau des parcelles pour chacune des caractéristiques mesurées, comme le volume marchand par hectare, par exemple.

```
> ufc.parc$vol.m3.ha <- tapply(ufc$vol.m3.ha, ufc$parc, sum)
> ufc.parc$vol.m3.ha[is.na(ufc.parc$vol.m3.ha)] <- 0
> head(merge(ufc, ufc.parc[, c("parc","nord","est")]))
```

À quoi cela ressemble t'il, dans l'espace ?

```
> contourplot(vol.m3.ha ~ est * nord,
+             main = expression(paste("Volume (", m^3, "/ha)", sep="")),
+             xlab = "Est (m)", ylab="Nord (m)",
+             region = TRUE,
+             col.regions = terrain.colors(11)[11:1],
+             data=ufc.parc)
```

Plutôt horrible. Mais sauvegardons l'objet de données parcelles pour plus tard.

```
> save("ufc.parc", file="../images/ufc_parc.RData")
```

Nous pouvons aussi calculer le total, les intervalles de confiance et tout ce qui suit.

```
> mean(ufc.parc$vol.m3.ha) * ufc.surface
> (mean(ufc.parc$vol.m3.ha) + qt(c(0.025,0.975),
+                               df = length(ufc.parc$vol.m3.ha) - 1) *
+   sd(ufc.parc$vol.m3.ha) / sqrt(length(ufc.parc$vol.m3.ha))) * ufc.surface
```

Enfin, estimons le volume total et l'intervalle de confiance à 95% pour chaque espèce.

D'abord, obtenons les estimations par espèces au niveau des parcelles.

```
> vol.par.especes <- tapply(ufc$vol.m3.ha, list(ufc$parc, ufc$especes), sum)
> vol.par.especes[is.na(vol.par.especes)] <- 0
> head(vol.par.especes)
```

On peut examiner les distributions comme suit :

```
> boxplot(as.data.frame(vol.par.especes))
```

Nous pouvons remarquer les miracles de la vectorisation !

```
> (totaux <- apply(vol.par.especes, 2, mean) * ufc.surface)
> (barres.ic <- apply(vol.par.especes, 2, sd) / sqrt(144) * 1.96 * ufc.surface)
```

Un tracé de boîtes à moustaches améliorées montre la variabilité dans les données et les estimations du volume moyen à l'hectare par parcelle.

```
> boxplot(as.data.frame(vol.par.especes))
> lines(1:11, (totaux - barres.ic) / ufc.surface, col="blue")
> lines(1:11, (totaux + barres.ic) / ufc.surface, col="blue")
> lines(1:11, totaux / ufc.surface, col="red")
```

# Chapitre 4

# Infrastructure

**R** fournit plusieurs outils qui nous aident à rester organisés et performants. Nous avons étudié jusqu'ici un cas d'outils simples et importants. Ce qu'il y a de compliqué et d'aussi important, comme les analyses de données, les environnements, la portée, etc., attendra.

## 4.1 Obtenir de l'aide

Il y a quatre sources principales d'assistance : les fichiers d'aide, les manuels, les archives **R**-help et **R**-help lui-même.

### 4.1.1 En local

En parcourant ce document, il est vraissemblable que vous trouviez des commandes utilisées en exemple qui n'ont pas été expliquées de façon adéquate voire complètement ignorée. Dans un tel cas vous devriez consulter ce qu'il en est dit dans `help()`.

On peut accéder aux fichiers d'aide en utilisant la commande `help()` ou son raccourci préfixe `?`. Nous pouvons obtenir de l'aide sur des commandes, ainsi :

```
> ?mean       # Fonctionne ! mean() est une commande R
> help(mean) # Fonctionne ! mean() est une commande R
```

Cependant, nous ne pouvons obtenir de l'information sur des concepts qui n'ont pas été enregistrés dans le système d'aide.

```
> help(regression) # Échoue ! regression() n'est pas une commande R
```

Cette demande d'aide échoue car « regression » n'est pas une commande **R**. Si l'on veut savoir à quelle commande se réfère `regression`, on utilise :

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

On va nous dire d'essayer `help(lm)` qui fait partie d'un grand nombre de commandes qui font référence à la régression. On nous dit aussi que la fonction `lm()` est dans le paquetage `stats`. On ne nous dit pas que le paquetage `stats` est déjà chargé, mais c'est le cas. Notez que la commande d'aide omet les parenthèses qui suivent normalement un appel de fonction. Notez aussi que le niveau d'aide fourni sur les différentes commandes est variable. Si vous pensez à un moyen plus clair d'exprimer quelque chose, faites-en part à la communauté en lui soumettant votre amélioration.

J'ai trouvé que le meilleur moyen d'avoir une idée de ce que les commandes attendent est d'essayer les exemples qui apparaissent en fin de l'information d'aide. Pour la plupart des fichiers d'aide il suffit de copier ces commandes exemples, de les coller sur la console **R** et de constater ce qui se passe. Les commandes peuvent être modifiées pour suivre vos besoins. En plus, ces exemples sont souvent une analyse de données miniature et pointent sur d'autres fonctions utiles à essayer.

Nous pouvons aussi accéder aux fichiers qui sont installés avec **R** en utilisant un fureteur web. Là encore, dans **R**, exécutez la commande suivante :

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

Un fureteur est ouvert (ou une fenêtre dedans, s'il y en a déjà un ouvert) dans lequel les résultats d'aide seront envoyés et avec lesquels vous pourrez pointer, cliquer et chercher des mots clefs. Vous pouvez fixer votre fureteur par défaut, selon votre système d'exploitation. Ce sera fait, par exemple, avec :

```
> options(browser="firefox")
```

Ce fureteur fournit aussi des hyper-liens d'accès aux manuels **R** qui donnent une grosse quantité d'informations utiles. Les manuels sont en constant développement aussi consultez-les périodiquement.

### 4.1.2 En déporté

Il y a une communauté bienveillante de programmeurs et d'utilisateurs qui seraient ravis de répondre à des questions soigneusement posées et, en fait, l'ont déjà organisé. Les questions et les réponses peuvent être aisément trouvées à l'intérieur de **R** en utilisant les commandes suivantes :

```
> RSiteSearch("Bates lmer")                # Douglas Bates à propos de lmer
> RSiteSearch("{logistic regression}")     # correspond à la phrase exacte
```

Si vous ne trouvez pas de réponse après une recherche soigneuse, il faudra faire appel à la communauté avec la liste **R**-Help. Il existe une **R**-Help-étiquette, pour vous aider à poser des questions susceptibles de recevoir des réponses utiles, expliquée à l'adresse :

http://www.r-project.rg/posting-guide.html

Des détails pour s'inscrire à la liste **R**-Help sont à :

http://stats.ethz.ch/mailman/listinfo/r-help

Je recommande l'option digest ; jusqu'à 100 mails arrivent par jour.

## 4.2 Écrire des scripts

**R** est en fait aussi simple que d'écrire des scripts. On peut écrire des scripts dans tout éditeur de texte – Winedt, MS Wordpad, Open Office, vi, emacs ou Xemacs, SciTE et d'autres. Certains éditeurs, comme emacs et Xemacs, vous rendront la vie plus facile en fournissant la tabulation intelligente, la coloration syntaxique et le complément des commandes, tous aussi intéressants que l'exécution de **R** en sous-processus signifiant que la sortie de **R** est aussi un tampon éditable.

Indépendamment de l'éditeur, on sauvegarde les scripts dans un répertoire connu et soit on utilise le copier-coller sur la console **R**, soit on les lit avecles commandes suivantes :

```
> source(file="C://chemin/vers/nomdefichier/fichier.R", echo=T)
> source(file="../repertoire/fichier.R", echo=T)
> source("fichier.R", echo=T) # Si fichier.R" est dans le répertoire de travail
```

Notez l'utilisation des slashes pour séparer les répertoires (même sous Windows). De même, les noms de répertoires sont sensibles à la casse et autorisés à contenir des blancs.

Écrire des scripts lisibles bien commentés est vraiment une bonne habitude à prendre rapidement ; cela rendra la vie future bien plus facile. De grand projets seront grandement simplifiés par une écriture de script rigoureuse.

Un élément clef d'une bonne écriture de script est la présence de commentaires. En **R** le symbole de commentaire est le symbole #. Tout ce qui suit un # est ignoré. Certains éditeurs tabuleront les commentaires sur la base du nombre de # utilisés.

Des instructions peuvent être délimitées par des nouvelles lignes ou des point-virgules. **R** est attentif à la syntaxe, aussi un retour chariot avant la fermeture d'une parenthèse ou d'un crochet, provoquera l'attente polie du reste de l'instruction.

L'écriture de scripts est un outil de collaboration très puissant. C'est très agréable de pouvoir envoyer votre code et un fichier de données brut à un collègue et savoir qu'il lui suffit d'exécuter `source()` sur le code pour effectuer votre analyse sur sa machine.

## 4.3 Espaces de travail

**R** utilise le concept d'*espaces de travail* pour nous aider à garder nos objets organisés. Tous les objets que nous fabriquons, directement ou indirectement, sont stockés dans un espace de travail. On peut sauvegarder, charger, partager ou archiver ces espaces de travail. On peut aussi lister le contenu de son espace de travail en utilisant `ls()` et en effacer les items avec `rm()`. Par exemple :

```
> test <- 1:10                       # Crée un objet
> ls()                               # Liste tous les objets de l'espace de travail
> save.image(file="../images/W-Shop.RData")
        # Sauvegarde tous les objets en fichier binaire dans le répertoire d'images
> save(test, file="../images/test.RData")
        # Sauvegarde test en fichier binaire
> rm(test)                           # Supprime l'objet "test".
> ls()                               # Liste les objets que nous avons
> rm(list=ls())                      # Les supprime tous
> load(file="../images/W-Shop.RData") # Les recharge
> ls()                               # Liste les objets rechargés
```

Les fichiers binaires dans lesquels la commande `save.image()` écrit les objets de notre espace de travail sont lisibles par **R** sous tous les systèmes d'exploitation. Ils sont considérablement compressés quand on les compare avec, disons, les fichiers à séparateurs virgule.

Les espaces de travail peuvent être très utiles. Les analyses de données peuvent souvent être séparées en morceaux, certains étant très coûteux en temps machine et d'autres très rapidement exécutés. C'est inutile de répéter les morveaux coûteux non nécessaires. Les résultats de ces morceaux peuvent être sauvegardés en images binaires et rechargés à volonté. Par exemple, on peut avoir un jeu de données de 20 Mega-octets qui requiert de considérables opérations de nettoyage et de manipulation avant que l'analyse voulue ne puisse débuter. Il serait fastidieux de répéter ce procédé à chaque fois. Il est bien plus facile de ne le faire qu'une fois et de sauvegarder l'espace de travail dans une image.

Les espaces de travail masquent aussi les pièges néfastes dus aux invariants. Il est possible à un utilisateur, disons un étudiant, de construire un script contenant de nombreuses étapes d'analyse et d'essayer de le partager avec un autre utilisateur, disons le chargé de travaux dirigés, seulement pour que ce dernier constate que le succès du script requiert un objet stocké et oublié de longue date dans l'espace de travail du premier utilisateur ! C'est un écueil commun à toute collaboration. Mes étudiants et moi aimerions, autant que possible, préfacer tous nos codes par :

```
> rm(list=ls())
```

Cela assure un espace de travail propre. `rm()` est la fonction qui efface les objets et `ls()` liste les objets de votre espace de travail.

## 4.4 Historique

La présence de l'historique dans **R** permet de garder trace des commandes que l'on a saisi, même les bêtises. Cela peut être très utile, spécialement car on peut y accéder à l'aide des flèches haute et basse. Ainsi, si l'on se trompe dans une commande, le remède est aussi simple que de taper sur la flèche haute, d'éditer la commande et de taper sur `enter`.

C'est bien pour de petits problèmes ; c'est rapide et efficace. Cela devient rapidement contre-performant, surtout quand vous voulez répéter de nombreuses commandes sans les changer beaucoup. Vous allez devenir nerveux. Écrivez dzs scripts (cf. Section 4.2).

Il est parfois utile après une session de sauvegarder l'historique dans un fichier pour le convertir ensuite en script.

```
> savehistory(file="History.txt")         # Historique sauvegardé en fichier texte
> loadhistory(file="History.txt")         # Fichier texte chargé en Historique
```

On peut alors utiliser cet historique comme base d'écriture de script (cf. Section 4.2). Des commentaires peuvent être insérés comme nous l'avons vu plus haut et ensuite le lire avec la commande `source()`. Je ne le fais presque jamais. Je ne développe presque jamais non plus de scripts à l'intérieur de **R**. Je préfère utiliser un éditeur externe optimisé pour le développement de scripts – des exemples sont emacs, Xemacs, Tinny et Bluefish.

## 4.5 Extensibilité

Une chose à noter pour **R** est la rapidité des chargements. C'est parce que beaucoup de ses fonctionnalités sont gardées en arrière plan, ignorées jussqu'à ce qu'elles soient appelées explicitement. Il y a trois niveaux de fonctions dans **R** :

1. celles qui sont chargées par défaut au démérrage (`base`),
2. celles qui sont chargées sur le disque dur au moment de l'installation mais non chargées en mémoire vive explicitement au démarrage de **R**, et
3. celles qui sont disponibles sur Internet, qui doivent être installées avant d'être chargées.

Une fonction dans le paquetage `base`, comme `mean()` est toujours disponible. Une fonction d'un des paquetages chargés, comme l'outil `lme()` du modèle d'ajustement linéaire à effets mixtes, peut n'être utilisée qu'après chargement du paquetage. On charge le paquetage avec la commande `require()`. `help.search()` vous dira quel paquetage charger pour utiliser une commande ou pour en savoir plus à son sujet.

```
> require(nlme)
```

Si l'on a besoin de savoir quels sortes de paquetages sont installés, **R** le dira, mais, de façon caractéristique, de la plus générale façon. Il faudra travailler sur la sortie pour obtenir l'information voulue.

```
> installed.packages()   # La sortie occupe tout l'écran
> sessionInfo()          # Présentation de l'état courant de votre session.


R version 2.7.1 (2008-06-23)
i386-pc-mingw32

locale:
LC_COLLATE=French_France.1252;LC_CTYPE=French_France.1252;LC_MONETARY=French_France.1252;
LC_NUMERIC=C;LC_TIME=French_France.1252

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
[1] tools_2.7.1

> ip <- installed.packages()   # Sauvegarde la sortie comme un objet
> class(ip)                    # C'est une matrice

[1] "matrix"

> ip <- as.data.frame(ip)      # C'est maintenant un data-frame
> names(ip)                    # Donne les noms des variables (colonnes)

 [1] "Package"  "LibPath"  "Version"  "Priority" "Bundle"   "Contains"
 [7] "Depends"  "Imports"  "Suggests" "OS_type"  "Built"

> length(ip$Package)

 [1] 142

> head(ip$Package)

 abind   acepack    ada    ade4   akima    amap
 abind   acepack    ada    ade4   akima    amap
 142 Levels: abind acepack ada ade4 akima amap ape arules base boot car ... zoo
```

Les noms de vos paquetages différeront des miens. Pour trouver les paquetages disponibles, utilisez :

```
> available.packages()   # La sortie occupe tout l'écran
```

Une fonction d'un paquetage non chargé est inaccessible tant que le paquetage n'est ni téléchargé ni installé. C'est très facile à faire si l'on est connecté à l'Internet et que l'on dispose des droits administrateur (root). C'est-à-dire, par exemple, que si, en cherchant sur le site web **R**, on détermine que l'on a besoin d'un merveilleux paquetage nommé « equivalence », il suffit de l'installer en utilisant le menu `Packages` (sous Windows) ou avec la commande :

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

Après son installation on peut le charger au cours d'une session particulière avec la commande `require(equivalence)`, comme ci-dessus.

Si l'on ne dispose pas des droits suffisants pour installer des paquetages il faut alors donner plus d'informations à **R**. Il faut télécharger et installer le paquetage en un lieu où l'on dispose des droits d'écriture et ensuite utiliser la commande `library(equivalence)` pour l'attacher à la session courante.

Installons le paquetage `equivalence`, en utilisant un dossier nommé « library ».

La commande `download.packages()` ci-dessous extraira la version appropriée pour votre système d'exploitation et fera part de son numéro de version, que nous sauvegardons dans l'objet nommé `get.equivalence`.

Nous pouvons utiliser les sorties des étapes suivantes.

```
> (get.equivalence <- download.packages("equivalence",
+                                       destdir="../library",
+                                       repos="http://cran.univ-lyon1.fr/"))
> install.packages(get.equivalence[1,2], repos=NULL, lib="../library")
> library(equivalence, lib.loc="../library")
```

Ah bon ! `equivalence` nécessite `boot`, entre autres, Ok ! (Attention il ne s'agit pas de relancer le système d'exploitation, pour les habitués de Windows, mais du paquetage utile au boostrapping ).

```
> (get.boot <- download.packages("boot",
+                                destdir="../library",
+                                repos="http://cran.univ-lyon1.fr/"))
> install.packages(get.boot[1,2], repos=NULL, lib="../library")
> library(boot, lib.loc="../library")
> library(equivalence, lib.loc="../library")
```

Le paquetage `equivalence` est maintenant prêt à être utilisé. Notez encore, svp., l'utilisation des slashes non inversés comme séparateurs de répertoire qui rendent ces instructions indépendantes de la plate-forme.

# Chapitre 5

# Interface

## 5.1 Importer et exporter des données

Il n'y a pas de meilleur moyen pour devenir familier avec un nouveau programme que de passer du temps avec, en utilisant des données connues. Il est tellement plus facile d'apprendre à interpréter des sorties variées quand on sait à quoi s'attendre ! Importer et exporter des données en partant de **R** peut sembler un peu confus quand vous êtes habitués aux (abêtissants) assistants Microsoft, mais c'est tout simple à apprivoiser et il y a bien plus de contrôle et de souplesse. Des manuels entiers ont été écrits sur le sujet – il y a tant de formats différents – mais nous n'en garderons qu'un seul : les fichiers à séparateur virgule.

Les fichiers à séparateur virgule sont devenus la *lingua franca* de communication des données de petites tailles : les entrées-sorties de presque toutes les bases de données. Ce sont de simples fichiers plats, c'est-à-dire qu'ils peuvent être représentés comme des tableaux à deux dimensions ; des lignes et des colonnes, sans autre structure. Les colonnes sont séparés par des virgules et les lignes par des nouvelles lignes (avec ou sans retour chariot selon le système d'exploitation). Les noms de colonne et les noms de ligne peuvent être enregistrés ou pas en première ligne et première colonne respectivement. Il faut toujours le vérifier !

```
Plot, Tree, Species, DBH, Height, Damage, Comment
1, 1, PiRa, 35.0, 22, 0,
1, 2, PiRa, 12.0, 120, 0, "Ce n'est certes pas un arbre réel !"
1, 3, PiRa, 32.0, 20, 0,
```

, etc.

**R** fournit un nombre limité de fonctions pour interagir avec le système d'exploitation. Je les utilise rarement, mais quand je le fais, ils sont inappréciables. La commande pour obtenir une liste des objets d'un répertoire est `list.files()`.

Cela peu être utile chaque fois que vous écrivez un script qui demande à **R** de lire un certain nombre de fichiers dont vous ne connaissez pas le nom au moment où vous écrivez.

```
> list.files("../data")
```

```
[1] "ehc.csv"   "rugby.csv" "stage.csv" "ufc.csv"
```

ou encore combien il y en a.

```
> length(list.files("../data"))
```

```
[1] 4
```

### 5.1.1 Importer

**R** a besoin de peu pour importer des données. D'abord, il doit connaître où elles sont, ensuite où il faut les mettre et enfin si elles ont des caractéristiques spéciales. Je commence toujours par examiner les données dans un tableur – Excel ou OO-calc le font bien – parce que j'ai besoin de savoir plusieurs choses :

1. Les colonnes et les lignes sont-elles cohérentes ? les feuilles de calcul Excel peuvent être difficiles à importer car les utilisateurs profitent de leur souplesse pour inclure différentes choses non compréhensibles pour **R**.
2. Les colonnes et lignes ont elles des libellés ? Les libellés sont ils facilement traitables ? (c'est-à-dire évitent-ils les soulignés, les espaces ou les signes pour-cent, etc.).
3. Y a t'il des données manquantes ? Les données manquantes sont elles explicitement représentées dans les données ? Si oui, quels symboles sont-ils utilisés pour les représenter.

4. Y a t'il des symboles dans la base de données qui peuvent rendre l'interprétation difficile ?

Nous devons aussi connaître le lieu où les fichiers seront ajoutés. Nous pouvons alors dire à **R** comment charger les données. En supposant que les données sont pertinentes et séparées par des virgules, nous pouvons utiliser la commande `read.csv()`. Noter qu'il faut donner à **R** le nom de l'objet (dans notre cas `ufc`) pour y stocker les données. En utilisant le chemin absolu (si nous le connaissons), cela donne :

```
> ufc <- read.csv(file=("C://chemin/vers/nomFichier/ufc.csv")
```

On peut aussi utiliser une adresse relative :

```
> ufc <- read.csv(file=("../data/ufc.csv")
```

Noter encore l'utilisation des slashes pour séparer les noms de répertoire. Ces noms sont sensibles à la casse et autorisés à contenir des espaces. Si l'on utilise `read.csv()`, **R** suppose que la première ligne contient les noms de colonne ; sinon, lui dire avec l'option `header=F`. Voir `?read.csv` pour des détails sur les options.

D'autres commandes sont utiles quand les données ont une structure plus générale : `read.fwf()` lit des données de format à largeur fixe, `read.table()` s'accommodera d'une plus grande collection de tableaux et `scan()` lira arbitrairement tout fichier texte.

Quand les données sont importées, quelques autres outils utiles aident à vérifier l'état d'exhaustivité du jeu de données et certains de ses attributs. Utilisez-les régulièrement pour vous assurer que les données sont telles que vous les attendiez.

```
> dim(ufc)

[1] 637   5

> names(ufc)

[1] "plot"    "tree"    "species" "dbh"     "height"
```

Je préfère toujours travailler avec des noms de variables en minuscules. Une façon d'assurer que les noms sont toujours en minuscules est de le forcer.

```
> names(ufc) <- tolower(names(ufc))
> str(ufc)

'data.frame': 637 obs. of  5 variables:
 $ plot   : int  1 2 2 3 3 3 3 3 3 3 ...
 $ tree   : int  1 1 2 1 2 3 4 5 6 7 ...
 $ species: Factor w/ 13 levels "","DF","ES","F",..: 1 2 12 11 6 11 11 11 11 11 ...
 $ dbh    : int  NA 390 480 150 520 310 280 360 340 260 ...
 $ height : int  NA 205 330 NA 300 NA NA 207 NA NA ...

> head(ufc)

  plot tree species dbh height
1    1    1          NA     NA
2    2    1      DF 390    205
3    2    2      WL 480    330
4    3    1      WC 150     NA
5    3    2      GF 520    300
6    3    3      WC 310     NA

> ufc[1:5, ]

  plot tree species dbh height
1    1    1          NA     NA
2    2    1      DF 390    205
3    2    2      WL 480    330
4    3    1      WC 150     NA
5    3    2      GF 520    300
```

La dernière de ces commandes est un des plus utiles éléments de manipulation en **R** : l'extraction indicée. Ce sera traité en Section 6.3.

### 5.1.2 Exporter

Exporter des données depuis **R** est une chose moins commune qu'importer, mais heureusement c'est aussi immédiat. Ici encore nous utiliserons le format de fichier `csv` à moins qu'il n'y ait une bonne raison de ne pas le faire.

```
> write.csv(ufc, file="C://chemin/vers/nomFichier/fichier.csv")
> write.csv(ufc, file="chemin/vers/nomFichier/fichier.csv")
```

Exporter les graphiques est tout aussi simple. En anticipant un peu, nous utiliserons la commande `plot()` pour créer un graphique de variables bi-dimensionnel.

```
> plot(1:10,1:10)  # ... ou quelque chose de plus sophistiqué ...
```

Sous Windows, on peut cliquer-droit sur ce graphique pour le copier dans le presse-papiers, soit en bitmap, soit en meta-fichier Windows. Le dernier type de fichier est très utile car on peut le coller dans des documents – MS Word, par exemple – et les adapter en taille ou les éditer par la suite.

Sauvegarder les graphiques dans un format plus permanent (voire plus courant) est aussi possible. Pour sauvegarder un graphique en pdf, par exemple, nous écrirons :

```
> pdf("../graphics/nomFichier.pdf")  # Ouvre un terminal pdf
> plot(1:10,1:10)  # ... ou quelque chose de plus sophistiqué ...
> dev.off()        # Ferme le terminal pdf et sauvegarde le fichier
```

Les mêmes protocoles sont disponibles pour `postscript()`, `jpeg()`, etc. Vous en apprendrez plus avec :

```
> ?Devices
```

# Chapitre 6

# Données

Les stratégies pour la manipulation des données est le cœur de l'expérience **R**. L'orientation objet assure à cette étape utile et importante une gestion faite de petits morceaux de code élégants.

## 6.1 Données : objets et classes

Alors, qu'est-ce que cela veux dire que R est orienté objet ? Simplement, cela signifie que toutes les interactions avec **R** passent par les objets. Les structures de données sont des objets, comme le sont les fonctions et les scripts.

Cela semble obscur pour le moment, mais dès que vous deviendrez plus familier avec, vous vous rendrez compte que cela permet une grande souplesse, de l'intuition dans la communication avec **R**, et occasionnellement une royale fessée.

Pour la plus grande partie, l'orientation objet ne changera pas la façon de faire les choses, sauf parfois les rendre plus faciles que nous ne le pensions. Cependant une conséquence importante de l'orientation objet est que tous les objets sont membres d'une ou plusieurs classes.

Nous utiliserons cette possibilité plus tard (en Section 7.5, pour être précis).

Nous créons des objets et leur affectons des noms avec la flèche gauche « `<-` ». **R** devinera de quelle classe nous voulons qu'il soit, ce qui affecte ce que nous pouvons en faire. Nous pouvons changer la classe si l'on n'est pas d'accord avec **R**.

```
> a <- 1                    # Crée un objet "a" et
                            # lui affecte la valeur 1.
> a <- 1.5                  # Retire le 1 et met 1.5 à la place.
> class(a)                  # De quelle classe est-il ?
> class(a) <- "character"   # En fait une classe caractère
> class(a)                  # Quelle est sa classe ?
> a                         # Valeur de a
> a <- "Arnaud"             # Retire le 1.5 et met "Arnaud" à la place.
> b <- a                    # Crée un objet "b" et
                            # lui affecte  tout ce qui est dans l'objet a.
> a <- c(1,2,3)             # Retire le "Arnaud" et en fait un vecteur
                            # avec les valeurs 1, 2 et 3.
                            # Ne fait jamais de c un objet !
> b <- c(1:3)               # Retire le "Arnaud" et en fait un vecteur
                            # avec les valeurs 1, 2 et 3.
> b <- mean(a)              # Affecte l'objet a à l'objet b.
> ls()                      # Liste tous les objets créés par l'utilisateur
> rm(b)                     # Efface b
```

Quelques points sont à noter : nous ne devons pas déclarer les variables d'une classe particulière. **R** les force à celle qu'il faut. De même, nous n'avons pas à déclarer la taille des vecteurs. C'est agréable pour les utilisateurs.

## 6.2 Classes des données

Il existe deux sortes fondamentales de données : les nombres et les chaînes de caractère (tout ce qui n'est pas un nombre est une chaîne de caractère). Il y a plusieurs types de chaîne de caractère et chacun d'eux a des propriétés particulières. **R** distingue entre ces différents types d'objet par leur classe.

**R** connaît ces différentes classes et de quoi elles sont capables. On peut connaître la nature de tout objet en utilisant la commande `class()`. Autrement, on peut demander s'il est d'une classe spécifique avec la commande `is.nomClasse()`. On peut aussi souvent changer de classe, avec la commande `as.nomClasse()`. Ce processus peut se produire par défaut et dans ce cas est nommé *coercition.*

### 6.2.1 Numeric

Un nombre peut être entier, réel ou complexe. **R** peut généralement dire la différence entre les deux premiers types à l'aide du contexte. On le teste par `is.numeric()` et le force par `as.numeric()`. **R** gère aussi les nombres complexes, mais ils n'intéressent pas ce document. On peut faire toutes les choses usuelles avec les nombres :

```
> a <- 2          # Crée un objet "a" et lui affecte le nombre 2.
> class(a)        # De quelle classe est-il ?
> is.numeric(a)   # Est-ce un nombre ?
> b <- 4          # Crée un objet "b" et lui affecte le nombre 4.
> a + b           # Addition
> a - b           # Soustraction
> a * b           # Multiplication
> a / b           # Division
> a ^ b           # Exponentiation (puissance)
> (a + b) ^ 3     # Parenthèses
> a == b          # Test d'égalité (retourne une valeur logique)
> a < b           # Comparaison (retourne une valeur logique)
> max(a,b)        # Le plus grand entre a et b
> min(a,b)        # Le plus petit entre a et b
```

### 6.2.2 String

Une collection d'un ou plusieurs caractères alpha-numériques, délimitée par simple- ou double-quotes. On le teste par `is.character()` et le force par `as.character()`. **R** procure beaucoup de fonctions de manipulation de chaîne de caractère, y compris des possibilités de recherche.

```
> a <- "chaîne"       # Crée un objet "a" et lui affecte la valeur "chaîne".
> class(a)            # De quelle classe est-il ?
> is.numeric(a)       # Est-ce un nombre ?
> is.character(a)     # Est-ce une chaîne ?
> b <- "spaghetti"    # Crée un objet "b" et lui affecte la valeur "spaghetti".
> paste(a, b)         # Joint les chaînes
> paste(a, b, sep="") # Joint les chaînes sans espace entre
> d <- paste(a, b, sep="")
> substring(d, 1, 4)  # Extrait la chaîne
```

### 6.2.3 Factor

Les facteurs représentent les variables qualitatives.

En fait, les facteurs ne sont pas terriblement différents des chaînes, si ce n'est qu'ils ne peuvent prendre qu'un nombre limité de valeurs, dont **R** garde un enregistrement et sait en faire des choses très utiles. On contrôle si un objet est un facteur avec `is.factor()` et le change en facteur, si c'est possible, avec `factor()`.

Même si **R** rend compte des opérations sur les facteurs par les niveaux qu'on leur affecte, il les représente en interne comme un entier. Par conséquent, les facteurs peuvent créer de considérables ennuis, sauf s'ils sont étroitement surveillés. Cela veut dire que chaque fois que vous faites une opération comprenant un facteur vous devez vous assurer qu'il fait ce que vous voulez, en examinant la sortie et les étapes intermédiaires. Par exemple, les niveaux d'un facteur sont classés par ordre alphabétique par défaut. Cela signifie que si vos niveaux commencent par des nombres, comme numéros d'identificateurs, vous pouvez penser que le numéro 10 arrive avant le numéro 2. Il n'y a pas de problème si vous le savez déjà !

```
> a <- c("A","B","A","B") # crée un vecteur a
> class(a)                # De quelle classe est-il ?
> is.character(a)         # est-ce une chaîne ?
> is.factor(a)            # est-ce un facteur ?
> a <- factor(a)          # le forcer en facteur
> levels(a)               # quels sont les niveaux ?
> table(a)                # quels en sont le décompte ?
> a <- factor(c("A","B","A","B"), levels=c("B","A"))
                          # crée un facteur à différents niveaux
```

Parfois, il sera nécessaire de travailler avec un sous-ensemble des données, peut-être pour des raisons de commodité, ou peut-être parce que certains niveaux de facteur représentent des données non pertinentes. Si nous partageons les données, il sera souvent nécessaire de redéfinir les facteurs dans le jeu de données pour faire savoir à **R** qu'il devrait abandonner les niveaux qui manquent. Nous en saurons beaucoup plus sur les facteurs lorsque nous commencerons à manipuler des vecteurs (cf. Section 6.3.1).

### 6.2.4 Logical

C'est un genre de facteur, qui a seulement deux niveaux : Vrai et Faux. Les variables logiques sont mises à part des facteurs car leurs niveaux sont interchangeables avec les chiffres 1 et 0 (respectivement) par coercition. La sortie de plusieurs fonctions utiles sont des variables logiques (également appelées booléen). Nous pouvons construire des instructions logiques en utilisant les opérateurs et (`&`), ou (`|`) et non (`!`).

```
> a <- 2        # Crée une variable a et lui affecte le nombre 2
> b <- 4        # Crée une variable b et lui affecte le nombre 4
> d <- a < b    # Comparaison
> class(d)      # Qu'est-ce ?
> e <- TRUE     # Crée une variable e et lui affecte la valeur TRUE
> d + e         # Qu'est-ce que ça fait ?
> d \& e        # d ET e est Vrai
> d | e         # d OU e est aussi Vrai
> d \& !e       # d ET (NON e) n'est pas Vrai
```

On peut faire appel aux extractions indicées de vecteur pour tous les objets dont une condition est vraie avec `which()`.

### 6.2.5 Données manquantes

La dernière et pire sorte de données est nommée donnée manquante (NA). Il ne s'agit pas d'une classe unique, à proprement parler. Elles peuvent être mélangées avec tous les autres types de données. Il est plus facile d'y penser comme des places des données qui auraient dû être présentes, mais qui, pour une raison quelconque, ne le sont pas. Malheureusement, leur traitement n'est pas uniforme dans toutes les fonctions. Parfois vous devez dire à la fonction de les ignorer et parfois non. Et, il y a différentes manières de dire à la fonction comment les ignorer suivant la personne qui l'a écrite et son but.

Il y a quelques fonctions pour la manipulation des valeurs manquantes. Nous pouvons les détecter par `is.na()`.

```
> a <- NA                # Affecte NA à la variable a
> is.na(a)               # Est-ce une valeur manquante ?
> class(a)               # De quelle classe est-elle ?
> class(d)               # Qu'est-ce ?
> a <- c(11,NA,13)       # Essayons maintenant avec un vecteur
> mean(a)                # Cela semble correct
> var(a)                 # Eh-eh !
> var(a, na.rm=TRUE)     # Pouah !
> is.na(a)               # Y a t'il une valeur manquante ?
```

On peut identifier les lignes complètes (c'est-à-dire les lignes qui n'ont pas de donnée manquante) dans un objet à deux dimensions avec la commande `complete.cases()`.

## 6.3 Structures des données

Ayant examiné les types les plus importants de données, examinons les mécanismes dont nous disposons pour les enregistrer et les manipuler en bloc. Il y en a plus que ceux que nous avons vu jusqu'ici – certains (matrice, liste) peuvent être très utile.

### 6.3.1 Vector

Un vecteur est une collection uni-dimensionnelle d'objets atomiques (c'est-à-dire des objets qui ne peuvent pas être plus séparés). Les vecteurs peuvent contenir des nombres, des chaînes de caractère, des facteurs ou des valeurs logiques. Tous les objets que nous avons créés jusqu'ici étaient des vecteurs bien que certains étaient de taille 1. La clef de la construction de vecteur est que tous les objets doivent être de la même classe. La clé de la manipulation de vecteur est en d'utiliser ses indices. Les indices sont accessibles en utilisant les crochets [ ].

```
> a <- c(11,12,13) # a est un vecteur
> a[1]             # le premier objet de a
> a[2]             # le second objet de a
> a[-2]            # a, mais sans le second objet
> a[c(2,3,1)]      # a, mais dans un ordre différent
> a + 1            # Ajoute 1 à tous les éléments de a
> length(a)        # le nombre d'unités dans le vecteur a
> order(c(a,b))    # retourne les indices de a et b en ordre croissant
> c(a,b)[order(c(a,b))] # retourne a et b en ordre croissant
> a <- c(11,NA,13) # a est toujours un vecteur
> a[!is.na(a)]     # quels sont les éléments de a qui ne sont pas manquants ?
> which(!is.na(a)) # quels sont les places des éléments non-manquants de a ?
```

Notez que nous avons pu imbriquer un vecteur dans le mécanisme d'indice d'un autre vecteur ! Cela donne une idée de la puissance de l'orienté objet. Cette introduit également une clef de ce qui donne à **R** un traitement si performant : la vectorisation.

**Vectorisation**
Le concept de vectorisation est simple : rendre un processus plus performant. On se souvient, Section 6.2.5, qu'en appliquant la fonction `is.na()` au vecteur `a`, il en est résulté que la fonction a été appliquée à *chaque élément* du vecteur et qu'il en est sorti un vecteur sans intervention de l'utilisateur. C'est la vectorisation. Imaginons que nous ayons un ensemble de 1 000 000 de diamètres d'arbre et qu'il faille les convertir en surface de base. En C ou Fortran nous écririons une boucle. La version **R** de la boucle aurait cet aspect (enveloppée dans un compteur horaire).

```
> diametres <- rgamma(n=1000000, shape=2, scale=20)
> surfaces.base <- rep(NA, length(diametres))
> system.time(
+ for (i in 1:length(diametres)) {
+ surfaces.base[i] <- diametres[i]^2 * pi / 40000
+ }
+ )
   user  system elapsed
   8.93    0.00    9.03
```

Cela prend presque 10 secondes sur mon portable. Cependant, si nous vectorisons l'opération, cela devient considérablement plus rapide.

```
> diametres <- rgamma(n = 1e+06, shape = 2, scale = 20)
> surfaces.base <- rep(NA, length(diametres))
> system.time(surfaces.base <- diametres^2 * pi/40000)
   user  system elapsed
   0.10    0.01    0.14
```

C'est quarante à cinquante fois plus rapide. Bien sûr, si nous avions programmé cette fonction en C ou Fortran, le résultat aurait été beaucoup plus rapide encore. Le mantra de programmation R pourrait être le suivante : ne compiler que si vous en avez besoin, boucler seulement s'il le faut, et vectoriser tout le temps. La vectorisation ne fonctionne que pour certaines fonctions, par exemple cela ne fonctionnera pas pour `mean()`, parce que cela n'aurait pas de sens, nous voulons la moyenne des nombres dans le vecteur, pas la moyenne de chaque unité. Mais, quand la vectorisation fonctionne elle rend la vie plus facile, un code plus propre et un temps de traitement plus rapide.

Noter que `pi` est une simple valeur, pas de taille $10^6$ et **R** suppose que nous aimerions qu'il soit répété $10^6$ fois. C'est ce qu'on appelle le recyclage.

**Recyclage**
On a peut-être remarqué jusqu'ici que **R** procure une commodité pour la manipulation des vecteurs. Lorsque nous avons tapé :

```
> a <- c(11, 12, 13)
> a + 1
```

```
[1] 12 13 14
```

**R** a supposé que nous voulions ajouter 1 à chaque élément de `a`. C'est ce qu'on appelle le recyclage, qui est généralement très utiles mais parfois très dangereux. **R** recycle 1 jusqu'à ce que ce soit de la même longueur que `a`. La fonction est interprétée comme :

```
> a <- c(11, 12, 13)
> a + c(1, 1, 1)
```

```
[1] 12 13 14
```

Pour aller plus loin, si nous voulons convertir tous les nombres d'un vecteur de pouce en centimètre, il suffit de faire :

```
> (a <- a * 2.54)
```

```
[1] 27.94 30.48 33.02
```

Le recyclage peut être dangereux parce que parfois nous voulons que les dimensions correspondent exactement et l'inadéquation conduira à de mauvais calculs. Si nous ne parvenons pas à aligner nos résultats exactement – par exemple en raison de quelques valeurs manquantes – **R** ira de l'avant et calculera le résultat de toute façon. Le seul moyen d'être sûr est de regarder les avertissements, examiner les sorties soigneusement et garder à l'esprit que la plupart du temps, le recyclage est vraiment utile.

```
> (a + c(1, -1, 1))
```

```
[1] 28.94 29.48 34.02
```

### 6.3.2 Data-frame

Un data-frame est une puissante structure de gestion vectorielle à deux dimensions. Il est optimisé pour représenter des jeux de données multidimensionnelles : chaque colonne correspond à une variable et chaque ligne correspond à une observation. Un data-frame peut contenir des vecteurs de toute classes de base d'objets à un moment donné. Ainsi, une colonne peut être en caractères avec une autre en facteur et une troisième numérique.

Il est toujours possible de se référer aux objets dans le data-frame par l'intermédiaire de leurs indices, en utilisant les crochets. Il y a maintenant deux dimensions : ligne et colonne. Si l'une est laissée vide, alors toute la dimension est prise en charge. C'est-à-dire `test[1 :10,]`, va récupérer les dix premières lignes de toutes les colonnes du data-frame `test`, en utilisant le développement mentionné ci-dessus que les deux-points représentent les entiers intermédiaires. `test[,c(2,5,4)]` va extraire toutes les lignes pour les deuxième, cinquième et quatrième colonnes. Ces sélections d'indices peuvent être imbriquées ou appliquées de façon séquentielle. Les nombres négatifs dans les sélections d'indices désignent les lignes qui seront omises.

Chaque colonne, ou variable, dans un data-frame a un nom unique. Nous pouvons extraire cette variable par le biais du nom de data-frame et du nom de colonne séparés par un signe dollar : *data-frame$variable*.

Une collection de colonnes peuvent être sélectionnées par leur nom en fournissant un vecteur des noms de colonne dans l'indice de spécification. Ceci est utile dans les data-frames où l'emplacement des colonnes est incertain ou dynamique, mais les noms restent fixes.

Si un fichier à délimiteur virgule est lu en utilisant les commandes de la Section 5.1, **R** dira que c'est censé être devenu un data-frame. La commande pour le vérifier est `is.data.frame()`, et celle pour le forcer est `as.data.frame()`. Il existe de nombreuses fonctions pour examiner les data-frames ; nous en présentons quelques-unes ci-dessous.

```
> ufc <- read.csv("../data/ufc.csv")
> is.data.frame(ufc)
```

```
[1] TRUE

> dim(ufc)

[1] 637   5

> names(ufc)

[1] "plot"    "tree"    "species" "dbh"     "height"

> ufc$height[1:5]

[1]  NA 205 330  NA 300

> ufc$species[1:5]

[1]    DF WL WC GF
Levels:  DF ES F FG GF HW LP PP SF WC WL WP

> ufc[1:5, c(3, 5)]

  species height
1             NA
2      DF    205
3      WL    330
4      WC     NA
5      GF    300

> ufc[1:5, c("species", "height")]

  species height
1             NA
2      DF    205
3      WL    330
4      WC     NA
5      GF    300

> table(ufc$species)

     DF  ES   F  FG  GF  HW  LP  PP  SF  WC  WL  WP
 10  77   3   1   2 185   5   7   4  14 251  34  44
```

La dernière commande donne à penser qu'il y a dix arbres avec des espèces vides. Ici, il est indispensable de connaître le protocole d'enregistrement. Les dix arbres sans espèces sont des lignes vides pour représenter les parcelles non plantées. Nous allons les supprimer, pour le moment. Notez que nous allons aussi redéfinir les facteurs d'espèces, pour qu'il retire le niveau est actuellement vide.

```
> ufc <- ufc[ufc$species != "",]
> ufc$species <- factor(ufc$species)
```

Nous pouvons également créer de nouvelles variables dans un data-frame, en les nommant et en leur affectant une valeur. Donc,

```
> ufc$dbh.cm <- ufc$dbh/10
> ufc$height.m <- ufc$height/10
```

Enfin, si nous voulons construire un data-frame de variables déjà existantes, ce qui est tout à fait courant, nous utilisons la commande `data.frame()`, c'est-à-dire :

```
> temp <- data.frame(my.species = ufc$species, my.dbh = ufc$dbh.cm)
> temp[1:5, ]

  my.species my.dbh
1         DF     39
2         WL     48
3         WC     15
4         GF     52
5         WC     31
```

Les data-frames sont les plus utiles des structures de données en ce qui nous concerne. Nous pouvons utiliser des vecteurs logiques qui renvoient à un aspect du data-frame : extraire des informations du reste du data-frame, ou même d'un autre data-frame. Enfin, il est possible d'extraire beaucoup d'informations en utilisant les outils que nous avons déjà vu.

```
> ufc$height.m[ufc$species == "LP"]  # Hauteurs des pins de lodgepole

[1] 24.5   NA   NA   NA 16.0 25.0   NA

> mean(ufc$height.m[ufc$species == "LP"], na.rm = TRUE)

[1] 21.83333
```

**Sapply**

`sapply()` permet de déployer une fonction sur chaque colonne d'un data-frame. C'est particulièrement utile si l'on veut trouver des sommes ou des moyennes de colonne, par exemple, mais elle a d'autres applications utiles.

```
> sapply(ufc[, 4:7], mean, na.rm = TRUE)

      dbh    height    dbh.cm  height.m
356.56619 240.66496  35.65662  24.06650
```

Par la suite, par exemple, nous utiliserons `sapply()` pour connaître la classe des colonnes de notre data-frame.

```
> sapply(ufc, class)

     plot      tree   species       dbh    height    dbh.cm  height.m
"integer" "integer"  "factor" "integer" "integer" "numeric" "numeric"
```

### 6.3.3 Matrice (Tableau)

Une matrice est simplement un vecteur avec des attributs supplémentaires, appelés dimensions. **R** prévoit des algorithmes spécifiques pour nous permettre de traiter ce vecteur comme si il était vraiment à deux dimensions. De nombreuses opérations matricielles sont disponibles :

```
> (mat.1 <- matrix(c(1, 0, 1, 1), nrow = 2))
     [,1] [,2]
[1,]    1    1
[2,]    0    1

> (mat.2 <- matrix(c(1, 1, 0, 1), nrow = 2))
     [,1] [,2]
[1,]    1    0
[2,]    1    1

> solve(mat.1)      # Inversion de la matrice
     [,1] [,2]
[1,]    1   -1
[2,]    0    1
```

```
> mat.1 %*% mat.2  # Multiplication de matrice

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

> mat.1 + mat.2    # Addition de matrice

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

> t(mat.1)         # Transposition de matrice

     [,1] [,2]
[1,]    1    0
[2,]    1    1

> det(mat.1)       # Déterminant de matrice

[1] 1
```

Il existe également diverses autres fonctions de matrices, par exemple, `qr()` produit la décomposition QR, `eigen()` produit les valeurs propres et vecteurs propres de matrices, et `svd()` produit la décomposition en valeurs singulières. Les tableaux sont des matrices à une, deux ou trois dimensions.

#### Apply

La fonction `apply()` nous permet de déployer une fonction sur une ligne ou une colonne dans une matrice ou un tableau. C'est particulièrement utile si nous voulons trouver des sommes ou des moyennes de ligne ou colonne, par exemple.

```
> apply(ufc[, 4:7], 2, mean, na.rm = TRUE)

      dbh    height    dbh.cm  height.m
356.56619 240.66496  35.65662  24.06650
```

### 6.3.4 Liste

Une liste est un conteneur pour les autres objets. Les listes sont précieuses, par exemple, pour la collecte et le stockage des sorties de fonctions complexes. Les listes deviennent de précieux dispositifs qui nous mettent plus à l'aise avec **R** et permettent de penser à différents moyens de résoudre nos problèmes. Nous accédons aux éléments d'une liste en utilisant le double crochet, comme indiqué ci-dessous.

```
> (ma.liste <- list("un", TRUE, 3))

[[1]]
[1] "un"

[[2]]
[1] TRUE

[[3]]
[1] 3

> ma.liste[[2]]

[1] TRUE
```

Si l'on utilise un simple crochet, on extrait alors l'élément déjà imbriqué dans l'infrastructure de liste.

```
> ma.liste[2]

[[1]]
[1] TRUE
```

On peut aussi nommer les éléments de la liste, soit au cours de sa construction, soit après.

```
> (ma.liste <- list(premier = "un", second = TRUE, troisieme = 3))

$premier
[1] "un"

$second
[1] TRUE

$troisieme
[1] 3

> names(ma.liste)

[1] "premier"   "second"    "troisieme"

> my.list$second

[1] TRUE

> names(ma.liste) <- c("Premier élément", "Second élément", "Troisième élément")
> ma.liste

$`Premier élément`
[1] "un"

$`Second élément`
[1] TRUE

$`Troisième élément`
[1] 3

> ma.liste$"Second élément"

[1] TRUE
```

Notez le déploiement des pointeurs arrières pour afficher les éléments de la liste, même si le nom contient des espaces.

Le résultat de beaucoup de fonctions est un objet liste. Par exemple, lorsque nous ajustons aux moindres carrés une régression, l'objet régression lui-même est une liste, et peut être manipulé avec les opérations de liste.

Ci-dessus, nous avons vu l'utilisation de `tapply()`, pratique qui nous a permis d'appliquer une fonction arbitraire à tous les éléments groupés d'un vecteur, groupe par groupe. Nous pouvons utiliser `lapply()` pour appliquer une fonction arbitraire à tous les éléments d'une liste.

## 6.4 Fusion des données

Il est souvent nécessaire de fusionner les jeux de données qui contenant des données à différents niveaux hiérarchiques, par exemple, en gestion forestière nous pourrions peut-être avoir besoin de paramètres au niveau des espèces que nous souhaiterions fusionner avec des mesures au niveau arbre pour faire des prédictions à partir d'un modèle connu. Nous rencontrons exactement ce problème en écrivant la fonction qui calcule le volume de garde-pied d'un arbre selon son espèce, son diamètre et sa hauteur. Voyons comment cela fonctionne à plus petite échelle. Tout d'abord, nous déclarons le data-frame qui contient les paramètres de niveau espèces.

Rappelons qu'en ajoutant les parenthèses, nous demandons à **R** d'afficher le résultat et de l'enregistrer.

```
> (params <- data.frame(species = c("WP", "WL"),
+                       b0 = c(32.516, 85.15),
+                       b1 = c(0.01181, 0.00841)))

  species     b0      b1
1      WP 32.516 0.01181
2      WL 85.150 0.00841
```

Ensuite, nous allons extraire de ufc les trois premiers arbres des espèces qui ont des hauteurs non manquantes. Nous vous ne garderons que les colonnes appropriées. Notez que nous pouvons "empiler" les appels d'indice et qu'ils sont évalués successivement de gauche à droite :

```
> (trees <- ufc[ufc$species %in% params$species & !is.na(ufc$height.m),
+     ][1:3, ])

   plot tree species dbh height dbh.cm height.m
3     2    2      WL 480    330   48.0       33
20    4    9      WP 299    240   29.9       24
26    5    6      WP 155    140   15.5       14
```

Maintenant nous fusionnons les paramètres avec les espèces :

```
> (trees <- merge(trees, params))

  species plot tree dbh height dbh.cm height.m     b0      b1
1      WL    2    2 480    330   48.0       33 85.150 0.00841
2      WP    4    9 299    240   29.9       24 32.516 0.01181
3      WP    5    6 155    140   15.5       14 32.516 0.01181
```

Il existe de nombreuses options pour la fusion, par exemple, sur la façon de traiter avec des caractéristiques pas-tout à fait imbriquées, ou des noms de colonnes qui ne correspondent pas.

Nous pouvons maintenant calculer le volume en utilisant une approche vectorielle, ce qui est considérablement plus efficace que d'utiliser une boucle. Notez également l'utilisation avec de `with()` pour attacher *temporairement* le data-frame à notre chemin de recherche. Cet usage simplifie le code.

```
> (trees$volume <- with(trees, b0 + b1 * (dbh.cm/2.54)^2 * (height.m *
+     3.281)) * 0.002359737)

[1] 0.9682839 0.3808219 0.1243991
```

## 6.5 Réadaptation des données

Les jeux de données chronologiques ont souvent un mauvais aspect pour l'analyse. Quand des données chronologiques sont conservées dans une feuille de calcul, il convient de consacrer une ligne pour chaque objet mesuré et d'inclure une colonne pour chaque instant, de sorte que l'ajout de nouvelles mesures exige simplement l'ajout d'une colonne. Du point de vue de l'analyse des données, toutefois, il est plus facile de considérer chaque observation de l'objet comme une ligne. **R** rend cette considération pratique. Encore une fois, nous allons voir comment cela fonctionne à petite échelle.

```
> (trees <- data.frame(tree = c(1, 2), species = c("WH", "WL"),
+                      dbh.1 = c(45, 52), dbh.2 = c(50, 55),
+                      ht.1 = c(30, 35), ht.2 = c(32, 36)))

  tree species dbh.1 dbh.2 ht.1 ht.2
1    1      WH    45    50   30   32
2    2      WL    52    55   35   36
```

```
> (trees.long <- reshape(trees, direction = "long",
+                         varying = list(c("dbh.1", "dbh.2"),
+                                        c("ht.1", "ht.2")),
+                         v.names = c("dbh", "height"),
+                         timevar = "time",
                          idvar = "tree"
+                         ))
```

```
    tree species time dbh height
1.1    1      WH    1  45     30
2.1    2      WL    1  52     35
1.2    1      WH    2  50     32
2.2    2      WL    2  55     36
```

Les arguments sont définis comme suit :

**direction** dit à **R** d'aller en « `long` » ou en large (« `wide` »),

**varying** est une liste de vecteurs des noms de colonne qui doivent être empilés,

**v.names** est un vecteur des nouveaux noms des colonnes empilées,

**timevar** est le nom de la nouvelle colonne qui différencie les mesures successives de chaque objet et

**idvar** est le nom d'une colonne existante qui différencie les objets.

## 6.6 Tri des données

**R** procure la possibilité d'ordonner les éléments objets. Le tri est essentiel pour résoudre les opérations de fusion, par exemple, comme nous l'avons fait dans le Chapitre 3. Voici les cinq premiers arbres triés par taille.

```
> ufc[order(ufc$height.m, decreasing = TRUE), ][1:5, ]
```

```
    plot tree species  dbh height dbh.cm height.m
413   78    3      WP 1030    480  103.0     48.0
532  110    4      GF  812    470   81.2     47.0
457   88    3      WL  708    425   70.8     42.5
297   55    2      DF  998    420   99.8     42.0
378   68    1      GF  780    420   78.0     42.0
```

La fonction `order()` permet de trier plusieurs vecteurs comme, par exemple, pour trier par parcelle, puis par espèces et par hauteur, nous utiliserons :

```
> ufc[order(ufc$plot, ufc$species, ufc$height.m), ][1:5, ]
```

```
   plot tree species dbh height dbh.cm height.m
2     2    1      DF 390    205     39     20.5
3     2    2      WL 480    330     48     33.0
5     3    2      GF 520    300     52     30.0
8     3    5      WC 360    207     36     20.7
11    3    8      WC 380    225     38     22.5
```

Les fonctions apparentées comprennent `sort()` et `rank()`, mais elle ne peuvent être utilisées qu'avec un indice (en général créé par `order()`).

# Chapitre 7

# Programmation

Une des plus grandes forces de **R** est son extensibilité. En effet, bon nombre d'outils dont nous nous sommes servi ont été créés par d'autres personnes. **R** peut être étendu par l'écriture de fonctions en utilisant le langage **R** (cf. Section 7.1) ou d'autres langages tels que C et Fortran (cf. Section 7.6). Être confortable en écriture de fonction est très utile, car cela ajoute une dimension entièrement nouvelle à la performance et la facilité de programmation.

## 7.1 Fonctions

L'écriture de fonctions en **R** mérite son propre atelier, dont nous allons simplement parcourir la surface. Nous déclarons une fonction de la manière suivante :

```
> ma.fonction <- function(arguments) {
+ }
```

Nous mettons ensuite les commandes **R** entre les accolades {}. À moins que nous n'utilisions une commande `return()` explicite, la recherche retournera le résultat de la dernière déclaration dans les accolades. Si la fonction est seulement d'une ligne, alors les accolades peuvent être omises. Voici une fonction qui pourrait être utile à la NASA :

```
> cm.en.pouces <- function(data) {
+     data/2.54
+ }
```

Nous pouvons appeler cette fonction comme toute autre :

```
> cm.en.pouces(25)

[1] 9.84252

> cm.en.pouces(c(25, 35, 45))

[1]  9.84252 13.77953 17.71654
```

Elle est même vectorisée (cf. Section 6.3.1) ! Évidemment, c'est trivial. Mais en ce qui concerne **R**, maintenant cette fonction est tout comme une autre.

Les fonctions peuvent être utilisées de plusieurs différentes façons. Un exemple utile est de rassembler les appels des fonctions que nous utilisons souvent. Cette fonction ouvre un fichier de données sur un terrain et éventuellement résout des problèmes communs et lui affecte des mesures de hauteur :

```
> donnees.terrain <- function(fichier = "../data/ufc.csv", clair = TRUE,
+     affecte = FALSE) {
+     terrain <- read.csv(file)
+     if (clair) {
+         terrain$dbh.cm <- terrain$dbh/10
+         terrain$height.m <- terrain$height/10
+         terrain$height.m[terrain$height.m < 0.1] <- NA
```

```
+         terrain$species[terrain$species %in% c("F", "FG")] <- "GF"
+         terrain$species <- factor(terrain$species)
+         if (affecte) {
+             require(nlme)
+             hd.lme <- lme(I(log(height.m)) ~ I(log(dbh.cm)) *
+                 species, random = ~I(log(dbh.cm)) | plot, data = terrain)
+             terrain$haut.pred.m <- exp(predict(hd.lme, level = 1))
+         }
+     }
+     return(terrain)
+ }
```

Des versions plus sophitiquées comprendraient contrôle d'erreur et compte rendu, plus propres, etc. En fait une fonction comme celle-ci croîtra de façon organique.

## 7.2 Contrôle de flux

**R** donne accès à un certain nombre de structures de contrôle pour la programmation - `if()`, `for()`, et `while()`, par exemple. Prenez le temps de les regarder dans le système d'aide. En général il vaut mieux éviter ces fonctions dans la mesure du possible, car beaucoup de problèmes pouvant être résolus par ces fonctions peuvent l'être plus rapidement et avec un code plus strict avec la vectorisation (cf. Section 6.3.1).

### 7.2.1 Étude de cas

À titre d'exemple, donc, imaginez que nous nous intéressions de connaître la proximité à la distribution normale des estimations de paramètres pour une régression linéaire et quel est l'aspect d'une distribution jointe empirique. Nous avons déjà trouvé un moyen de le faire en Section 10.9 avec les fonctions de bootstrapping, mais faisons-le maintenant avec une boucle explicite. D'abord, obtenons les données :

```
> rugby <- read.csv("../data/rugby.csv")
> rugby$rules <- "New"
> rugby$rules[rugby$game < 5.5] <- "Old"
> rugby$game.no <- as.numeric(rugby$game)
> rugby$rules <- factor(rugby$rules)
> rugby$game <- factor(rugby$game)
> str(rugby)
```

```
'data.frame': 979 obs. of  4 variables:
 $ game   : Factor w/ 10 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ time   : num  39.2 2.7 9.2 14.6 1.9 17.8 15.5 53.8 17.5 27.5 ...
 $ rules  : Factor w/ 2 levels "New","Old": 2 2 2 2 2 2 2 2 2 2 ...
 $ game.no: num  1 1 1 1 1 1 1 1 1 1 ...
```

**Boucle**
Ensuite, nous construisons la boucle :

```
> reps <- 200                   # Choisir un nombre de répliques
> differences <- rep(NA, reps) # Préparer un réceptacle
> (n <- nrow(rugby))            # Trouver la taille d'échantillon
```

```
[1] 979
```

```
> system.time({                                   # Juste pour chronométrer
+     for (i in 1:reps) {                         # Exécuter une pour chaque valeur de i
+         new.index <- sample(1:n, replace = TRUE) # Échantillonner l'indice
+         differences[i] <-                       # Ajuster au modèle et en extraire les estimations
+           coef(lm(time[new.index] ~ rules[new.index], data = rugby))[2]
+     }
+ })
```

```
   user  system elapsed 
   1.90    0.02    2.31 
```

Enfin nous examinons la moyenne, l'écart-type et la distribution des estimations paramétriques (cf. Figure 7.1).

```
> mean(differences)

[1] 3.486977

> sd(differences)

[1] 0.9179585

> qqnorm(differences, main = "Quantiles-Quantiles des Estimations",
+     xlab = "quantiles théoriques", ylab = "quantiles d'échantillon")
> qqline(differences)
```

Quantiles−Quantiles des Estimations

quantiles d'échantillon

quantiles théoriques

Figure 7.1 – Tracé des quantiles d'estimations d'un paramètre aléatoire

### 7.2.2 Boucles et autres choses

Le `for()` est plus générique que l'on ne pourrait soupçonner, compte tenu de sa simplicité. La gamme de valeurs à prendre par l'indice n'a pas à être séquentielle, ni dans n'importe quel ordre, ou même des numéros. Cela peut être très utile lorsque vous le souhaitez, par exemple, lire un grand nombre de fichiers pour de futures opérations.

```
> (noms.fichiers <- list.files("../data"))

[1] "ehc.csv"   "rugby.csv" "stage.csv" "ufc.csv"  

> (nb.car.noms <- nchar(noms.fichiers))

[1] 7 9 9 7

> (noms.fichiers <- noms.fichiers[substr(noms.fichiers,
+                                    nb.car.noms - 3,
+                                    nb.car.noms) == ".csv"])
```

```
[1] "ehc.csv"   "rugby.csv" "stage.csv" "ufc.csv"

> for (my.file in noms.fichiers) {
+     in.file <- read.csv(paste("../data", my.file, sep = "/"))
+     cat(paste("File", my.file, "has", nrow(in.file), "rows.\n",
+         sep = " "))
+ }

File ehc.csv has 488 rows.
File rugby.csv has 979 rows.
File stage.csv has 542 rows.
File ufc.csv has 637 rows.
```

## 7.3 Portée

Lorsque nous écrivons des fonctions dans certaines langages, nous devons affecter ou déclarer toutes les variables qui seront utilisées dans la fonction. C'est ce qu'on appelle une règle de délimitation de portée, et cela signifie que le langage sait toujours où trouver ses variables. **R** a une approche plus souple : il va d'abord chercher dans la fonction, et s'il n'y trouve pas ce qu'il cherche, il cherchera à un niveau au-dessus dans l'environnement d'appel, et ainsi de suite. Cela a deux conséquences importantes. Tout d'abord, nous n'avons pas à transmettre toutes les variables que nous pourrions souhaiter utiliser. Cela rend le code plus clair et plus facile à lire dans de nombreux cas. Toutefois, d'autre part, on peut trouver en amont des variables qui affectent les résultats en aval si on a une erreur dans le code qu'on ne connaîtrait jamais sinon. Donc, chaque fois que je développe du code,je prends l'habitude de faire une sauvegarde du script, en supprimant tous les objets, et en l'exécutant à nouveau, peut-être plusieurs fois. La portée est importante – elle peut vous retirer beaucoup de tracas, mais elle peut coûter beaucoup de temps si vous n'y prenez garde.

```
> x <- pi
> radius.to.area <- function(radius) {
+     x * radius^2
+ }
> radius.to.area(4)

[1] 50.26548
```

## 7.4 Déverminage

Un conseil qui concerne le déverminage en général est également utile pour **R**. C'est-à-dire, profilez votre code en utilisant de simples entrées dont vous êtes certain de savoir ce que la réponse devrait fournir, soyez généreux en commentaires et veillez à examiner les étapes intermédiaires.

De plus, utilisez `browser()` à l'intérieur de vos fonctions. `browser()` arrêtera le traitement, et vous permettra d'inspecter tous les objets, et, dans certains cas, votre fonction. Voir `?browser` pour plus d'informations. Vous pouvez utiliser :

```
> eval(expression(fonction()),
+ envir = sys.frame(n),
+ enclos = NULL)
```

pour exécuter `fonction()` en environnement relatif `n`, qui peut être, par exemple, l'environnement d'où votre fonction a été appelée. Par exemple, pour lister les objets de niveau supérieur, vous pourriez faire ce qui suit :

```
> rm(list = ls())
> ls()

character(0)

> x <- 2
> ls.up <- function() {
+     eval(expression(ls()), envir = sys.frame(-1), enclos = NULL)
+ }
> ls.up()
```

```
[1] "ls.up" "x"
```

Cet exemple n'est pas particulièrement utile en soi, mais montre comment construire des commandes qui opèrent dans des environnements différents, et ces commandes sont très utiles pendant le déverminage dans un fureteur.

## 7.5 Classes S3 et objets

Il n'est pas obligatoire d'appliquer les principes de la programmation orientée objet lors de l'écriture de code en **R**, mais cela peut avoir des avantages. Ici, je vais vous montrer l'utilisation des classes de style S3. En bref, nous allons créer une classe, qui est un type d'objet, et nous écrirons des fonctions qui s'appliquent spécifiquement à cette classe. Les fonctions remplaceront les fonctions soi-disant générique, comme `print()`, `plot()` et `mean()`. Notez l'absence la classe `inventaire`.

```
> methods(class = "inventaire")

no methods were found
```

Inventons-en une. Disons que la classe inventaire est une `list`, avec certains attributs supplémentaires. Pour commencer nous écrivons une fonction qui crée la classe à partir d'un fichier et d'information supplémentaire.

```
> inventaire <- function(fichier = "../data/ufc.csv",
+                        clair = TRUE,
+                        imputer = FALSE,
+                        plan = c("VRP", "FAP"),
+                        poids = NULL,
+                        surface = NULL
+                        ) {
+     donnees <- read.csv(fichier)
+     if (clair) {
+         donnees$diam.cm <- donnees$dbh/10
+         donnees$haut.m <- donnees$height/10
+         donnees$haut.m[donnees$haut.m < 0.1] <- NA
+         donnees$especes <- donnees$species
+         donnees$especes[donnees$especes %in% c("F", "FG")] <- "GF"
+         donnees$especes <- factor(donnees$especes)
+         if (imputer) {
+             require(nlme)
+             hd.lme <- lme(I(log(haut.m)) ~ I(log(diam.cm)) *
+                 especes, random = ~I(log(diam.cm)) | plot, na.action = na.exclude,
+                 data = donnees)
+             log.haut.predites <-
+                predict(hd.lme, na.action = na.omit, newdata = donnees)
+             donnees$haut.m.p[!is.na(donnees$diam.cm)] <- exp(log.haut.predites)
+             donnees$haut.m.p[!is.na(donnees$haut.m)] <-
+               donnees$haut.m[!is.na(donnees$haut.m)]
+         } else {
+             donnees$haut.m.p <- donnees$haut.m
+         }
+     }
+     resultats <- list(data = donnees,
+                       plan = plan,
+                       poids = poids,
+                       surface = surface)
+     class(resultats) <- "inventaire"
+     return(resultats)
+ }
```

La classe présumera qu'il n'existe que deux types d'inventaire ; parcelles de rayon variable (VRP), auquel cas l'argument poids sera interprété comme le Coefficient de Biotope par Surface (BAF), et les parcelles de surface fixe (FAP), auquel cas l'argument poids sera interprétée comme la taille en hectares de la parcelle. Notez les hypothèses que la fonction fait sur le format des données et les données incluses.

Maintenant, nous écrivons une fonction d'affichage qui sera utilisée pour tous les objets de la classe `inventaire`. Nous supposons que l'objectif de l'affichage d'inventaire est une description des données. L'affichage peut ensuite être invoqué d'une de deux façons :

```
> print.inventaire <- function(x) str(x)
```

Nous pouvons maintenant l'essayer :

```
> ufc <- inventaire(fichier = "../data/ufc.csv",
+                   clair = TRUE,
+                   imputer = TRUE,
+                   plan = "VRP",
+                   poids = 7,
+                   surface = 300)
> ufc
```

```
List of 4
 $ data   :'data.frame': 637 obs. of  9 variables:
  ..$ plot    : int [1:637] 1 2 2 3 3 3 3 3 3 3 ...
  ..$ tree    : int [1:637] 1 1 2 1 2 3 4 5 6 7 ...
  ..$ species : Factor w/ 13 levels "","DF","ES","F",..: 1 2 12 11 6 11 11 11 11 11 ...
  ..$ dbh     : int [1:637] NA 390 480 150 520 310 280 360 340 260 ...
  ..$ height  : int [1:637] NA 205 330 NA 300 NA NA 207 NA NA ...
  ..$ diam.cm : num [1:637] NA 39 48 15 52 31 28 36 34 26 ...
  ..$ haut.m  : num [1:637] NA 20.5 33 NA 30 NA NA 20.7 NA NA ...
  ..$ especes : Factor w/ 11 levels "","DF","ES","GF",..: 1 2 10 9 4 9 9 9 9 9 ...
  ..$ haut.m.p: num [1:637]   NA 20.5 33.0 13.8 30.0 ...
 $ plan   : chr "VRP"
 $ poids  : num 7
 $ surface: num 300
 - attr(*, "class")= chr "inventaire"
```

Nous pouvons maintenant commencer à réfléchir à la façon dont nous interagissons avec des données d'inventaire. Par exemple, nous pourrions décider qu'un tracé d'inventaire devrait toujours avoir une série de quatre panneaux, ainsi :

```
> plot.inventaire <- function(inventaire, ...) {
+     plot(inventaire$donnees$diam.cm, inventaire$donnees$haut.m,
+         main = "Hauteur en fonction du Diamètre", xlab = "Diam. (cm)",
+         ylab = "Haut. (m)", ...)
+     require(MASS)
+     truehist(inventory$data$dbh.cm, h = 5, x0 = 0, xlab = "Diam. (cm)",
+         main = "Distribution du diamètre, non pondéré", ...)
+     boxplot(diam.cm ~ especes, data = inventaire$donnees, ylab = "Diam. (cm)",
+         varwidth = TRUE)
+     boxplot(haut.m ~ especes, data = inventaire$donnees, ylab = "Haut. (m)",
+         varwidth = TRUE)
+ }
```

Notez tout d'abord que nous avons dû extraire les données provenant de deux couches d'infrastructure, en utilisant une séquence de signes $ et, d'autre part, qu'il est très facile de le faire. Le rôle des trois points dans la liste des arguments est d'alerter la fonction qu'elle peut recevoir d'autres arguments, et que si elle le fait, elle doit s'attendre à passer ces arguments aux fonctions `plot()` et `boxplot()` inchangées.

Nous avons ensuite utilisé cette fonction comme suit, pour créer la Figure 7.2 :

```
> par(mfrow = c(2, 2), mar = c(4, 4, 3, 1), las = 1)
> plot(ufc, col = "seagreen3")
```

Figure 7.2 – Récapitulatif graphique d'un objet *inventaire*

En bref, alors, pour écrire une fonction, disons `truc()`, qui sera *automatiquement* appelée pour remplacer une fonction générique `chose()` lorsqu'elle est appliquée à un objet de classe`machin`, nous devons simplement faire appel à la fonction `truc.machin`.

Des fonctions non-génériques peuvent également être écrites, mais ne nécessitent pas le nom d'objet en queue. Par exemple, nous aimerions calculer le volume de tous les arbres, mais avoir des options entre nos nombreuses fonctions de volume.

```
> calcul.volume.arbres <- function(inventaire, fonction.vol) {
+     volume.arbre.m3 <- do.call(fonction.vol, list(inventaire$donnees$especes,
+                                                      inventaire$donnees$diam.cm,
+                                                      inventaire$donnees$haut.m.p))
+     return(volume.arbre.m3)
+ }
```

Notez encore une fois, que nous faisons des hypothèses importantes sur ce quelle information sera disponible pour la fonction, et comment elle interagit avec elle. Par exemple, nous supposons qu'il y aura une variable appelée `haut.m.p` disponible pour chaque arbre. Nous utiliserons la fonction de volume FVS (northerm Idaho) de Wykoff et al. (1982)[29], mais avec la conversion d'unité dans la fonction plutôt qu'à l'extérieur.

```
> vol.fvs.ni.m3 <- function(esp, diam.cm, haut.m) {
+     diam.in <- diam.cm/2.54
+     haut.ft <- haut.m * 3.281
+     biofac.params <- data.frame(
+                                   especes = c("WP", "WL", "DF", "GF", "WH", "WC", "LP", "ES",
+                         "SF", "PP", "HW"),
+                       b0.petit = c(26.729, 29.79, 25.332, 34.127, 37.314, 10.472,
+                         8.059, 11.851, 11.403, 50.34, 37.314),
+                       b1.petit = c(0.01189, 0.00997, 0.01003, 0.01293, 0.01203,
+                         0.00878, 0.01208, 0.01149, 0.01011, 0.01201, 0.01203),
+                       b0.grand = c(32.516, 85.15, 9.522, 10.603, 50.68, 4.064,
+                         14.111, 1.62, 124.425, 298.784, 50.68),
+                       b1.grand = c(0.01181, 0.00841, 0.01011, 0.01218, 0.01306,
+                         0.00799, 0.01103, 0.01158, 0.00694, 0.01595, 0.01306)
+                       )
+     dimensions <- data.frame(diam.in = diam.in,
+                              haut.ft = haut.ft,
+                              especes = as.character(esp),
+                              cet.ordre = 1:length(esp))
```

```
+     dimensions <- merge(y = dimensions, x = biofac.params, all.y=TRUE, all.x=FALSE)
+     dimensions <- dimensions[order(dimensions$cet.ordre, decreasing=FALSE),]
+     b0 <- with(dimensions, ifelse(diam.in <= 20.5, b0.petit, b0.grand))
+     b1 <- with(dimensions, ifelse(diam.in <= 20.5, b1.petit, b1.grand))
+     volumes.biofac <- b0 + b1 * dimensions$diam.in^2 * dimensions$haut.ft
+     volumes.m3 <- volumes.biofac * 0.002359737
+     return(volumes.m3)
+ }
```

Maintenant on peut calculer et sauvegarder les volumes des arbres dans l'objet inventaire ufc comme suit :

```
> ufc$donnees$vol.m3 <- calcul.volume.arbres(ufc, vol.fvs.ni.m3)
```

Noter qu'ajouter une colonne à ,l'attribut de données de l'objet n'a pas changé sa classe :

```
> class(ufc)

[1] "inventory"
```

Finalement, nous écrivons une fonction moyenne produisant la moyenne par hectare d'un simple attribut nommé, plutôt que la moyenne de l'attribut.

```
> mean.inventory <- function(inventory, attribute, ...)
+     data <- inventory$data[, names(inventory$data) == attribute]
+     if (inventory$design == "VRP")
+         tree.factor <- inventory$weight * 40000/(inventory$data$dbh.cm^2 *
+             pi)
+
+     else
+         tree.factor <- 1/inventory$weight
+
+     total <- sum(data * tree.factor, na.rm = TRUE)
+     mean <- total/length(unique(inventory$data$plot))
+     return(mean)
+
> mean(ufc, "vol.m3")

[1] 149.5207
```

Nous sommes en train de construire une petite collection de fonctions spécifiques de classe pour notre nouvelle classe.

```
> methods(class = "inventory")

[1] mean.inventory  plot.inventory  print.inventory
```

Il y a un écueil final. Que se passe t'il si plusieurs auteurs écrivent un jeu de fonctions qui requièrent la classe `inventory` et que les définitions de classe sont incompatibles ? C'est un des problèmes que les classes S4 évitent. La couverture des classes S4 va au delà de ce document, mais voir le dernier Chambers [4] pour de plus amples informations.

## 7.6 Autres langages

Juste une anecdote ici, avec quelques instructions sur la façon de compiler et exécuter vos propres programmes sous Unix. Je travaillais avec Robert Froese, mon premier thésard, sur le développement de tests d'équivalence pour la validation de modèle. Nous utilisions les données de la FIA (Forest Inventory and Analysis) pour valider l'engin de calcul de diamètre du Simulateur de Végétation Forestière (FVS) par espèce. Il y avait 41 000 arbres pour 10 espèces. Le mieux représenté est le Douglas-fir, avec 12 601 observations, le moins bien, le pin blanc, avec 289 et la moyenne de 3 725.

```
> (n <- tapply(species, species, length))
```

```
ABGR ABLA LAOC OTHE PICO PIEN PIMO PIPO PSME THPL TSHE
3393 4532 3794 1629 6013 3251 289 2531 12601 2037 909
```

En résumé :

Nous appliquions le test de rang signé. Nous aurions aimé utiliser le test de Student pour des données appariées qui est simple, immédiat et sans boucle détournée. Mais malheureusement les erreurs de prédiction étaient loin de la normale ; elles étaient très asymétriques. Aussi avons nous décidé d'utiliser un test non paramétrique. Le problème, ou dois-je dire le défi voire même l'**opportunité**, était que le calcul de métrique demande une double boucle sur les jeux de données.

$$\hat{U}_+ = \frac{1}{C_n^2} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} I_+(D_i + D_j) \tag{7.1}$$

où $I_+(x)$ est un indicateur qui prend la valeur 1 si $x > 0$ et 0 sinon. Cela représente

```
> sum(choose(n,2))

 [1] 133017037
```

soit plus d'un million d'opérations. Mais ce n'est pas tout ! Pour estimer la variance de la métrique, il faut implémenter une *triple* boucle sur les jeux de données. Et faire chacune à chacune de ces combinaisons au moins sept choses.

$$\hat{\sigma}^2_{\hat{U}_+} = \frac{1}{C_n^2}\{2(n-2)[(\hat{q}^+_{1(2,3)} - \hat{U}_+^2) + (\hat{U}_+ - \hat{U}_+^2)]\} \tag{7.2}$$

où

$$\hat{q}^+_{1(2,3)} = \frac{1}{C_n^3} \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} \frac{1}{3}(I_{ij}Iik + I_{ij}Ijk + I_{ik}Ijk) \tag{7.3}$$

pour lequel on a $I_{ij} = I_+(D_i + D_j)$. C'est à dire maintenant

```
> sum(choose(n,3))*7

 [1] 2.879732e+12
```

soit environ 2.9 milliards d'operations ! Eh bien, je suppose que cela prendra du temps. Aussi ai-je écrit un script,

```
n <- length(diffs)
cat("This is the sign rank test for equivalence.\n")
zeta <- 0
for (i in 1:(n-1)) {
for (j in (i+1):n) {
zeta <- zeta + ((diffs[i] + diffs[j]) > 0)
}
}
Uplus <- zeta / choose(n, 2)
cat("UPlus ", Uplus, ". \n")
zeta <- 0
for (i in 1:(n-2)) {
for (j in (i+1):(n-1)) {
for (k in (j+1):n) {
IJpos <- ((diffs[i] + diffs[j]) > 0)
IKpos <- ((diffs[i] + diffs[k]) > 0)
JKpos <- ((diffs[j] + diffs[k]) > 0)
zeta <- zeta + IJpos*IKpos + IJpos*JKpos + IKpos*JKpos
}
}
cat("Triple loop. We have ", n-2-i, "iterations left from ", n,". \n")
}
q123 <- zeta / 3 / choose(n, 3)
```

mis en marche mon serveur déporté et attendu. Toute la nuit. Au réveil je n'avais obtenu que 10 % de la première espèce (une toute petite). Je ne voulais pas attendre aussi longtemps. C'était comme si le chargement dynamique d'un code compilé lui était destiné.

### 7.6.1 Écriture

Le code doit être modulaire et écrit pour que toute communication passe par les arguments. Aussi en C ce doit être un type `void` et en Fortran un sous-programme.

En voici un exemple :

```
{
  int i, j, k, m;
  int IJpos, IKpos, JKpos;
  zeta[0] = ((a[i] + a[j]) > 0);
  zeta[1] = 0;
  for (i = 0; i < *na-2; i++) {
    for (j = i+1; j < *na-1; j++) {
      zeta[0] += ((a[i] + a[j]) > 0);
      for (k = j+1; k < *na; k++) {
        IJpos = ((a[i] + a[j]) > 0);
        IKpos = ((a[i] + a[k]) > 0);
        JKpos = ((a[j] + a[k]) > 0);
        zeta[1] += (IJpos*IKpos + IJpos*JKpos + IKpos*JKpos);
      }
    }
  }
  for (m = 0; m < *na-1; m++)
    zeta[0] += ((a[m] + a[*na-1]) > 0);
}
```

À la base, tout ce que j'ai fait était de convertir la triple boucle en code C. Noter que j'ai passé la taille de l'objet plutôt que de faire confiance à C (faites-moi confiance) pour l'évaluer. Faites en **R** ce que l'on peut bien faire en **R**.

- Noter que C indice autrement que **R** : le premier élément d'un tableau est l'élément 0, *pas* l'élément 1 ! Ce peut être une fessée.
- Ne pas utiliser de pointeurs. On en parle ailleurs. Pour de simples problèmes on peut s'en passer.
- J'ai aussi montré comment imbriquer une double boucle dans une triple, pour obtenir quelque amélioration de performance.
- Finalement, noter que les arguments sont les entrées / sorties et que ce sous-programme se moque de leur taille.

### 7.6.2 Compilation

La compilation sous Linux ou BSD est facile. Se placer sur une console, créer le fichier ci-dessus (nommé par exemple `signRankSum.c`) et dans le même répertoire, taper :

```
$ R CMD SHLIB signRankSum.c
```

On compile le code et édite ainsi les liens pour créer un objet à lier à la session **R** (cf. ci-dessous). Si cela ne fonctionne pas, on obtiendra des messages d'erreur d'une aide raisonnable.

Sous Windows, les outils dont on a besoin pour développer du logiciel sont un peu plus disparates. Se référer à des ressources disponibles pour éclairer ce point. Brièvement, on a besoin de :

- Installer des outils de ligne de commande de type Unix pour compiler et lier, par exemple MinGW ou autres.
- Éditer la variable système PATH pour qu'il soit visible par le système d'exploitation.
- Écrire le code approprié dans des répertoires de nom pertinent.
- Dans le shell DOS, exécuter : `R CMD SHLIB signRankSum.c`

### 7.6.3 Attachement

Dans le corps de la fonction que j'écris pour appliquer le test, j'ai ajouté l'appel suivant :

```
dy.load("~/Rosetta/Programming/r/signRankSum/signRankSum.so")
signRankSum <- function(a)
  .C("signRankSum",
     as.double(a),
     as.integer(length(a)),
     zeta = double(2))$zeta
```

Cela dit à **R** où trouver la bibliothèque. Noter que j'ai inclus le chemin entier vers l'objet *.so et que dans la fonction, j'ai donné un nom au sous-programme C et aux arguments. Ainsi, quand je fais appel à `signRankSum(a)`, `a` pointe sur un tableau double précision, j'ai besoin de la taille de `a`, un entier, et de `zeta`, tableau double précision de taille 2 pour la sortie. Alors le `$zeta` de la fin lui dit de retourner ce tableau.

### 7.6.4 Appel

Plus tard dans la fonction on appelle le sous-programme. C'est facile :

```
zeta <- signRankSum(diffs)
Uplus <- zeta[1] / choose(n, 2)
cat("Uplus ", Uplus, ". \n")
q123 <- zeta[2] / 3 / choose(n, 3)
```

Noter que l'on est revenu dans **R**, à présent, donc que le premier élément est référencé comme 1.

### 7.6.5 Bénéfice !

La sortie suivante est le compte rendu d'une comparaison utilisant le code C en premier et le pur code **R** en second.

```
> test <- rnorm(1000)
> system.time(sgnrk(test, limit=1000))
This is the sign rank test for equivalence.
Starting. Actual: 1000 . Used: 1000 .
UPlus 0.484042 .
VarUplus is calculated: 0.0003321017
Levels are calculated: 0.2397501 and 0.76025 , ncp is 203.9437
Cutoff: 12.63603
Value computed: -0.8756727 .
Finished. Outcome: dissimilarity is rejected.
[1] 3.882812 0.000000 3.906250 0.000000 0.000000
> system.time(sgnrk.old(test, limit=1000))
This is the sign rank test for equivalence.
Starting. Actual: 1000 . Used: 1000 .
UPlus 0.484042 .
VarUplus is calculated: 0.0003321017
Levels are calculated: 0.2397501 and 0.76025 , ncp is 203.9437
Cutoff: 12.63603
Value computed: -0.8756727 .
Finished. Outcome: dissimilarity is rejected.
[1] 5092.87500 14.27344 5120.23438 0.00000 0.00000
```

C'est quatre secondes comparées à presque une heure et demi. Et le temps pour finir le test pour toutes les espèces et toutes les données était :

```
[1] 9605.687500 5.367188 9621.734375 0.000000 0.000000
```

Moins de *trois heures*, plutôt que *cent soixante neuf jours*.

Cet exemple qui était très extrême requérait une triple boucle sur un grand jeu de données. Néanmoins passer par C peut apporter des gains substantiels même dans des circonstances plus modestes.

# Chapitre 8

# Simples descriptions

Nous avons déjà vu un certain nombre d'explorations de données et des outils d'analyse, mais il est utile de les examiner de façon plus coordonnée et globale. Commençons par la lecture et le traitement des données conformément à notre stratégie antérieure.

```
> ufc <- read.csv("../data/ufc.csv")
> ufc$height.m <- ufc$height/10
> ufc$dbh.cm <- ufc$dbh/10
> ufc$height.m[ufc$height.m < 0.1] <- NA
> ufc$species[ufc$species %in% c("F", "FG")] <- "GF"
> ufc$species <- factor(ufc$species)
> ufc.baf <- 7
> cm.to.inches <- 1/2.54
> m.to.feet <- 3.281
> bd.ft.to.m3 <- 0.002359737
```

Maintenant nous allons laisser tomber les parcelles qui n'ont pas d'arbres bien que nous en gardions trace dans leur décompte.

```
> number.of.plots <- length(unique(ufc$plot))
> ufc <- ufc[!is.na(ufc$dbh.cm), ]
> ufc$species <- factor(ufc$species)
```

## 8.1 Une seule variable

Dans ce qui suit nous distinguons les explorations de données continues numériques, nommées quantitatives, des données catégorielles, nommées qualitatives. Les données ordinales peuvent être considérées comme des cas particulier des dernières (qualitatives) pour notre propos actuel. Je ne vois pas de raison de distinguer, dans les données quantitatives, les intervalles des rapports.

### 8.1.1 Quantitative

Obtenir des estimations standard du centre et de l'étendue des variables est très simple ; on l'obtient avec les commandes suivantes. Noter l'utilisation de `na.rm=TRUE` qui dit à **R** d'ignorer les données manquantes.

#### Mesures du centre

```
> mean(ufc$dbh.cm, na.rm = TRUE)

[1] 35.65662

> median(ufc$dbh.cm, na.rm = TRUE)

[1] 32.9
```

### Mesures de l'étendue

```
> sd(ufc$dbh.cm, na.rm = TRUE)

[1] 17.68945

> range(ufc$dbh.cm, na.rm = TRUE)

[1]  10 112

> IQR(ufc$dbh.cm, na.rm = TRUE)

[1] 23.45
```

### Mesures de l'asymétrie

Une fonction `skewness()` se trouve dans le paquetage `moments`. Sinon on peut comparer les données avec un jeu connu comme symétrique ; la distribution normale (cf. Figure 8.1)

Ces données sont, bien évidemment, positivement asymétriques (c'est-à-dire avec une longue queue à droite).

```
> qqnorm(ufc$dbh.cm, xlab = "Diamètre (cm)", ylab = "Quantiles d'échantillon")
> qqline(ufc$dbh.cm, col = "darkgrey")
```

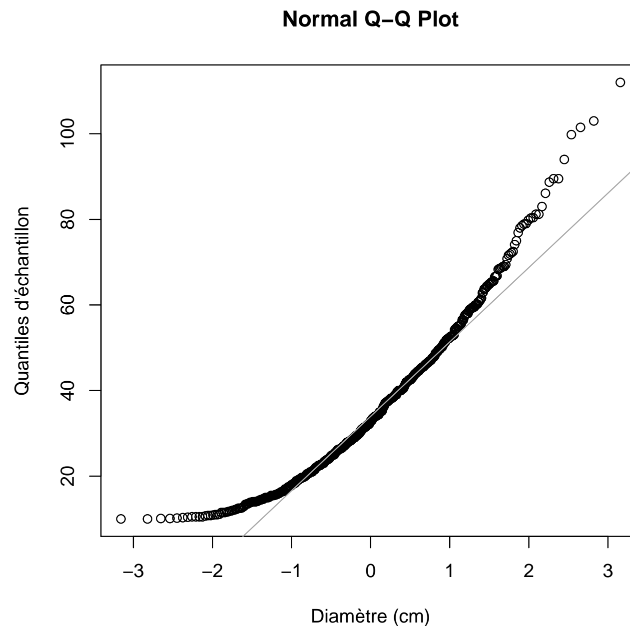


Figure 8.1 – Démonstration du tracé Normal des quantiles pour juger de l'asymétrie

### Pondérations

Rappelez-vous la fonction de volume, que nous avons utilisée en section 3.5, qui calcule le volume mesuré en pied pour les espèces d'arbre, diamètre et hauteur donnés.

```
> vol.fvs.ni.bdft <- function(spp, dbh.in, ht.ft) {
+     bf.params <- data.frame(species = c("WP", "WL", "DF", "GF",
+         "WH", "WC", "LP", "ES", "SF", "PP", "HW"), b0.small = c(26.729,
+         29.79, 25.332, 34.127, 37.314, 10.472, 8.059, 11.851,
+         11.403, 50.34, 37.314), b1.small = c(0.01189, 0.00997,
+         0.01003, 0.01293, 0.01203, 0.00878, 0.01208, 0.01149,
+         0.01011, 0.01201, 0.01203), b0.large = c(32.516, 85.15,
```

```
+         9.522, 10.603, 50.68, 4.064, 14.111, 1.62, 124.425, 298.784,
+         50.68), b1.large = c(0.01181, 0.00841, 0.01011, 0.01218,
+         0.01306, 0.00799, 0.01103, 0.01158, 0.00694, 0.01595,
+         0.01306))
+     dimensions <- data.frame(dbh.in = dbh.in, ht.ft = ht.ft,
+         species = as.character(spp), this.order = 1:length(spp))
+     dimensions <- merge(y = dimensions, x = bf.params, all.y = TRUE,
+         all.x = FALSE)
+     dimensions <- dimensions[order(dimensions$this.order, decreasing = FALSE),
+         ]
+     b0 <- with(dimensions, ifelse(dbh.in <= 20.5, b0.small, b0.large))
+     b1 <- with(dimensions, ifelse(dbh.in <= 20.5, b1.small, b1.large))
+     volumes <- b0 + b1 * dimensions$dbh.in^2 * dimensions$ht.ft
+     return(volumes)
+ }
```

La sylviculture est une des peu comparables disciplines qui régulièrement se soucie de la pondération d'échantillonnage. Des fonctionnalités sont disponibles dans **R** pour pondérer les données, voir, par exemple, la fonction `weighted.mean()`. Il y a aussi le paquetage `survey` de Thomas Lumley qui fournit un support très détaillé pour les opérations de pondération d'échantillon.

Cependant, je trouve qu'il est, la plupart du temps, plus facile de travailler directement avec les données après pondération. Nous en avons vu quelques approches dans le chapitre d'étude de cas (Chapitre 3, où nous avons utilisé des données de niveau arbre de parcelles à rayon variable pour obtenir une estimation du volume sylvestre d'une plantation. Souvenez-vous que nous avons calculé le facteur arbre constitué du nombre d'arbres par hectare pour chaque plantation et utilisé cette quantité pour calibrer les volumes d'arbre individuels.

```
> require(nlme)
> hd.lme <- lme(I(log(height.m)) ~ I(log(dbh.cm)) * species, random = ~I(log(dbh.cm)) |
+     plot, na.action = na.exclude, data = ufc)
> predicted.log.heights <- predict(hd.lme, na.action = na.exclude,
+     newdata = ufc)
> ufc$p.height.m[!is.na(ufc$dbh.cm)] <- exp(predicted.log.heights)
> ufc$p.height.m[!is.na(ufc$height.m)] <- ufc$height.m[!is.na(ufc$height.m)]
> ufc$vol.m3 <- with(ufc, vol.fvs.ni.bdft(species, dbh.cm * cm.to.inches,
+     p.height.m * m.to.feet) * bd.ft.to.m3)
> ufc$g.ma2 <- ufc$dbh.cm^2 * pi/40000
> ufc$tree.factor <- ufc.baf/ufc$g.ma2
> ufc$tree.factor[is.na(ufc$dbh.cm)] <- 0
> ufc$vol.m3.ha <- ufc$vol.m3 * ufc$tree.factor
```

À ce moment, nous avons le volume par hectare dans la parcelle représentée par chaque arbre mesuré en m3/ha. Tout en préservant l'unité des opérations que nous effectuons sur ces données, on aura la même heureuse interprétation. Pour obtenir le volume par hectare représenté par chaque parcelle, qui est l'unité d'échantillonnage, nous sommons le volume par hectare représenté par chaque arbre au niveau parcelle. Comme nous l'avons déjà vu, il existe plusieurs façons de faire cette sommation. Fait important, la probabilité de sélection de chaque arbre a été traitée dans la construction de l'estimation au niveau parcelle, et n'est plus pertinente pour une analyse plus approfondie.

Les variables continues peuvent également être transformées en variables ordinales pour une analyse plus approfondie en utilisant la fonction `cut()`. Par exemple, ici on découpe les diamètres d'arbre en classes de 20 cm, en faisant un nouveau facteur qui a ordonné les niveaux.

```
> ufc$dbh.20 <- cut(ufc$dbh.cm, breaks = (2:6) * 20)
> table(ufc$dbh.20)

  (40,60]   (60,80]  (80,100] (100,120]
      163        43        11         3
```

### 8.1.2 Qualitative

L'exploration de données la plus commune pour les variables « qualitatives » est la mise en tableaux. Il existe plusieurs moyens pour la mise en tableaux des données dans **R**. Leur différence est la souplesse des commandes et le type de sortie générée.

```
> table(ufc$species)

 DF  ES  GF  HW  LP  PP  SF  WC  WL  WP
 77   3 188   5   7   4  14 251  34  44

> tapply(ufc$species, ufc$species, length)

 DF  ES  GF  HW  LP  PP  SF  WC  WL  WP
 77   3 188   5   7   4  14 251  34  44

> aggregate(x = list(count = ufc$species), by = list(species = ufc$species),
+     FUN = length)

   species count
1       DF    77
2       ES     3
3       GF   188
4       HW     5
5       LP     7
6       PP     4
7       SF    14
8       WC   251
9       WL    34
10      WP    44
```

Ces tableaux peuvent être facilement convertis en figures (cf. Figure 8.2).

```
> plot(table(ufc$species), ylab = "Compte brut des troncs")
> plot(as.table(tapply(ufc$tree.factor/number.of.plots, ufc$species,
+     sum)), ylab = "Troncs par hectare")
```

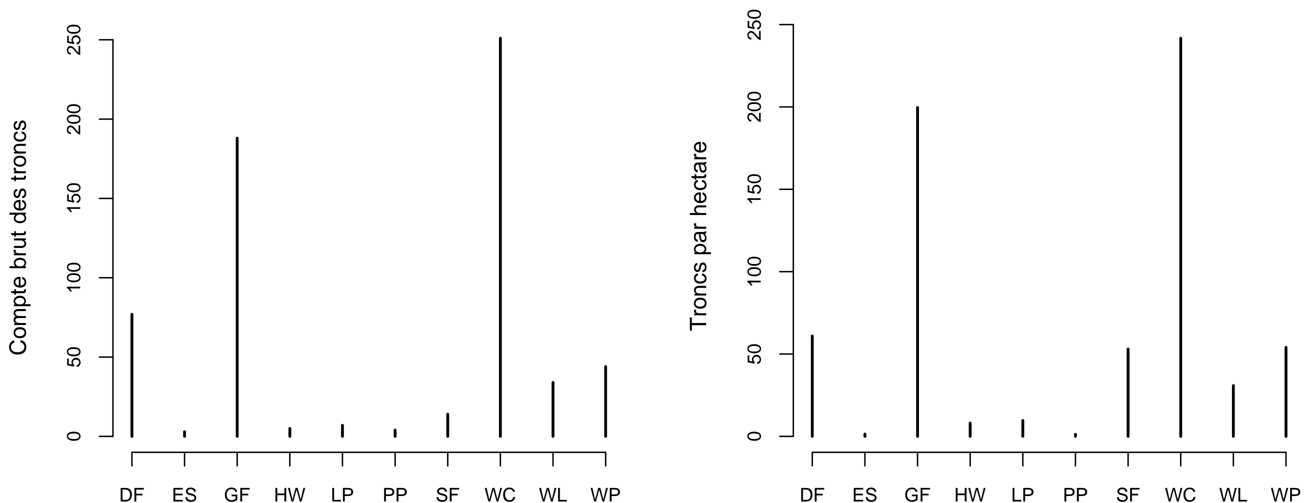


Figure 8.2 – Compte brut des troncs par espèce (à gauche) et facteur de pondération (à droite)

## 8.2 Plusieurs variables

Nous allons examiner trois cas qui offrent de l'information bi-variée ; quand il y a conditionnement sur une variable catégorielle (par exemple les espèces), et aussi lorsqu'il y a plusieurs variables intéressantes, qu'elles soient numériques ou catégorielles.

### 8.2.1 Quantitative/quantitative

Nous avons déjà vu un nuage de points de deux variables continues, et nous ne nous sommes pas souciés d'ajuster formellement des modèles à ce moment. Mais, la corrélation est rapide et utile une exploration d'accord entre deux variables continues. On notera que l'argument `na.rm` auquel nous sommes devenus habitués n'est pas utilisé dans cette fonction et que nous utilisons plutôt l'argument `use`. Voir `?cor` pour de plus amples détails.

```
> cor(ufc$dbh.cm, ufc$height.m, use = "complete.obs")

[1] 0.7794116
```

### 8.2.2 Quantitative/qualitative

Nous serons le plus souvent intéressés par l'exploration de la variable numérique (quantitative) conditionnée par la variable catégorielle (qualitative). Nous avons vu comment le faire de plusieurs moyens. La différence principale entre les fonctions `tapply()` et `aggregate()` réside dans la structure de la sortie. `tapply()` crée une liste nommée et `aggregate()` un data-frame avec les libellés en colonnes séparées. Aussi `aggregate()` calculera t'il la fonction nommée pour plusieurs variables d'intérêt.

```
> tapply(ufc$dbh.cm, ufc$species, mean, na.rm = TRUE)

      DF       ES       GF       HW       LP       PP       SF       WC
38.37143 40.33333 35.20106 20.90000 23.28571 56.85000 13.64286 37.50757
      WL       WP
34.00588 31.97273
```

```
> aggregate(x = list(dbh.cm = ufc$dbh.cm, height.m = ufc$height.m),
+     by = list(species = ufc$species), FUN = mean, na.rm = TRUE)

   species   dbh.cm height.m
1       DF 38.37143 25.30000
2       ES 40.33333 28.00000
3       GF 35.20106 24.34322
4       HW 20.90000 19.80000
5       LP 23.28571 21.83333
6       PP 56.85000 33.00000
7       SF 13.64286 15.41000
8       WC 37.50757 23.48777
9       WL 34.00588 25.47273
10      WP 31.97273 25.92500
```

### 8.2.3 Qualitative/qualitative

Les tableaux croisés numériques ou graphiques sont utiles pour explorer la relation entre plusieurs variables qualitatives.

```
> table(ufc$species, ufc$dbh.20)

     (40,60] (60,80] (80,100] (100,120]
  DF      27       5        2         0
  ES       1       0        0         0
  GF      47      12        4         0
  HW       0       0        0         0
  LP       0       0        0         0
  PP       0       1        1         0
  SF       0       0        0         0
  WC      71      23        4         1
  WL      10       1        0         0
  WP       7       1        0         2
```

Ce tableau peut être converti en tracé (cf. Figure 8.3) avec l'appel à la fonction `plot()` utilisant la fonction table comme seul argument. Ici nous utilisons le code pour sommer les facteurs arbres et obtenir des estimations non biaisées. Noter qu'il faut remplacer les données manquantes par des zéros.

```
> species.by.dbh <- as.table(tapply(ufc$tree.factor/number.of.plots,
+     list(ufc$dbh.20, ufc$species), sum))
> species.by.dbh[is.na(species.by.dbh)] <- 0
> par(mar = c(4, 4, 3, 1), las = 1)
> plot(species.by.dbh, main = "Espèces par classe de diamètre")
```

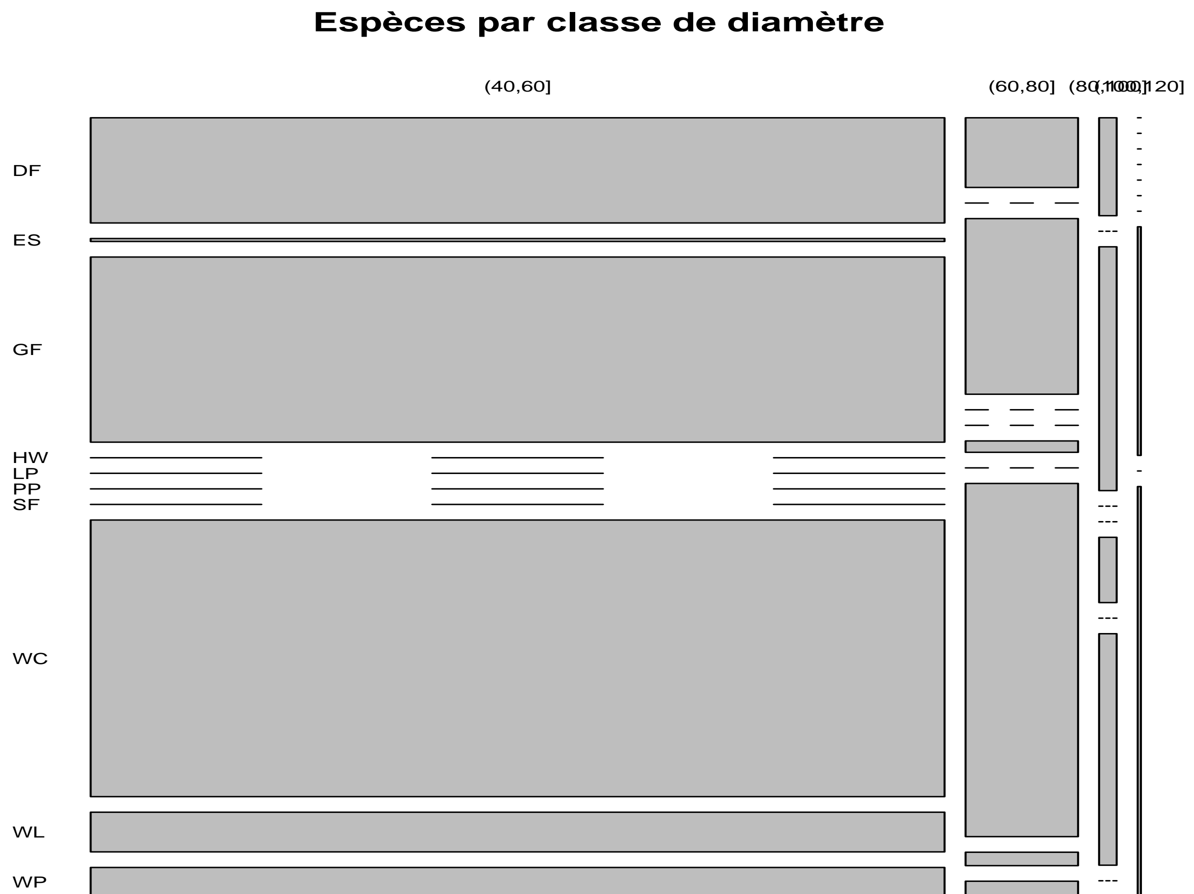


Figure 8.3 – Troncs/ha par espèce selon les diamètres en classes de 20 cm.

# Chapitre 9

# Graphiques

L'un des principaux argument de vente de **R** est qu'il dispose de plus de capacités graphiques que bon nombre d'autres produits commerciaux. Que vous souhaitiez des graphiques rapides qui vous aident à comprendre la structure de vos données, ou des graphiques la graphique de qualité publication qui communiquent précisément votre message à vos lecteurs, R suffira [1].

Les graphiques sont contrôlés par des scripts, et commencent à un niveau très simple. Si vous avez `dbh.cm` et `height.m` dans votre data-frame `ufc` [2], par exemple :

```
> plot(ufc$dbh.cm, ufc$height.m)
```

ouvrira une fenêtre graphique et tracera un nuage de points de la hauteur en fonction du diamètre pour les données de Upper Flat Creek en nommant les axes de façon appropriée. Un petit ajout fournira des étiquettes plus informatives (cf. Figure 9.1). Notez également qu'il y a plusieurs façons de faire appel à un tracé, parfois la construction suivante peut être plus pratique.

```
> plot(height.m ~ dbh.cm, data = ufc, xlab = "Diamètre (cm)",
+      ylab = "Hauteur (m)")
```

La commande `plot()` offre une grande variété d'options pour personnaliser le graphique. Chacun des arguments suivants peuvent être utilisés dans une déclaration `plot()`, séparément ou ensemble, séparés par des virgules. En d'autres termes, l'instruction de tracé ressemblera à

```
> plot(x, y, xlab="Une étiquette pour l'axe des x", ... )
```

- `xlim=c(a,b)` fixera les limites inférieure et supérieure de l'axe des $x$ à $a$ et $b$, respectivement. Noter qu'il faut connaître $a$ et $b$ pour faire ce travail.
- `ylim=c(a,b)` fixera les limites inférieure et supérieure de l'axe des $y$ à $a$ et $b$, respectivement. Noter qu'il faut connaître $a$ et $b$ pour faire ce travail.
- `xlab="Étiquette pour l'axe des x, ici"`
- `ylab="Étiquette pour l'axe des y, ici"`
- `main="Titre du tracé, ici"`
- `col="red"` fait que tous les points sont rouges. C'est particulièrement intéressant pour les tracés superposés.

1. Mon frère est dans le marketing.
2. Vous pouvez le tester avec `str()`. S'ils sont absents, le code pour les calculer et ajouter est en page 13.

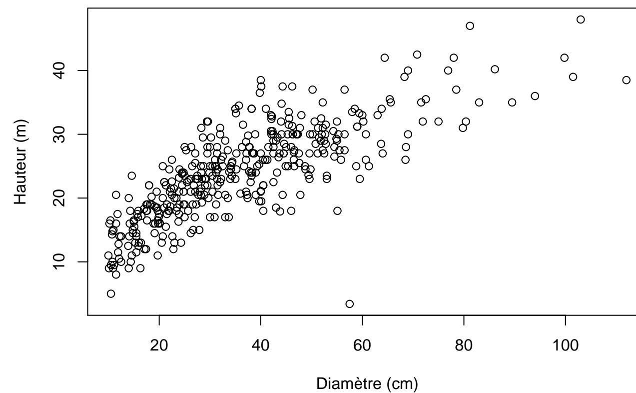


Figure 9.1 – Tracé de la hauteur en fonction du diamètre pour les données sylvestres de Upper Flat Creek.

## 9.1 Paramètres d'organisation

À partir d'ici, nous avons une grande souplesse pour choisir le symbole, la couleur, la taille, l'étiquetage d'axe, la superposition, etc. Nous présenterons quelques capacités graphiques en temps voulu. Ces options peuvent être étudiées grâce à `?par`. La meilleure implémentation est d'ouvrir un nouveau jeu de paramètres, de créer le graphique, et de restaurer l'état original, par le biais des simples mais plutôt étranges commandes :

```
> opar <- par( {fixations des paramètres ici, séparés par des virgules} )
> plot( {instructions de tracé ici} )
> par(opar)
```

La raison d'un tel fonctionnement est que quand la fonction `par()` est utilisée pour fixer un nouveau paramètre, elle retourne de façon invisible la valeur de ce paramètre. Ainsi, le code

```
> opar <- par(las = 1)
```

change t'il le paramètre `las` en 1 et enregistre `las=0` dans `opar`. L'instruction suivante

```
> par(opar)
```

rétablit donc le paramètre `las` à 0.

Il existe plusieurs options pour modifier la présentation des graphiques sur la page. Elles peuvent être utilisées en conjonction avec d'autres. Il y a beaucoup plus que je note ci-dessous, mais ce sont celles que je finis par utiliser le plus souvent. Mon appel à `par` le plus commun est quelque chose comme :

```
> opar <- par(mfrow=c(3,2), mar=c(4,4,1,1), las=1)
> plot( ...
> par(opar)
```

- `par(mfrow=c(a,b))` où $a$ et $b$ sont des entiers créera une matrice de graphiques sur une page, avec $a$ lignes et $b$ colonnes, remplie par ligne(s).`mfcol()` remplit la page par colonne(s).
- `par(mar=c(s,w,n,e))` créera un espace de caractères en taille autour de la marge interne du(des) tracé(s).
- `par(oma=c(s,w,n,e))` créera un espace de caractères en taille autour de la marge externe du(des) tracé(s).
- `par(las=1)` fait tourner les étiquettes d'axe $y$ en horizontal plutôt que vertical.
- `par(pty="s")` oblige l'aspect du graphique à être carré ou rectangulaire. L'alternative est un aspect mutable, par défaut, `pty="m"`.
- `par(new=TRUE)`, quand il est inséré entre les tracés, permet de tracer le suivant sur le même graphique en superposé. Les axes ne correspondront pas sauf si on les y force. Accroître les paramètres de `mar` pour le second tracé le forcera à être *interne* au premier.
- Plusieurs facteurs de grossissement des caractères peuvent aussi être fixés.

Vous pouvez interroger votre graphique précédent avec

```
> par("usr")
```

```
[1] 5.920 116.080 1.616 49.784
```

qui vous donne les limites d'axes utilisées (à comparer à celles de la figure 9.1)

## 9.2 Agrémentation d'un graphique

Un tracé traditionnel, une fois créé, peut être élargi en utilisant l'un des divers outils.

L'infrastructure du tracé peut être modifiée. Par exemple, les axes peuvent être omis dans le graphique initial avec `(axes = FALSE)` et ensuite ajoutés en utilisant `axis()`, qui donne plus de souplesse et de contrôle sur l'emplacement et le format des graduations, des étiquettes d'axe et aussi leur contenu. Il est à noter que `axis()` ne prévoit comme information dans l'axe que la partie des données de l'étendue, afin de réduire l'encombrement inutile du tracé.

Le cadre du tracé peut d'habitude être ajouté en utilisant la fonction `box()`. Le texte peut être situé dans les marges du tracé, pour libeller certaines zones ou pour augmenter les étiquettes d'axe, en utilisant la fonction `mtext()`. Une légende peut être ajoutée en utilisant la fonction `legend()`, qui comprend une très utile routine de placement de légende, comme indiqué ci-dessous. Des ajouts peuvent également être apportés au contenu du tracé, avec les fonctions `points()`, `lignes()`, et `abline()`. Un certain nombre de ces différentes étapes sont montrées sur la figure 9.2.

1. On commence par créer l'objet tracé qui fixe les dimensions de l'espace mais omet de tracer les objets pour le moment.

```
> par(las = 1, mar = c(4, 4, 3, 2))
> plot(ufc$dbh.cm, ufc$height.m, axes = FALSE, xlab = "",
+     ylab = "", type = "n")
```

2. Ensuite on ajoute les points. Avec des couleurs différentes selon les arbres.

```
> points(ufc$dbh.cm, ufc$height.m, col="darkseagreen4")
> points(ufc$dbh.cm[ufc$height.m < 5.0],
+        ufc$height.m[ufc$height.m < 5.0], col="red")
```

3. On ajoute les axes. Ce sont les plus simples appels, nous avons une plus grande souplesse que montré ici.

```
> axis(1)
> axis(2)
```

4. On ajoute les étiquettes aux axes avec le texte en marge (basculé à la verticale pour l'axe des $y$).

```
> par(las = 0)
> mtext("Diamètre (cm)", side = 1, line = 3, col = "blue")
> mtext("Hauteur (m)", side = 2, line = 3, col = "blue")
```

5. Enveloppe le tracé dans le cadre traditionnel.

```
> box()
```

6. Enfin, ajout de la légende.

```
> legend("bottomright",
+        c("Arbre normal","Arbre bizarre"),
+        col=c("darkseagreen3","red"),
+        pch=c(1,1),
+        bty="n")
```

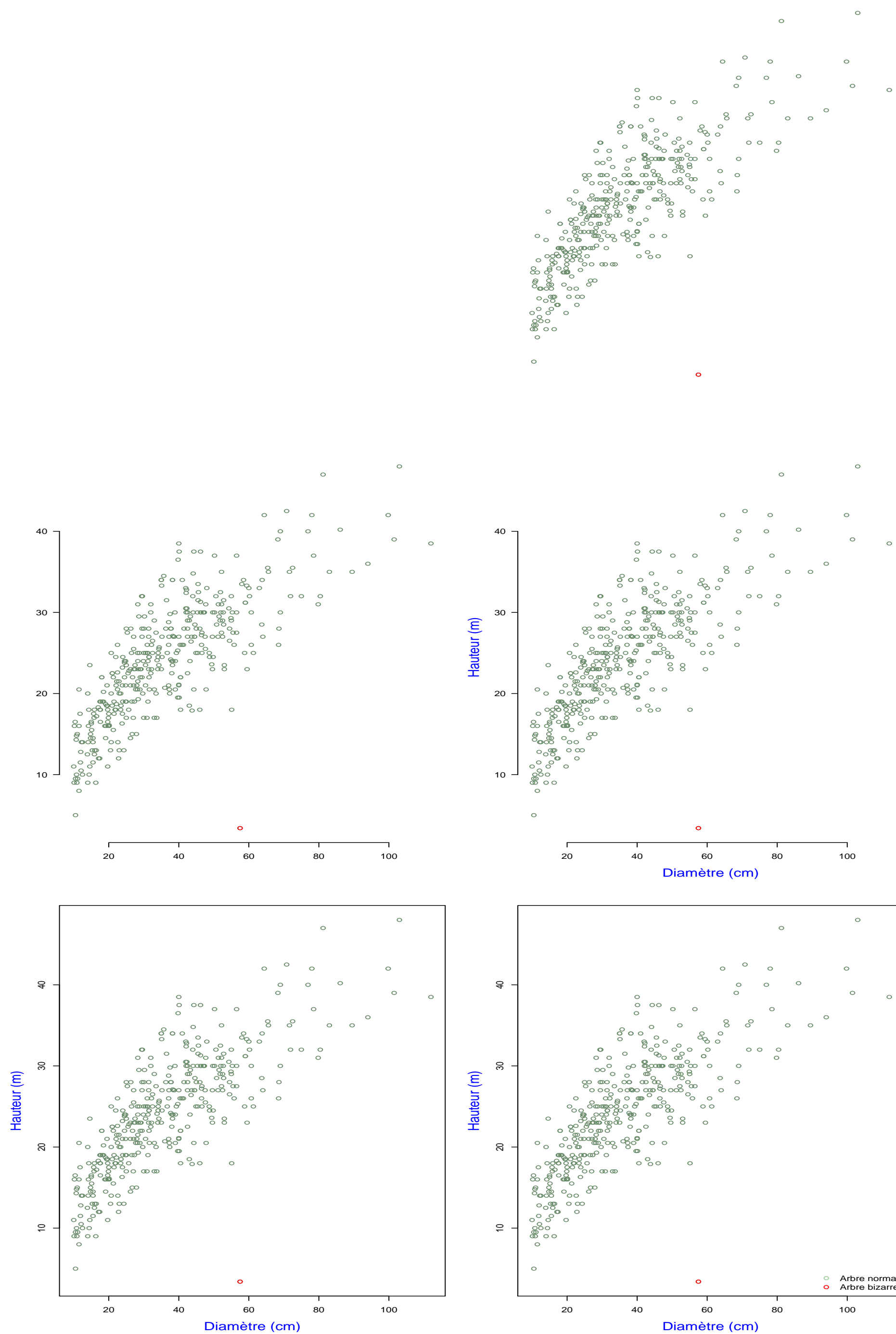


Figure 9.2 – Construction d'un graphique par ses composants.

## 9.3 Permanence

Produire des graphiques plus permanents est toujours aussi simple. Par exemple pour créer le graphique comme un fichier **pdf**, qui peut être importé dans des documents variés on écrit ce qui suit :

```
> pdf(file="../graphics/graphic.pdf")
> plot(ufc$dbh.cm, ufc$height.m)
> abline(lm(height.m ~ dbh.cm, data=ufc), col="red")
> dev.off()
```

Cela placera le **pdf** dans votre répertoire de graphiques. C'est une option particulièrement bonne si les graphiques que vous voulez produire prennent d'ordinaire plusieurs pages, par exemple, si vous les produisez en boucle. Le format **pdf** est bien accepté par l'Internet, pour que les graphiques soient portables. Le PostScript encapsulé est également pris en charge.

Sous Windows, vous pouvez également copier directement de la fenêtre graphique dans le presse-papiers soit comme un méta fichier soit comme une image **bmp** (bitmap). Ou encore être collé directement dans un document Word, par exemple. Autrement, en utilisant une approche similaire à celle indiquée ci-dessus, on peut créer une image **jpeg** qui peut aussi être importée dans des documents Word. Mon expérience a été que l'éclat de la couleur souffre en utilisant **jpeg**, et certaines mesures d'amélioration pourraient être nécessaires.

## 9.4 Mises à jour

Le plaisir des graphiques avancés en **R** est que toutes les bonnes images possibles peuvent être faites avec les outils déjà mentionnés. De plus, comme l'interface est un script, il est très facile de prendre les graphiques là où ils ont été créés dans un but précis et les faire évoluer à une autre fin. Le bouclage est un choix judicieux pour contrôler les paramètres et obtenir un résultat informatif et intéressant. Par exemple, la figure 9.5.1 est précédée du code qui la construit :

```
> opar <- par(oma = c(0, 0, 0, 0), mar = c(0, 0, 0, 0))
> x1 <- rep(1:10, 10)
> x2 <- rep(1:10, each = 10)
> x3 <- 1:100
> numeros.couleurs.interessants <- c(1:152, 253:259, 362:657)
> tracez.nous <- sample(numeros.couleurs.interessants, size = max(x3))
> plot(x1, x2, col = colors()[tracez.nous], pch = 20, cex = 10,
+     axes = F, xlim = c(0, 10.5), ylim = c(0, 10))
> text(x1, x2 - 0.5, colors()[tracez.nous], cex = 0.3)
> text(x1 + 0.4, x2 - 0.4, tracez.nous, cex = 0.5)
> par(opar)
```

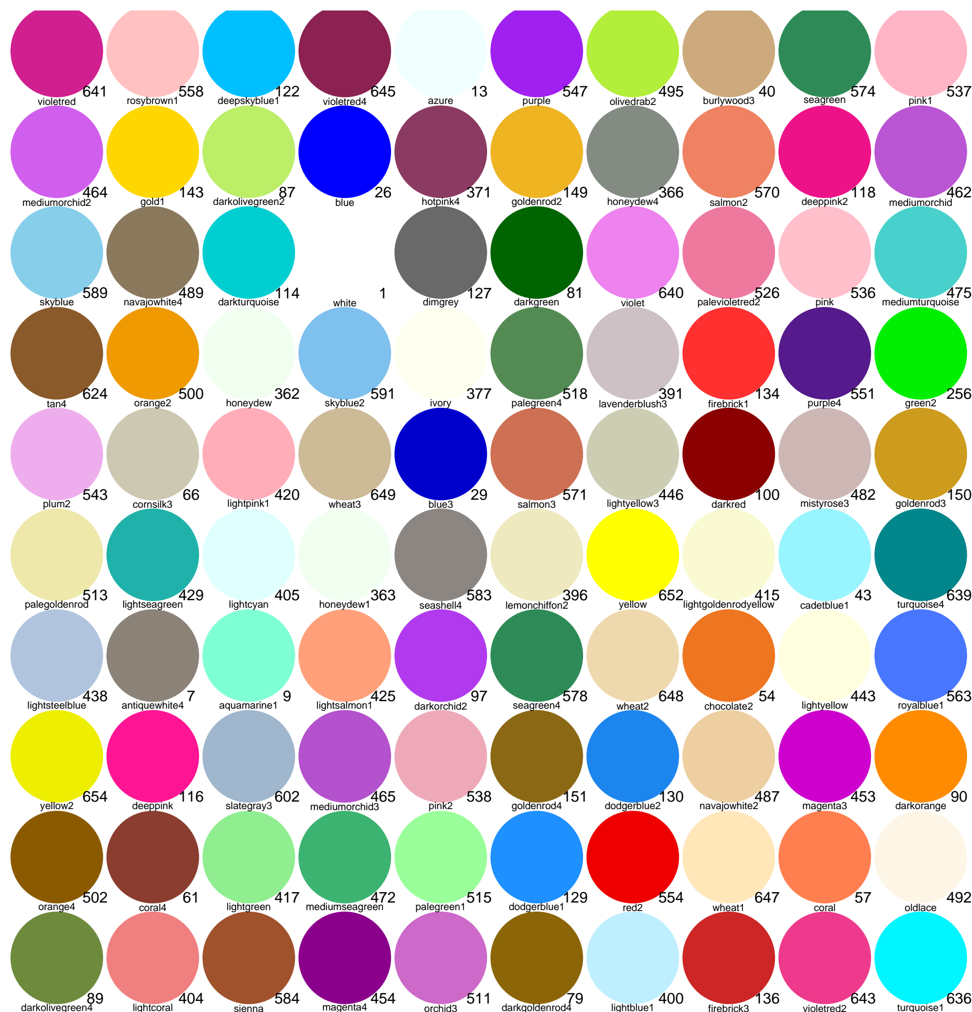


Figure 9.3 – Tracé aléatoire de points colorés

## 9.5 Défis courants

Cette section est destinée à montrer comment résoudre les problèmes graphiques avec **R**.

### 9.5.1 Barres d'erreur

On nous demande souvent de fournir des graphiques avec des barres d'erreur. Nous pouvons utiliser la fonction `arrows()` pour avoir les barres et la vectorisation nous gardera un code simple. La figure **??** montre les moyennes et les intervalles de confiance à 95% (non pondéré) des diamètres d'arbre par espèce.

```
> moyennes <- tapply(ufc$dbh.cm, ufc$species, mean, na.rm = TRUE)
> errstd <- tapply(ufc$dbh.cm, ufc$species, sd, na.rm = TRUE)/sqrt(tapply(ufc$dbh.cm,
+     ufc$species, length))
> limy <- range(c(moyennes + 1.96 * errstd, moyennes - 1.96 * errstd))
> par(las = 1, mar = c(4, 4, 2, 2))
> plot(1:10, moyennes, axes = FALSE, ylim = limy, xlab = "Espèces",
+     ylab = "Diam. (cm)", cex=0.8)
> axis(1, at = 1:10, labels = levels(ufc$species))
> axis(2)
> arrows(1:10, moyennes - 1.96 * errstd, 1:10, moyennes + 1.96 *
+     errstd, col = "darkgrey", angle = 90, code = 3, length = 0.1)
```

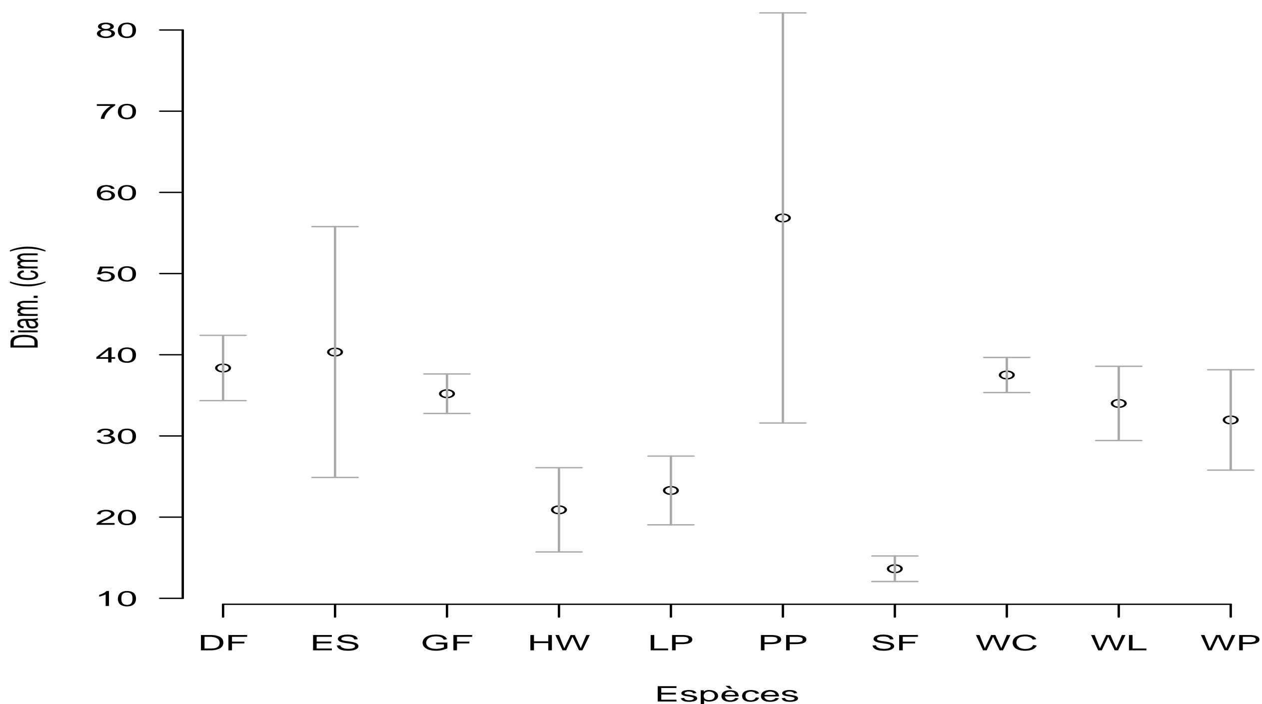


Figure 9.4 – Moyennes et intervalles de confiance à 95% des diamètres d'arbre par espèce.

### 9.5.2 Graphiques colorés par groupe

Intégrer la couleur utile dans un autre défi courant. Il existe des paquetages qui contiennent des fonctions de le faire rapidement et facilement (`lattice`, voir la section 9.6.1, et `car`, pour n'en citer que deux), mais il vaut mieux voir comment le faire avec les outils disponibles.

Utiliser toutes les espèces disponibles peut faire de dégâts, donc limitons nous aux arbres en provenance des quatre plus communes. Le code ci-dessous identifie les quatre espèces les plus courantes et construit un petit data-frame qui contient seulement les arbres de ces espèces.

```
> les.quatre <- levels(ufc$species)[order(table(ufc$species), decreasing = TRUE)][1:4]
> ufc.reduit <- ufc[ufc$species %in% les.quatre, ]
> ufc.reduit$species <- factor(ufc.reduit$species)
```

Maintenant nous pouvons présenter le tracé avec le data-frame réduit, le code suivant et la figure 9.5 :

```
> mes.couleurs <- c("red", "blue", "goldenrod", "darkseagreen4")
> par(las = 1, mar = c(4, 4, 3, 2))
> plot(ufc.reduit$dbh.cm, ufc.reduit$height.m, xlab = "Diam. (cm)",
+     ylab = "Haut. (m)", col = mes.couleurs[ufc.reduit$species])
> legend("bottomright", legend = levels(ufc.reduit$species), col = mes.couleurs,
+     lty = rep(NULL, 4), pch = rep(19, 4), bty = "n")
```

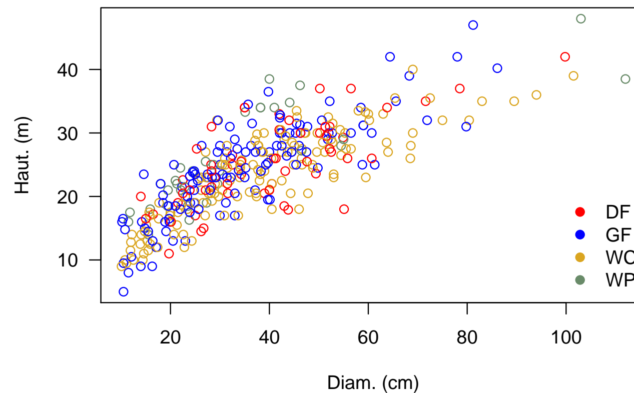


Figure 9.5 – Tracé de la hauteur en fonction du diamètre pour les arbres d'espèces les plus courantes repérées par la couleur.

## 9.6 Contributions

Les contributeurs à **R** ont écrit des paquetages qui facilitent la construction des graphiques. Nous allons explorer ici certains d'entre eux. Leur documentation est un défi car le code est en constante révision et développement.

### 9.6.1 Treillis

Le treillis est un outil plutôt formel de virtuosité graphique. Ils permettent une grande souplesse dans la production de graphiques conditionnels. L'implémentation du treillis dans **R** est nommée *lattice* et elle est écrite/maintenue par Deepayan Sankar.

On charge le paquetage `lattice` au moyen de la fonction `require`, comme c'est expliqué en détail Section 4.5.

```
> require(lattice)
```

Le but de `lattice` est de simplifier la production de graphiques simples et de haute qualité permettant l'exploration de données multi-dimensionnelles.

Nous avons déjà vu l'utilisation de ces graphiques dans le chapitre de l'Étude de cas (chapitre 3) où nous avons tracé la hauteur en fonction du diamètre pour chaque espèce, ainsi que les distributions de diamètre aussi pour chaque espèce. Nous pouvons obtenir d'autres graphiques utiles avec un minimum d'embaras, par exemple, en se concentrant sur les quatre espèces les plus prolifiques des données ufc (figure 9.6).

```
> ufc <- read.csv("../data/ufc.csv")     # ufc est un data-frame
> ufc$diam.cm <- ufc$dbh/10              # diam est en cm
> ufc$haut.m <- ufc$height/10            # haut est en mètres
tres
> ufc$haut.m[ufc$haut.m < 0.1] <- NA
> ufc <- ufc[complete.cases(ufc), ]
> ufc$species[ufc$species %in% c("F", "FG")] <- "GF"
> ufc$especes <- factor(ufc$species)
> les.quatre <- levels(ufc$especes)[order(table(ufc$especes), decreasing = TRUE)][1:4]
```

Les panneaux dans la liste ci-dessous sont directement reportés sur la figure 9.6 et ne correspondent pas nécessairement à la position de ce que vous verrez à l'écran.

**Panneau en haut à gauche :**

```
> densityplot(~diam.cm | especes, data = ufc,
+             subset = especes %in% les.quatre)
```

**Panneau en haut à droite :**

```
> bwplot(diam.cm ~ especes, data = ufc, subset = especes %in% les.quatre)
```

**Panneau en bas à gauche :**

```
> histogram(~diam.cm | especes, data = ufc,
+             subset = especes %in% les.quatre)
```

**Panneau en bas à droite :**
On aurait pu charger une image et utiliser `contourplot()` pour représenter les courbes de densité ($m^3/ha$).

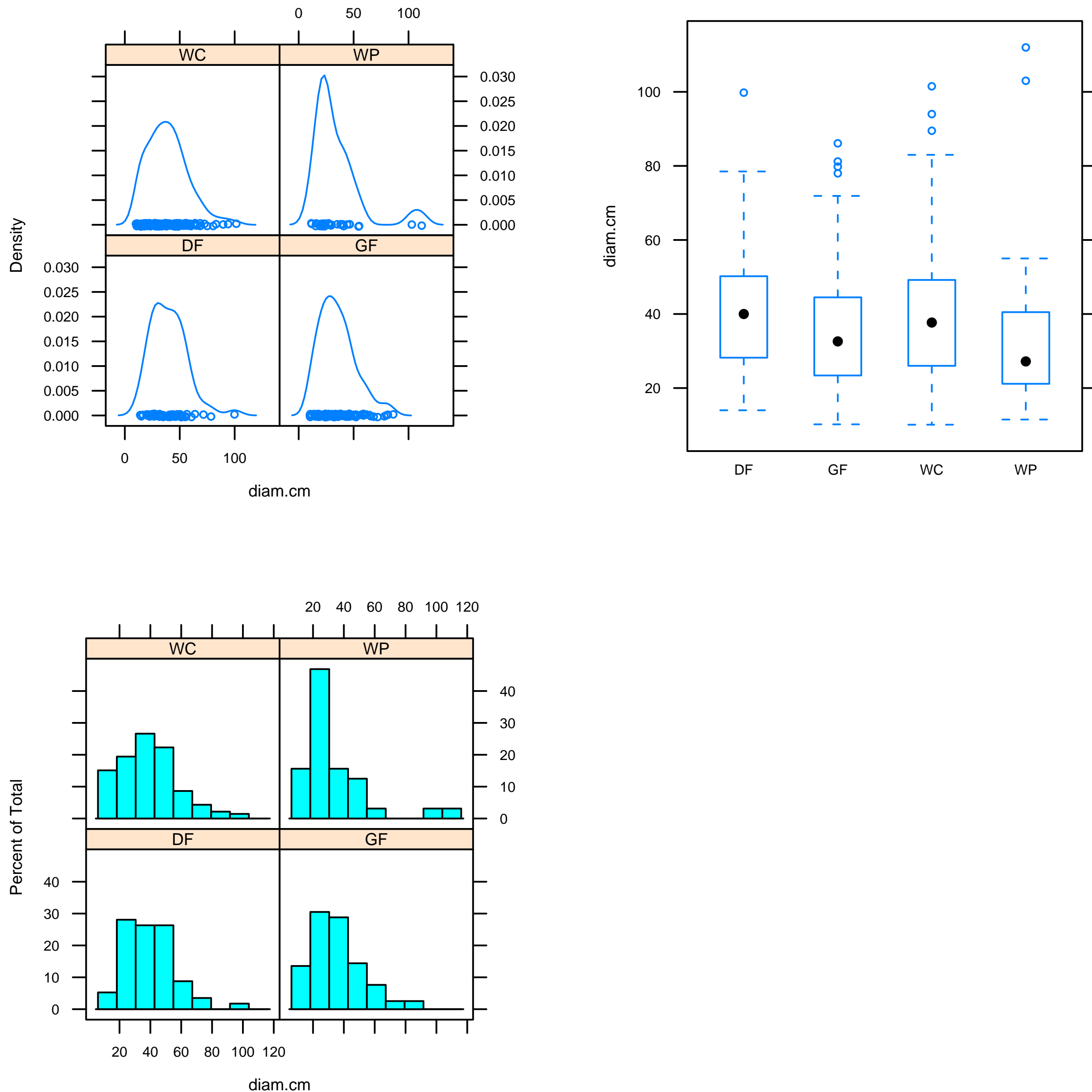


Figure 9.6 – Exemple de tracé lattice, montrant l'information sur le diamètre conditionnée par espèces. Le panneau du haut à gauche montre les courbes de densité empirique des diamètres, le panneau d'à côté montre des boîtes à moustaches, en bas à gauche, des histogrammes et son voisin est laissé vide pour les raisons ci-dessus.

Les graphiques produits par `lattice` sont hautement personnalisable. Par exemple, si nous devions tracer la hauteur en fonction de la hauteur prédite pour les six espèces de plus grand nombre d'arbres, utilisant le modèle d'attribution

à effets mixtes du chapitre 3, et ajouter quelques droites d'illustration au graphique, nous pourrions le faire avec le code ci-dessous. Tout d'abord, nous ajustons un modèle adapté et extrayons les prévisions, à être utilisés comme des attributions.

```
> require(nlme)
> require(nlme)
> hd.lme <- lme(I(log(height.m)) ~ I(log(dbh.cm)) *
+     species, random = ~I(log(dbh.cm)) | plot,
+     data = ufc)
> ufc$p.height.m <- exp(predict(hd.lme, level = 1))
> top.six <- levels(ufc$species)[order(table(ufc$species),
+     decreasing = TRUE)][1:6]
> require(MASS)
```

Ensuite nous construisons le graphique, en ajoutant une nouvelle fonction qui sera exécutée dans chaque panneau. Cette version produit un diagramme de dispersion représenté par des points, ajoute une diagonale 1 : 1, une droite de régression linéaire ajustée aux données du panneau et une courbe de lissage ajusté local (loess) noire.

```
xyplot(height.m ~ p.height.m | species, data = ufc,
+      xlab = "Hauteur prédite (m)",
+      ylab = "Hauteur mesurée (m)",
+      panel = function(x, y) {
+       panel.xyplot(x, y)
+       panel.abline(0, 1, col = "blue", lty = 2)
+       panel.abline(lm(y ~ x), col = "red")
+       panel.abline(rlm(y ~ x), col = "purple")
+       panel.loess(x, y, col = "black")
+      },
+      subset = species %in% top.six
+      )
```

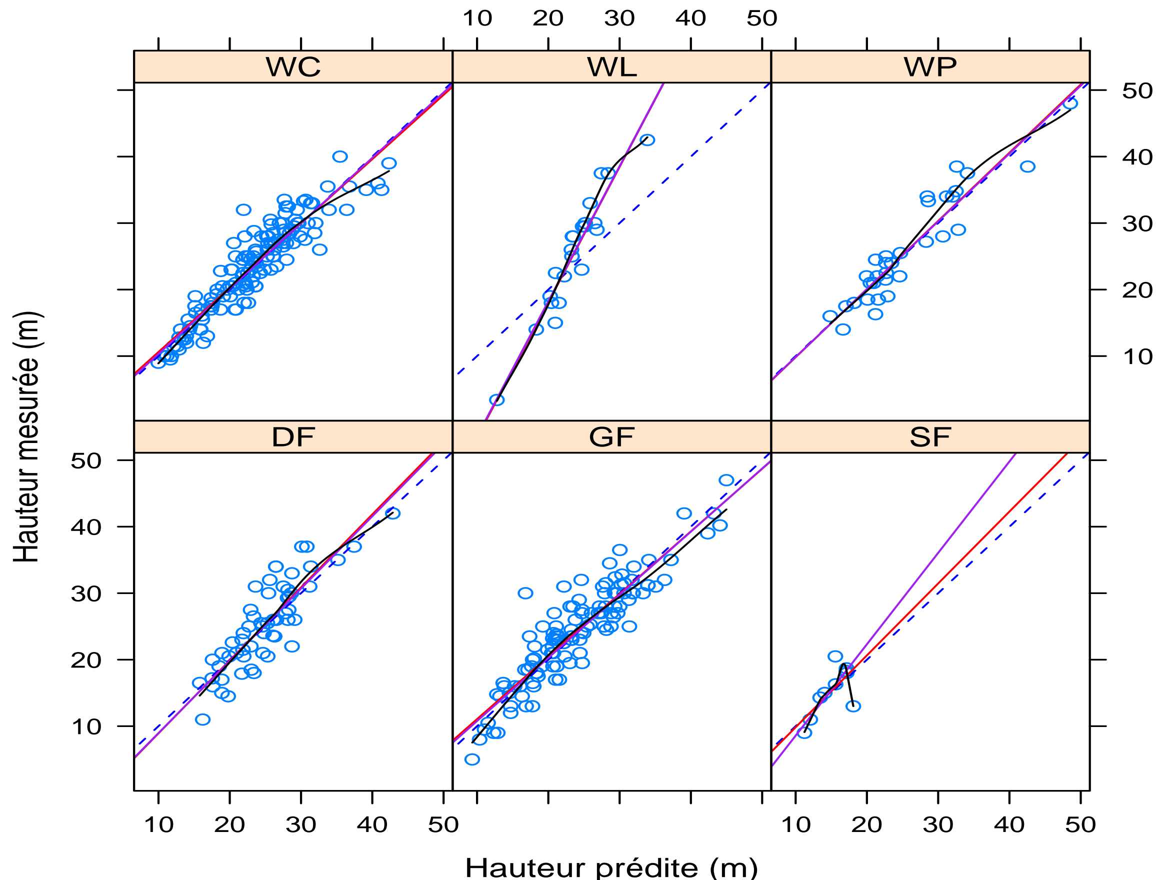


Figure 9.7 – Un tracé lattice de la hauteur mesurée en fonction de la hauteur prédite par espèces les plus fournies. La droite pointillée est la première diagonale, la rouge pleine est de régression robuste et la courbe noire de lissage loess.

Si vous êtes inquiet pour le panneau des mélèzes de l'ouest (WL), essayez le même exercice en utilisant une régression linéaire au lieu du modèle d'attribution à effets mixtes ! Il faudrait faire les modifications suivantes des valeurs prédites :

```
> hd.lm <- lm(I(log(height.m)) ~ I(log(dbh.cm)) * species, data = ufc)
> ufc$p.height.m <- exp(predict(hd.lm))
```

On peut changer l'ordre des panneaux en utilisant l'option `index.cond`. On peut aussi ajouter d'autres commandes dans chacun d'eux. `?xyplot` est très utile pour nous ici, il donne une information détaillée sur les diverses options et des exemples attractifs que l'on peut adapter à son utilisation personnelle.

Le code source est toujours en développement rapide. Par exemple, pour obtenir un tracé en treillis avec des points, des droites de régression de meilleur ajustement et des lissages superposés nous n'avons besoin que du code suivant :

```
> xyplot(height.m ~ p.height.m | species, data = ufc,
+        xlab = "Hauteur prédite (m)",
+        ylab = "Hauteur mesurée (m)",
+        type = c("p", "r", "smooth"),
+        subset = species %in% top.six
+ )
```

## 9.7 Grammaire des graphiques

La *Grammaire des graphiques* en est à sa seconde édition (Wilkinson [27]). Hadley Wickham a développé en **R** des outils pour réaliser quelques-un de ses principes dans un format d'utilisation pratique. Le code est en version 2, la version originale est déphasée (caduque, *deprecated* en langue **R**) et l'on utilise le paquetage `ggplot2`. Ce paquetage est toujours en grand développement, mais il pourrait contribuer à simplifier la communication statistique.

```
> require(ggplot2)
```

Nous traçons de simples développements en figure 9.8.

1. On démarre en créant l'espace de tracé et on ajoute des points.

   ```
   > ggplot(ufc, aes(y = height.m, x = dbh.cm)) + geom_point()
   ```

2. On ajoute une courbe de régression avec les espaces d'erreur.

   ```
   > ggplot(ufc, aes(y = height.m, x = dbh.cm)) + geom_point() +
   +     stat_smooth(fill = alpha("blue", 0.2),
   +     colour = "darkblue", size = 2)
   ```

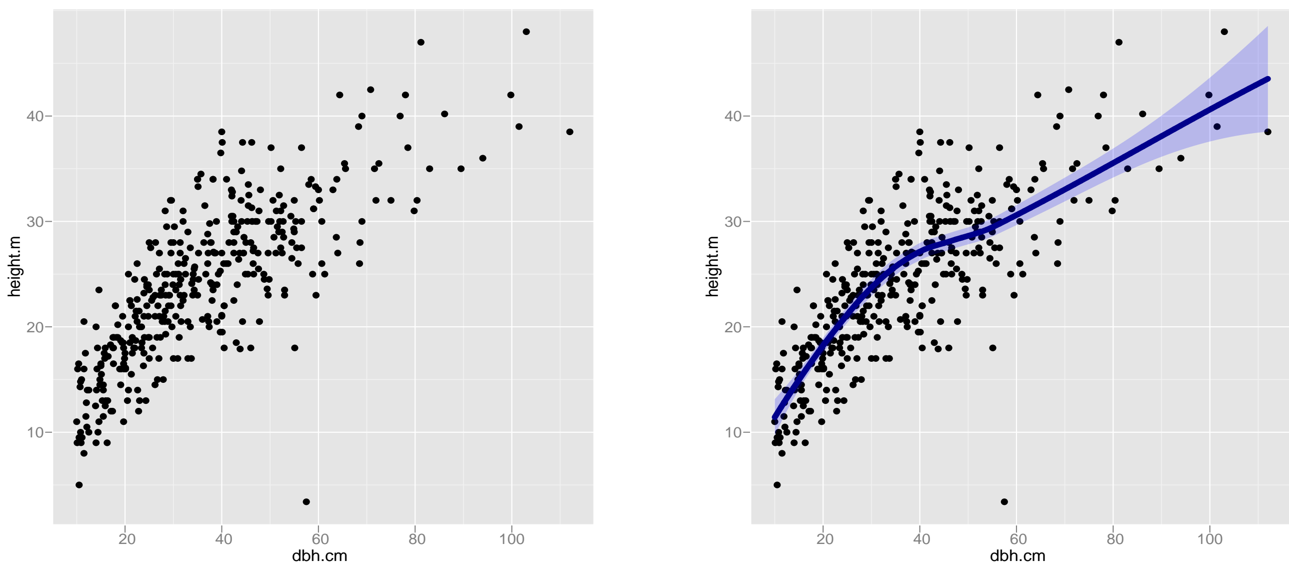


Figure 9.8 – Construction d'un graphique avec des composants ggplot.

# Chapitre 10

# Régression linéaire

Ce chapitre se concentre sur les outils **R** qui peuvent être utilisés pour l'ajustement d'une régression linéaire ordinaire, sous diverses formes. Je ne présentent pas ici la théorie car elle a été beaucoup plus efficacement traitée ailleurs.

## 10.1 Préparation

Les données sont les données sylvestres de la forêt de la zone Upper Flat Creek d'expérience sylvestre de l'Université de l'Idaho. Elles ont été collectées comme échantillon aléatoire systématique de la variable des lots. Nous avons des mesures de diamètre prises à la hauteur de 1,37 m sur l'ensemble de l'échantillon des arbres et des mesures de hauteur sur un sous-ensemble. Notre objectif immédiat est de construire une relation entre la hauteur et le diamètre pour nous permettre de prédire le premier à partir de ce dernier. Pour le moment, nous allons ignorer le mécanisme par lequel les arbres ont été sélectionnés pour chaque phase de mesure, et supposer qu'ils ont été sélectionnés selon une même probabilité. Certaines parcelles n'ont pas de mesures d'arbre ; pour s'assurer qu'ils sont correctement comptabilisés, un arbre « vide » a été utilisés pour les représenter.

Lire les données, après les avoir examinées dans une feuille de calcul.

```
> rm(list = ls())
> ufc <- read.csv("../data/ufc.csv")
```

Quelles en sont les dimensions ?

```
> dim(ufc)
```

```
[1] 627  13
```

Quels sont les noms des variables ?

```
> names(ufc)
```

```
[1] "plot"    "tree"    "species" "dbh"     "height"
```

Faisons-en un cliché – Je jette généralement un coup d'œil aux premières 100–200 observations et choisis quelques jeux de 100 au hasard. Par exemple `ufc[1 :100,]` et `ufc[201 :300,]`. Ces crochets sont fabuleusement utiles, ils permettent la manipulation des données dans un format très simple.

```
> ufc[1:10, ]
```

```
   plot tree species dbh height
1     1    1          NA     NA
2     2    1      DF 390    205
3     2    2      WL 480    330
4     3    1      WC 150     NA
5     3    2      GF 520    300
6     3    3      WC 310     NA
7     3    4      WC 280     NA
8     3    5      WC 360    207
9     3    6      WC 340     NA
10    3    7      WC 260     NA
```

Francisons et faisons quelques conversions d'unités : `diam`ètre à 1,37 m converti en cm et `haut`eur en mètres.

```
> ufc$diam.cm <- ufc$dbh/10
> ufc$haut.m <- ufc$height/10
```

Maintenant comptons les arbres de chaque espèces :

```
> table(ufc$species)

     DF  ES   F  FG  GF  HW  LP  PP  SF  WC  WL  WP
 10  77   3   1   2 185   5   7   4  14 251  34  44
```

Noter les 10 non-arbres qui marquent les lots vides – nous n'en avons pas besoin pour ce propos. Rendons tout cela propre. On le fait ici en fixant comme manquantes les espèces plutôt que la valeur courante qui est vide. (C'est en fait redondant avec la commande qui suit, on peut l'ignorer si on le souhaite, mais on s'en sert pour franciser).

```
> ufc$species[is.na(ufc$dbh)] <- NA
> ufc$especes <- factor(ufc$species)
```

Le F(ir) et le FG( ?) sont probablement mis pour GF (Grand Fir). Faisons le changement.

```
> ufc$species[ufc$species %in% c("F", "FG")] <- "GF"
> ufc$especes <- factor(ufc$species)
> table(ufc$especes)

 DF  ES  GF  HW  LP  PP  SF  WC  WL  WP
 77   3 188   5   7   4  14 251  34  44
```

Nous redéfinissons le facteur `espece` pour retirer les niveaux vides. On retire les arbres avec des mesures de hauteur manquantes.

```
> ufc <- ufc[!is.na(ufc$haut.m), ]
```

On trace le graphique des données d'arbre (cf. figure 9.1). C'est toujours informatif. Nous voyons une sorte de courbure, et certains points particuliers. Qu'est-ce qui pourrait les avoir causés ? Néanmoins, c'est un tracé encourageant - nous devrions être à même de corriger la hauteur dans une bonne étendue.

Avec quelle sorte de variabilité sous-tendue devons-nous travailler ?

```
> sd(ufc$haut.m, na.rm = T)

[1] 7.491152
```

Voici la souplesse de `tapply`, montrant un écart-type *par espèces* :

```
> tapply(ufc$haut.m, ufc$especes, sd, na.rm = TRUE)

      DF       ES       GF       HW       LP       PP       SF       WC
6.633357 5.656854 7.610491 2.489980 5.057997 9.899495 3.656486 6.972851
      WL       WP
8.848308 9.177620
```

## 10.2 Ajustement

Maintenant, nous allons voir si nous pouvons produire une sorte de régression pour prédire la hauteur en fonction du diamètre. Il est à noter que **R** est orienté objet, ainsi nous pouvons créer des objets qui sont eux-mêmes des ajustements de modèle.

```
> hd.lm.1 <- lm(haut.m ~ diam.cm, data = ufc)
```

## 10.3 Diagnostics

Examinons d'abord les diagnostics du modèle (cf.figure 10.1).

```
> opar <- par(mfrow = c(2, 2), mar = c(4, 4, 3, 1), las = 1)
> plot(hd.lm.1)
> par(opar)
```

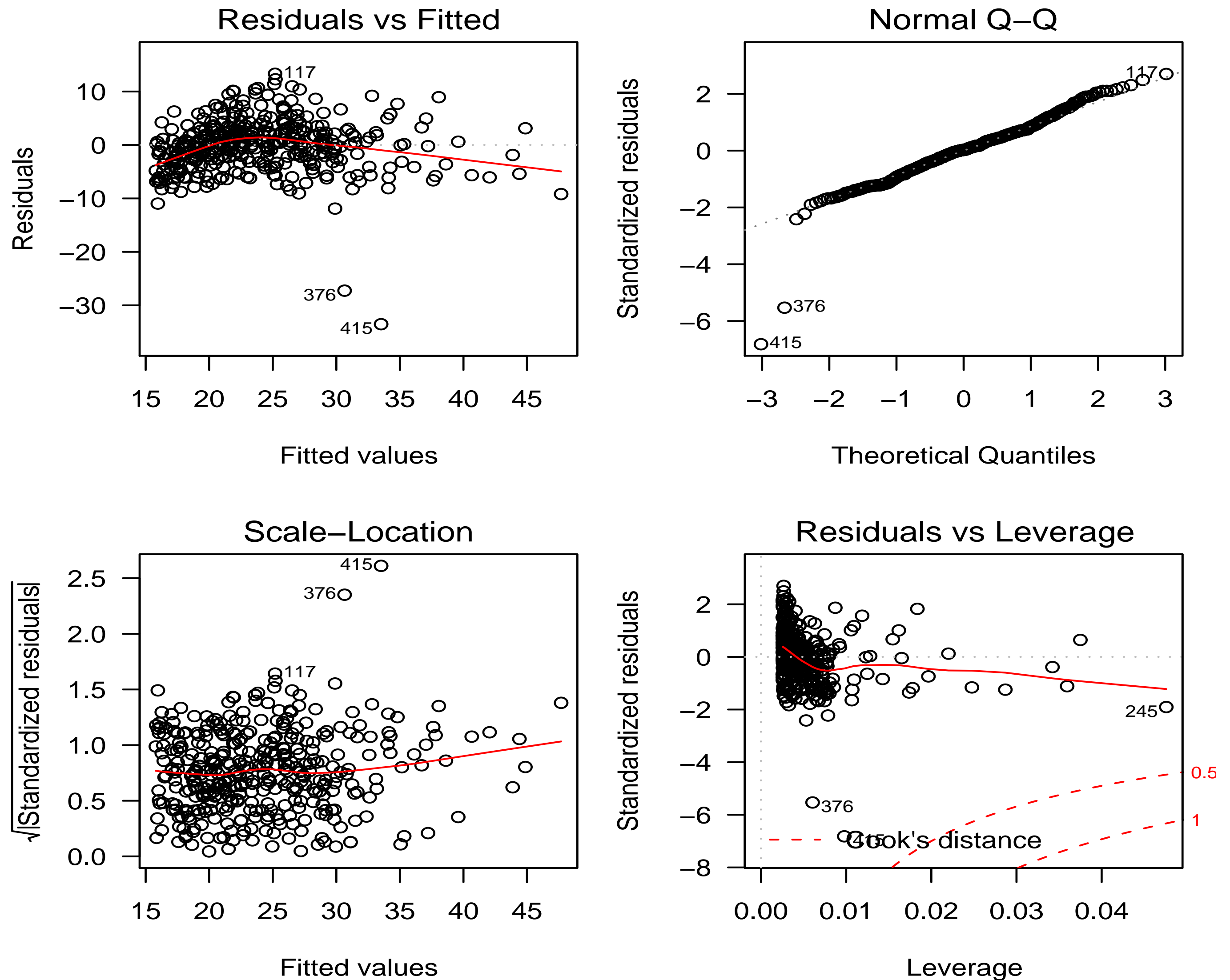


Figure 10.1 – Tracés diagnostiques de régression des hauteurs par les diamètres.

L'instruction de tracé par défaut produit quatre graphiques quand elle est appliquée à un modèle linéaire (`lm`).

- Le panneau en haut à gauche montre un tracé des résidus en fonction des valeurs ajustées, avec une courbe de lissage rouge superposée. Ici nous cherchons à mettre en évidence une courbure, une variance non constante (hétéroscédasticité) et des valeurs aberrantes. Notre tracé montre la courbure qui nous concerne, pas l'évidence d'une hétéroscédasticité et deux points qui ne ressemblent pas aux autres.
- Le panneau en haut à droite montre un tracé quantile des résidus standardisés opposés à une distribution normale. Ici, le tracé idéal est une ligne droite bien que de modestes écarts de rectitude soient usuellement acceptables (dus à la théorie des grands échantillons, cf. section 7.1) et les points aberrants évidents. Des écarts à la rectitude dans ce tracé peuvent indiquer la non nrmalité des résidus *ou* une variance non constante.
- Le panneau en bas à gauche montre la racine carrée des résidus absolus en fonction des valeurs ajustées, à côté d'une ligne droite rouge. Des écarts à l'horizontale signifient l'hétéroscédasticité.
- Le panneau en haut à droite montre un tracé des observations influentes (bras de levier) vis-à-vis des résidus standardisés. Ce sont les deux composantes de la distance de Cook, statistique qui rend compte de l'impact général sur l'estimation des paramètres des observations (noter qu'un grand résidu ou un point de haute influence seul n'est pas une garantie d'impact substantiel sur l'estimation des paramètres).

Une règle d'or, généralement bien acceptée, est qu'une distance de Cook supérieure à 1 doit attirer notre attention.

Les contours de ces distances (isoCook ?) à 0.5 et 1.0 sont ajoutés au graphique pour assister l'interprétation. Il y a certains points préoccupants, correspondant à des observations exceptionnelles que nous avions noté plus tôt. Toutefois, le diagnostic implique qu'elles ne devraient pas changer beaucoup les choses. Aucun de ces points ne sont dans la zone de danger des distances de Cook, par exemple.

Voyons comment nous pouvons les examiner. La méthode la plus simple est probablement d'utiliser les résidus pour localiser les délinquants. C'est un peu délicat – pour être en mesure de faire correspondre les résidus aux observations, il nous faut d'abord les ordonner selon l'ampleur du résidu, puis prendre les deux premières valeurs. Les crochets permettent d'accéder à l'extraction, qui est un des moteurs de la commodité de **R**.

```
> ufc[order(abs(residuals(hd.lm.1)), decreasing = TRUE), ][1:2, ]

    plot tree species dbh height diam.cm haut.m especes
415   78    5      WP 667      0    66.7    0.0      WP
376   67    6      WL 575     34    57.5    3.4      WL
```

Il faut bien constater que c'est une curieuse présentation des arbres ! Si on veut les exclure du modèle, on peut le faire via :

```
> hd.res.1 <- abs(residuals(hd.lm.1))
> hd.lm.1a <- lm(haut.m ~ diam.cm, data = ufc, subset = (hd.res.1 <
+     hd.res.1[order(hd.res.1, decreasing = TRUE)][2]))
```

Comment les estimations changent-elles ? Voici une beau petit tracé qui montre que les estimations ont très peu changé. Nous ne pouvons le faire que parce que tout est à la même échelle.

```
> opar <- par(las = 1)
> plot(coef(hd.lm.1), coef(hd.lm.1a), xlab = "Paramètres (toutes les données)",
+     ylab = "Paramètres (sans valeurs aberrantes)")
> text(coef(hd.lm.1)[1] - 2, coef(hd.lm.1a)[1] - 0.5, expression(hat(beta)[0]),
+     cex = 2, col = "blue")
> text(coef(hd.lm.1)[2] + 2, coef(hd.lm.1a)[2] + 0.5, expression(hat(beta)[1]),
+     cex = 2, col = "blue")
> points(summary(hd.lm.1)$sigma, summary(hd.lm.1a)$sigma)
> text(summary(hd.lm.1)$sigma + 1, summary(hd.lm.1a)$sigma, expression(hat(sigma)[epsilon]),
+     cex = 2, col = "darkgreen")
> abline(0, 1, col = "darkgrey")
> par(opar)
```

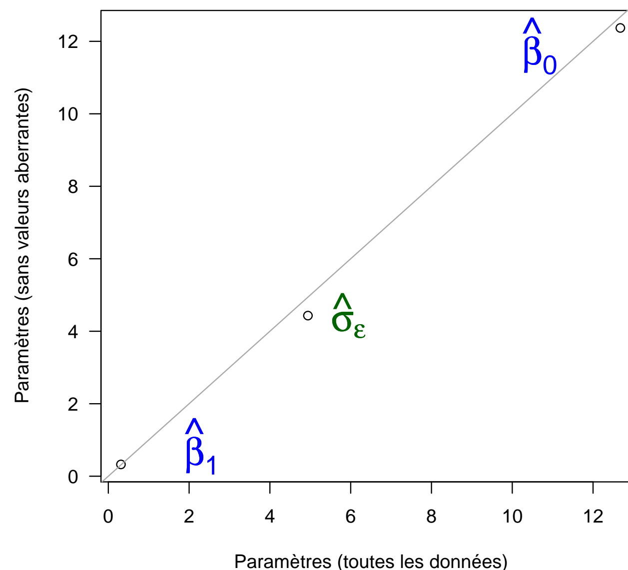


Figure 10.2 – Changement de l'estimation de paramètre après avoir ôté les points aberrants.

## 10.4 Autres outils

D'autres outils pour examiner les modèles de régression doivent être mentionnés :

```
> ?influence.measures
```

## 10.5 Examiner le modèle

Il est clair qu'une observation particulière n'a pas une très forte incidence sur le modèle, l'ordonnée à l'origine et la pente restent en grande partie inchangées, et la variance des résidus diminue légèrement. Jetons un coup d'oeil à l'exploration du modèle.

```
> summary(hd.lm.1)

Call:
lm(formula = haut.m ~ diam.cm, data = ufc)

Residuals:
     Min       1Q   Median       3Q      Max
-33.5257  -2.8619   0.1320   2.8512  13.3206

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 12.67570    0.56406   22.47   <2e-16 ***
diam.cm      0.31259    0.01388   22.52   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 " " 1

Residual standard error: 4.941 on 389 degrees of freedom
Multiple R-squared: 0.566,       Adjusted R-squared: 0.5649
F-statistic: 507.4 on 1 and 389 DF,  p-value: < 2.2e-16
```

Nous constatons que déjà, il ya une grande différence entre la variabilité marginale, qui est 7.49 m, et la variabilité conditionnelle, qui est 4.94 m. Le modèle semble être utile.

## 10.6 Autres angles

Le processus de d'interrogation du modèle est simplifié si nous nous rendons compte que l'ajustement du modèle est un objet, et son exploration un objet différent.

```
> names(hd.lm.1)

 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

> names(summary(hd.lm.1))

 [1] "call"          "terms"         "residuals"     "coefficients"
 [5] "aliased"       "sigma"         "df"            "r.squared"
 [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

Des fonctions de haut niveau, nommées `methods`, existent pour permettre une extraction fiable de l'information du modèle, par exemple, `residuals()` et `fitted()`. Nous apprendrons quelle méthodes sont disponibles pour nos objets comme suit :

```
> class(hd.lm.1)

[1] "lm"
```

```
> methods(class = class(hd.lm.1))

 [1] add1.lm*           alias.lm*          anova.lm           case.names.lm*
 [5] confint.lm*        cooks.distance.lm* deviance.lm*       dfbeta.lm*
 [9] dfbetas.lm*        drop1.lm*          dummy.coef.lm*     effects.lm*
[13] extractAIC.lm*     family.lm*         formula.lm*        hatvalues.lm
[17] influence.lm*      kappa.lm           labels.lm*         logLik.lm*
[21] model.frame.lm     model.matrix.lm    plot.lm            predict.lm
[25] print.lm           proj.lm*           residuals.lm       rstandard.lm
[29] rstudent.lm        simulate.lm*       summary.lm         variable.names.lm*
[33] vcov.lm*

   Non-visible functions are asterisked
```

Nous pouvons aussi extraire ou manipuler autrement les attributs des objets au moyen du signe $ :

```
> hd.lm.1$call

lm(formula = haut.m ~ diam.cm, data = ufc)

> summary(hd.lm.1)$sigma

[1] 4.941222
```

## 10.7 Autres modèles

Nous pouvons ajouter de la complexité à nos modèles comme suit. Essayez ceci et voyez ce que vous apprendrez :

```
> hd.lm.2 <- lm(haut.m ~ diam.cm + species, data = ufc)
> hd.lm.3 <- lm(haut.m ~ diam.cm * species, data = ufc)
```

## 10.8 Pondération

Bien que l'échantillon des arbres puisse être considéré comme un échantillon aléatoire, ils ne sont pas choisis avec une probabilité égale, et ils sont regroupés en un endroit, parce qu'ils sont de rayon variable dans un échantillon de parcelles. Quelle différence d'estimations de paramètre représente un tel plan d'échantillonnage ? Ici, nous allons évaluer l'effet de la pondération d'échantillon. Nous examinerons les outils qui éclairent l'effet du regroupement dans le prochain chapitre.

Rappelons que dans une parcelle à rayon variable, la probabilité de sélection d'un arbre est proportionnelle à sa surface de base. Nous pouvons ajuster un modèle qui s'accommode cet effet comme suit :

```
> hd.lm.5 <- lm(haut.m ~ diam.cm * especes,
+     weights = diam.cm^-2,
+     data = ufc)
```

Maintenant, nous voudrions connaître l'effet de ce changement sur nos estimations de paramètres. Pour retirer les estimations de chaque pente et d'ordonnée à l'origine nous pouvons utiliser la fonction `estimable()` fournie dans le paquetage `gmodels`, mais pour certains modèles, il est un moyen plus simple de les obtenir.

```
> unweighted <- coef(lm(haut.m ~ diam.cm * species - 1 - diam.cm,
+     data = ufc))
> weighted <- coef(lm(haut.m ~ diam.cm * species - 1 - diam.cm,
+     weights = diam.cm^-2, data = ufc))
```

Nous pouvons maintenant tracer ces estimations par divers moyens plus ou moins informatifs. Le code suivant construit la figure 10.3

```
> par(mfrow = c(1, 2), las = 1, mar = c(4, 4, 3, 1))
> plot(unweighted[1:10], weighted[1:10], main = "Ordonnée à l'origine",
+     type = "n", xlab = "Non pondéré", ylab = "Pondéré")
> abline(0, 1, col = "darkgrey")
> text(unweighted[1:10], weighted[1:10], levels(ufc$especes))
> plot(unweighted[11:20], weighted[11:20], main = "Pente", type = "n",
+     xlab = "Non pondéré", ylab = "Pondéré")
> abline(0, 1, col = "darkgrey")
> text(unweighted[11:20], weighted[11:20], levels(ufc$especes))
```

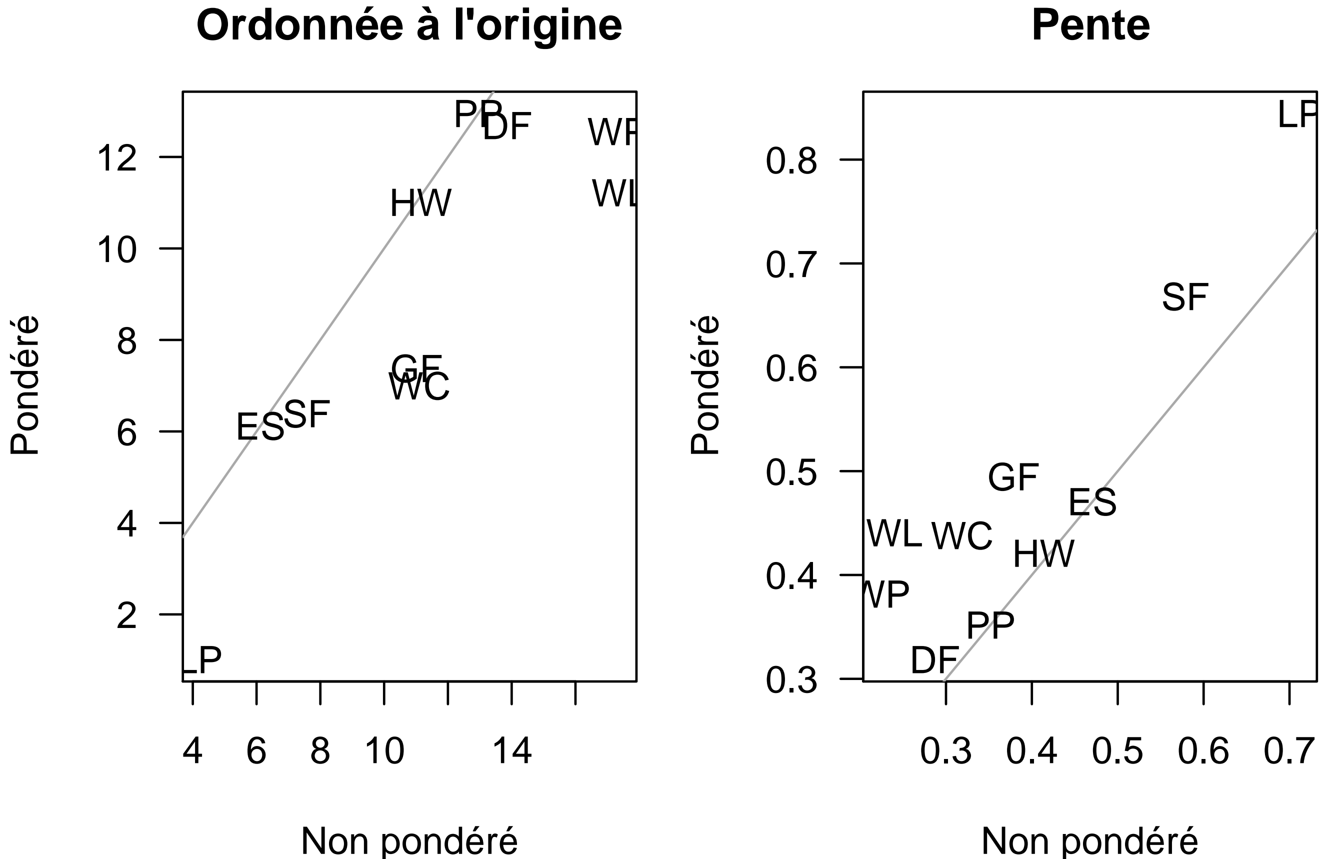


Figure 10.3 – Exploration de l'effet de pondération d'échantillon sur les estimations hauteur-diamètre par espèce.

L'effet de pondération semble être assez important. Cet effet peut être interprété comme nous dire que les petits et les grands arbres ont besoin de modèles différents.

## 10.9 Transformations

Parfois, le modélisateur a des raisons biologiques ou statistiques de changer la nature des données qui vont être modélisées. Ces transformations peuvent être fait soit par la création d'une nouvelle variable dans le data-frame avec la fonction appropriée, soit en faisant la transformation dans l'appel au modèle.

```
> ufc$log.haut.m <- log(ufc$haut.m)
> hd.lm.4a <- lm(log.haut.m ~ diam.cm * especes, data = ufc, subset = haut.m >
+     0)
> hd.lm.4b <- lm(I(log(haut.m)) ~ diam.cm * especes, data = ufc,
+     subset = haut.m > 0)
```

Notez l'utilisation de la fonction `I()` utilisée dans cette dernière approche. Cela dit à **R** d'interpréter la fonction de la manière habituelle, pas dans le cadre du modèle linéaire. La différence devient plus évidente quand on pense au double-usage que **R** fait de nos opérateurs familliers. Par exemple, * signifie habituellement la multiplication, mais

dans le modèle linéaire cela signifie les effets principaux et l'interaction. De même, + signifie généralement plus, mais dans le modèle linéaire, il est utilisé pour ponctuer une déclaration du modèle.

Dans mon expérience, la transformation est un outil sur-utilisé. Je crois qu'il vaut mieux si possible travailler dans les unités qui seront les plus logiques à la personne qui utilise le modèle. Bien sûr, il est possible de revenir en arrière sur la transformation, mais pourquoi le faire si elle est inutile ?

Il existe certainement des cas où la transformation est nécessaire, mais très souvent une stratégie plus appropriée et satisfaisante est disponible, qu'il s'agisse d'un ajustement au modèle linéaire généralisé, un modèle additif, ou d'effectuer une petite expérience de Monte-Carlo sur les résidus en faisant confiance au Théorème Central Limite.

### 10.9.1 Une étude de cas

Nous avons besoin d'un exemple assez difficile. Les données suivantes sont un assez bon exemple à essayer, pour faire une pause sylvestre. Ces données sont tirées de OzDASL :

http://www.statsci.org/data/oz/rugby.html

La description qui suit est de Lee (1994). Le rugby est un sport populaire quasi-amateur largement joué aux États-Unis, au Royaume Uni, en France, Australie, Nouvelle-Zélande, Afrique du Sud et ailleurs. Il a rapidement gagne en popularité aux États-Unis, au Canada, au Japon et dans certaines parties de l'Europe. Récemment, certaines des règles du jeu ont été changées, dans le but de rendre le jeu plus excitant. Dans une étude visant à examiner les effets des modifications des règles, Hollings et Triggs (1993)[11] ont recueilli des données sur certains jeux récents.

En règle générale, un jeu consiste en rafales d'activité qui se terminent lorsque des points sont marqués, si la balle est déplacé hors du champ de jeu ou si une violation des règles se produit. En 1992, les enquêteurs ont recueilli des données sur dix matches internationaux face à l'équipe nationale de Nouvelle-Zélande,les « All Blacks ». Les cinq premiers jeux étudiés ont été les dernières parties internationales jouées selon les anciennes règles, et la deuxième série de cinq ont été les premiers matches internationaux joués selon les nouvelles règles.

Pour chacune des dix parties, les données listent les temps successifs de jeu (en secondes) dans le match. On s'intéresse à voir si les passages étaient, en moyenne, plus ou moins grands avec les nouvelles règles. (L'intention lorsque les règles été modifiées était presque certainement de rendre le jeu plus continu.)

```
> require(lattice)
> require(MASS)
> require(boot)
> require(nlme)
```

On lit les données et fait les adaptations nécessaires.

```
> rugby <- read.csv("../data/rugby.csv")
> rugby$rules <- "New"
> rugby$rules[rugby$game < 5.5] <- "Old"
> rugby$game.no <- as.numeric(rugby$game)
> rugby$rules <- factor(rugby$rules)
> rugby$game <- factor(rugby$game)
> str(rugby)

'data.frame': 979 obs. of  4 variables:
 $ game   : Factor w/ 10 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ time   : num  39.2 2.7 9.2 14.6 1.9 17.8 15.5 53.8 17.5 27.5 ...
 $ rules  : Factor w/ 2 levels "New","Old": 2 2 2 2 2 2 2 2 2 2 ...
 $ game.no: num  1 1 1 1 1 1 1 1 1 1 ...
```

L'analyse suivante est tirée du site web OzDASL :

> On peut utiliser une simple analyse de variance sur ces données pour comparer les dix jeux.
> Un contraste serait alors pertinent pour comparer les cinq premiers jeux aux cinq deuxièmes.
>
> Autrement, on pourrait regrouper les temps pour les cinq premiers jeux et les cinq derniers jeux pris ensemble ce qui conduirait à deux échantillons de test.
>
> Les temps sont très asymétriquement distribués à droite. Ils ont besoin d'être transformé en une symétrie approximative ; ou encore, ils pourraient être considérés comme une distribution approximativement exponentielle ou gamma.

La figure 10.4 confirme que les données sont assez asymétriquement distribuées.

```
> densityplot(~time | game, data = rugby, layout = c(5, 2))
```

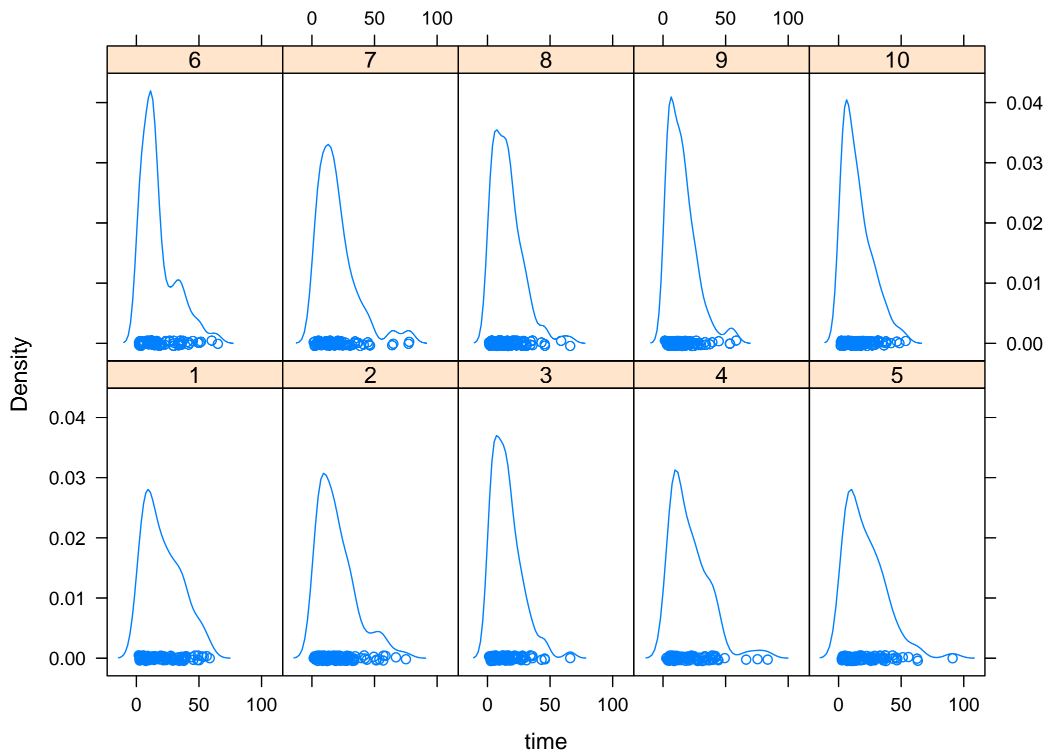


Figure 10.4 – Tracé lattice des densités du temps de jeu, par jeu, au rugby.

En supposant que les choix de conception faits par l'analyse recommandée soient raisonnables, avons-nous vraiment besoin de transformer les données pour l'analyse ? C'est une démarche standard. La figure 10.7 présente un important diagnostic ; le tracé quantile-quantile des résidus standard.

```
> times.lm <- lm(time ~ rules, data = rugby)
> qqnorm(stdres(times.lm))
> qqline(stdres(times.lm))
> boxcox(times.lm)
```

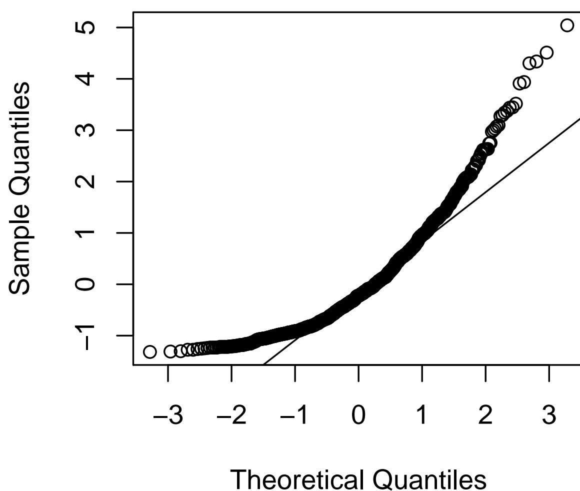


Figure 10.5 – Graphique diagnostique-clef de régression aux moindres carrés ordinaires des données de rugby : le tracé quantile-quantile des résidus standard.

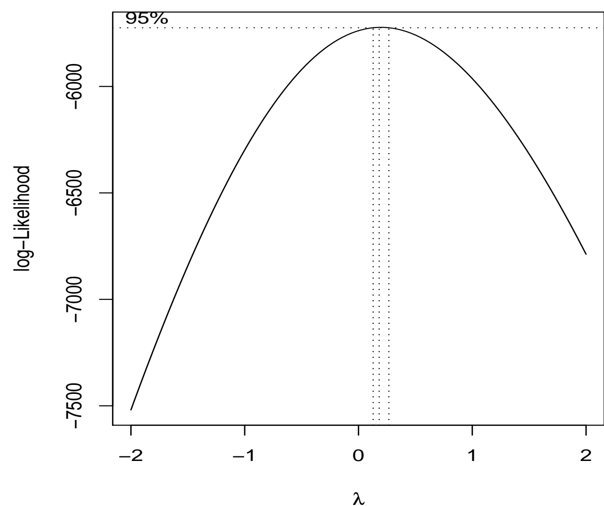


Figure 10.6 – Test diagnostique Box-Cox du paquetage MASS. Noter l'intervalle de confiance étroit, qui recommande la transformation.

Si nous devions suivre la stratégie préconisée par le site (et un grand nombre de livres de régression !), nous serions en train d'essayer quelque chose comme un diagnostic Box-Cox, qui nous donne la figure 10.6. Il semble nous recommander une transformation d'environ 0.2.

Effectuer la transformation et exécuter l'analyse nous donne un tracé de probabilité normal des résidus présenté en figure 10.7. Pas parfait, mais beaucoup mieux qu'en figure 10.7.

```
> times.lm.2 <- lm(I(time^0.2) ~ rules, data = rugby)
> qqnorm(stdres(times.lm.2))
> qqline(stdres(times.lm.2))
```

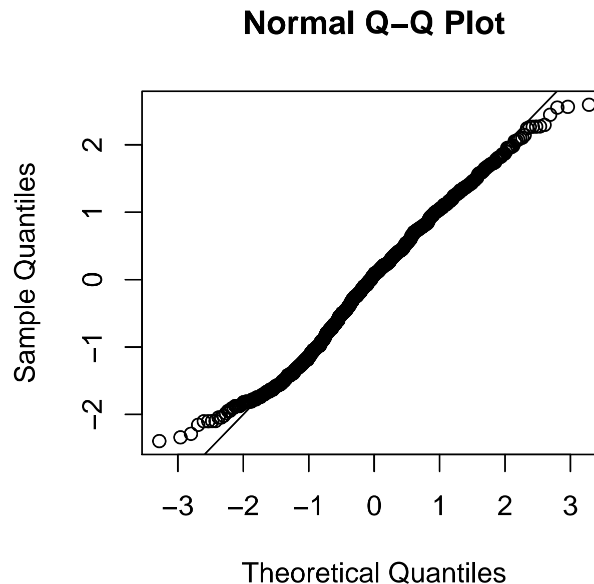


Figure 10.7 – Graphique diagnostique-clef de régression aux moindres carrés sur les données de rugby transformées : tracé quantile-quantile des résidus standard.

L'hypothèse nulle qu'il n'y a pas de différence entre les anciennes et nouvelles règles peut maintenant être testée.

```
> anova(times.lm.2)

Analysis of Variance Table

Response: I(time^0.2)
           Df Sum Sq Mean Sq F value    Pr(>F)
rules       1  1.147   1.147  14.301 0.0001652 ***
Residuals 977 78.335   0.080
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 " " 1
```

Un effet `rules` très significatif suggère qu'il y a bien une différence, pour les données transformées. En testant le modèle avec des données non transformées nous aurions conclu la même chose.

Les livres sur la régression recommandent communément un examen du tracé de probabilité normale des résidus, afin d'évaluer le clivage modèle/données avec l'hypothèse de normalité. Si les résidus sont normaux, alors on peut estimer les intervalles de confiance et les paramètres de test, commodément, et tester le modèle complet, etc.

En toute justice, je dois ajouter que tous ces livres ne le font pas, par exemple Weisberg (2005) [26] et Gelman et Hill (2007) [10] donnent à ce diagnostic une faible priorité.

Huber (1981) [12] dit (p. 159-160) que la distribution d'échantillonnage d'une combinaison linéaire arbitraire de paramètres de régression estimés aux moindres carrés est asymptotiquement normale si $h = \max_i(h_i) \to 0$ quand $n$ augmente, où les $h_i$ sont des éléments diagonaux de la matrice $\mathbf{H}$ (c'est-à-dire les leviers). Par conséquent, la condition précitée de la normalité des résidus est vraiment trop restrictive. C'est-à-dire, nous devrions être en mesure de s'acquitter de ces estimations et de tester que dans ces circonstances l'une de ces deux cas se produit :

1. Les résidus sont normalement distribués ou
2. la taille de l'échantillon est suffisamment grande pour faire appel à la Loi des Grands Nombres.

Le tracé qq normal diagnostic de régression vérifie le cas 1, mais pas le cas 2. Par conséquent, la tracé de diagnostic pourrait signifier qu'il faut transformer ou utiliser une pondération ou s'inquiéter d'une certaine façon, si les résidus ne sont pas issus d'une loi normale. Toutefois, dans certaines situations, nous n'avez pas à vous inquiéter car il est raisonnable d'interpréter le Théorème de la limite centrale en affirmant que, pour une assez grande taille de l'échantillon, la distribution d'échantillonnage d'un estimateur peut être considérée comme étant normalement distribuée avec erreur négligeable. Le problème consiste à déterminer ce qui est une taille suffisante pour l'échantillon ou plus immédiatement, si l'échantillon dont nous disposons est suffisamment grand.

Je souhaiterais éviter de transformer car il s'agit de faire un gâchis avec ce qui est souvent en fait un bon modèle abstrait. Par exemple, j'aimerais vraiment ne pas transformer les données de rugby, parce que le modèle y perd ses unités de mesure, et, par conséquent, une tranche substantielle de sens du monde réel : l'interprétation et l'utilité. De ce point de vue, le test diagnostic de Box-Cox, comme sur la figure 10.6, nous donne l'exacte inverse asymptotique d'un bon conseil. Si la taille de l'échantillon augmente, l'IC à 95% du paramètre qui représente la meilleure transformation, se resserre, ce qui semble rendre la transformation plus urgente - quand elle est en fait moins urgente selon de la théorie des grands échantillons.

Pour éviter ce pénible problème, nous pourrions lancer un bootstrapping des données, ajuster le modèle, obtenir des estimations non-paramétriques des distributions d'échantillonnage du(des) paramètre(s) d'intérêt (qu'elles soient des estimations ou des statistiques de test), et alors soit évaluer leur normalité, soit simplement utiliser les distributions estimées directement. Une telle approche nous permettra de équilibrer les effets de la distribution sous-jacente des paramètres avec la théorie des grands échantillons, pour nous faire savoir si nous avons réellement besoin de transformer, appliquer des poids, ou quoi que ce soit.

Ainsi, nous pouvons écrire une fonction qui permet de calculer les estimations de paramètres d'un échantillon arbitraire. Pour ce modèle, ce sera tout simplement la différence entre les moyennes de temps pour les anciennes et les nouvelles règles, qui ne nécessite pas un modèle linéaire d'ajustement formalisé, mais nous garderons le modèle de toute façon. Nous utilisons un bootstrap studentisé. Nous verrons comment attaquer ce problème avec une boucle à la section 7.2.

```
> boot.lm <- function(data, i) {
+     model.t <- coefficients(summary(lm(data[i, 2] ~ data[i, 3])))
+     c(model.t[2, 1], model.t[2.2]^2)
+ }
```

Cette fonction extrait l'estimation de la différence de temps entre les deux groupes et sa variance.

Le commande de bootstrap est :

```
> boot.rugby <- boot(rugby, boot.lm, R = 199)
```

On peut extraire et vérifier la distribution des simulations bootstrappée via le code suivant dont la sortie est représentée en figure 10.8 :

```
> plot(boot.rugby)
```

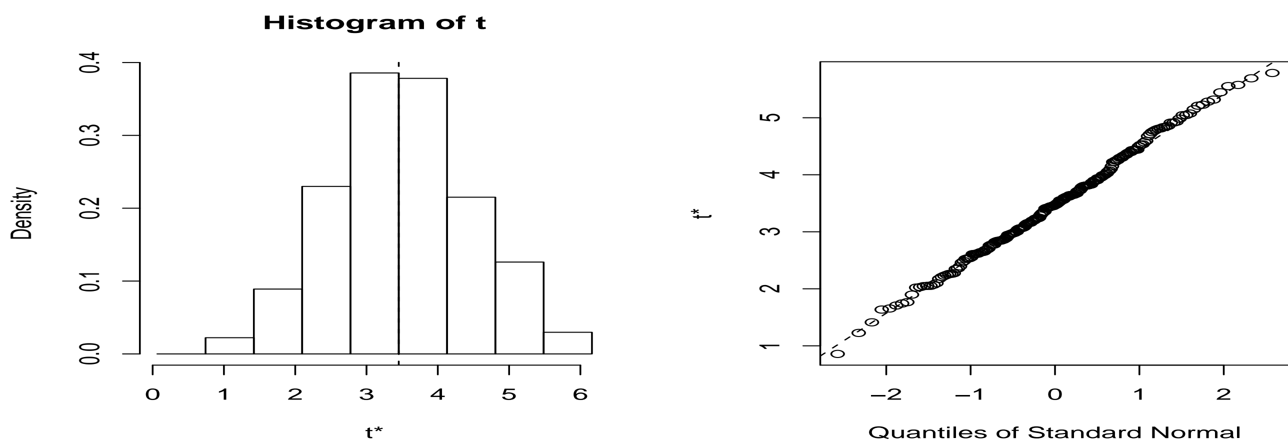


Figure 10.8 – Graphique diagnostique-clef sur des données rugby non transformées : histogramme et tracé quantile-quantile des valeurs simulées.

Cette distribution semble être acceptablement normale. Ce diagnostic suggère que nous pouvons procéder à l'analyse des résultats bootstrappés (Davison et Hinkley, 1997) [6]. Pour conclure, nous pouvons comparer l'estimation bootstrap de l'erreur standard

```
> sd(boot.rugby$t[, 1])

[1] 0.8599485
```

avec l'estimation originale basée sur le modèle des données originales non transformées :

```
> coefficients(summary(times.lm))[2, 2]

[1] 0.9038698
```

Elles apparaissent aussi relativement proche. Ce résultat suggère, dans ce cas, qu'un appel à la théorie des grands-échantillon pour le modèle initial aurait été raisonnable. La transformation semble avoir été inutile, et nous a coûté un modèle facilement interprétable. Il serait aussi intéressant de savoir si la transformation est coûteuse en puissance et en taille.

## 10.10 Tester des effets spécifiques

**R** utilise un algorithme pour contrôler l'ordre dans lequel les termes entrent dans un modèle. L'algorithme respecte la hiérarchie, toutes les interactions entrent dans le modèle après les effets principaux, etc. Toutefois, il nous faut parfois modifier l'ordre dans lequel les termes entrent dans le modèle. Ce peut être parce que nous voulons tester certains termes avec d'autres termes déjà dans le modèle. Par exemple, en utilisant les données NPK fournies par `MASS` (Venables et Ripley, 2002 [24]), imaginez que N et P sont des variables du plan d'expérience, et que nous tenions à tester l'effet de K dans un modèle qui contient déjà l'interaction N:P.

```
> require(MASS)
> data(npk)
> anova(lm(yield ~ block + N * P + K, npk))

Analysis of Variance Table

Response: yield
          Df Sum Sq Mean Sq F value   Pr(>F)
block      5 343.29   68.66  4.3911 0.012954 *
N          1 189.28  189.28 12.1055 0.003684 **
P          1   8.40    8.40  0.5373 0.475637
K          1  95.20   95.20  6.0886 0.027114 *
N:P        1  21.28   21.28  1.3611 0.262841
Residuals 14 218.90   15.64
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 " " 1
```

Le code précédent échoue à cause de l'effet de l'algorithme d'ordre dans **R**. Pour le contrer, on doit utiliser l'option de respect des termes saisis :

```
> anova(lm(terms(yield ~ block + N * P + K, keep.order = TRUE),
+     npk))

Analysis of Variance Table

Response: yield
          Df Sum Sq Mean Sq F value   Pr(>F)
block      5 343.29   68.66  4.3911 0.012954 *
N          1 189.28  189.28 12.1055 0.003684 **
P          1   8.40    8.40  0.5373 0.475637
N:P        1  21.28   21.28  1.3611 0.262841
K          1  95.20   95.20  6.0886 0.027114 *
Residuals 14 218.90   15.64
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 " " 1
```

Heureusement, nous apprenons que le terme d'ordre ne fait pas de différence dans ce cas.

## 10.11 Contributions

Les facteurs d'inflation de variance sont disponibles dans le paquetage `car` du Professeur John Fox.

## 10.12 Autres moyens d'ajustement

Nous pouvons utiliser d'autres outils pour ajuster la régression. Par exemple, le code ci-dessous ajuste le même modèle de régression utilisant le maximum de vraisemblance, et fournit des estimations des paramètres et de leurs erreurs-standard asymptotique.

```
> normal.ll <- function(parameters, x, y)
+     sum(dnorm(y, parameters[1] + parameters[2] * x, parameters[3],
+         log = T))
+
> good.fit <- optim(c(1, 1, 1), normal.ll, hessian = TRUE, control = list(fnscale = -1),
+     x = ufc$diam.cm, y = ufc$haut.m)
> good.fit$par
```

```
[1] 12.6781358  0.3127113  4.9327072
```

```
> sqrt(diag(solve(-good.fit$hessian)))
```

```
[1] 0.56309282 0.01385363 0.17661565
```

On peut aussi ajuster une distribution d'erreur différente en ajustant une famille de l'endroit. On se souvient que pour toute distribution de probabilité $f(x)$ et des constantes $\mu$ et $\sigma > 0$,

$$g(x|\mu, \sigma) = \frac{1}{\sigma} f\left(\frac{x - \mu}{\sigma}\right) \tag{10.1}$$

est aussi une distribution de probabilité (cf., par exemple, Casella et Berger 1990, [3] p. 116). Ici nous utiliserons $\exp(\sigma)$ plutôt que $\sigma$ parce que $\exp(\sigma) > 0$.

Nous ajustons ici une régression linéaire avec des erreurs décrites par une distribution de Student à 3 degrés de liberté, qui donne de la robustesse vis-à-vis des valeurs aberrantes.

```
> t3.ll <- function(parameters, x, y)
+     sum(dt((y - parameters[1] - x * parameters[2])/exp(parameters[3]),
+         df = 3, log = T) - parameters[3])
+
> good.fit.t <- optim(c(1, 1, 1), t3.ll, hessian = TRUE, control = list(fnscale = -1),
+     x = ufc$diam.cm, y = ufc$haut.m)
> good.fit.t$par
```

```
[1] 12.1153235  0.3310526  1.2474535
```

```
> sqrt(diag(solve(-good.fit.t$hessian)))
```

```
[1] 0.52052602 0.01352119 0.04918994
```

L'interprétation des paramètres $\mu$ et $\sigma$ est laissée à l'analyste. N'oubliez pas que la variance de $t_\nu$ est $\frac{\nu}{\nu-2}$, aussi le paramètre d'échelle rapportée ne doit pas être interprété comme l'écart-type conditionnel des données.

Finalement, nous pouvons utiliser un modèle non linéaire en changeant la fonction moyenne.

```
> normal.ll.nl <- function(parameters, x, y)
+     sum(dnorm(y, parameters[1] * x^parameters[2], parameters[3],
+         log = T))
+
> good.fit <- optim(c(1, 1, 1), normal.ll.nl, hessian = TRUE, control = list(fnscale = -1),
+     x = ufc$diam.cm, y = ufc$haut.m)
> good.fit$par
```

```
[1] 4.4587482 0.4775113 4.7196199
```

Tous ces estimateurs de paramètre sont des estimateurs du maximum de vraisemblance, conditionnés au modèle, et sont donc des estimateurs asymptotiquement normaux et performants. J'ai inclus plus d'informations sur la rédaction de vos propres fonctions au chapitre 7.

# Chapitre 11

# Modèles hiérarchiques

Nous allons maintenant passer à l'analyse des données hiérarchiques en utilisant des modèles à effets mixtes. Ces modèles sont une correspondance naturelle pour de nombreux problèmes qui se produisent couramment dans les ressources naturelles.

## 11.1 Introduction

Rappelons que pour un ajustement de régression linéaire en utilisant les techniques ordinaires avec lesquelles vous pourriez être familier, vous deviez faire quelques hypothèses sur la nature des résidus. Plus précisément, il était nécessaire de supposer que les résidus étaient :

1. indépendants,
2. identiquement distribués et le plus souvent
3. distribués normalement.

L'hypothèse d'une variance constante (homoscédasticité) correspond à l'hypothèse de distribution identique (point 2, ci dessus).

Si ces hypothèses sont vérifiées, voire même seulement défendable, alors la vie est belle. Toutefois, le plus souvent, nous savons qu'elles ne sont pas vérifiées. Cela peut se produire pour les ressources naturelles dans les collectes de données parce que les données peuvent avoir une structure temporelle, spatiale ou hiérarchique voire même les trois [1]. Cette structure peut ou ne peut pas être pertinente pour votre question scientifique, mais il est très pertinente à l'analyse des données et à la modélisation !

Je parle de plusieurs références. Aucune sont obligatoirement à acheter ou lire, mais toutes seront utiles à un moment ou à un autre. Elles sont mentionnées en ordre décroissant *approximatif* d'utilité *à ce niveau.*

### 11.1.1 Méthodologique

Pinheiro et Bates (2000) [15] détaillent l'ajustement dans **R** et dans **Splus** qui, tous deux, fournissent un modèle de discrimination graphique et des outils de vérification de premier ordre à travers les bibliothèques écrites par les auteurs. De bons exemples sont fournis. Schabenberger et Pierce (2002) [20] est une mine d'or des conseils judicieux pour la compréhension et l'ajustement des modèles généralisés et à effets mixtes qui forment le noyau de cette classe. Il existe de nombreux exemples de travail et plein de code SAS. Vous êtes invités à interagir avec le matériel pour voir l'ajustement. Enfin, Fitzmaurice et al. (2004) [8] et Gelman et Hill (2007) [10] ont fait un travail de premier choix en expliquant la pratique et l'ajustement des modèles à effets mixtes.

En plus de ces livres, il existe de nombreux articles qui tentent d'expliquer divers éléments de ces sujets de manière plus ou moins détaillée. Dans le passé, j'ai trouvé Robinson (1991) (aucun rapport !) Et les discussions qui suivent particulièrement utiles.

1. « La première loi de l'écologie est que tout est relié à tout. » – Barry Commoner, biologiste/environnementaliste US. Le cercle infernal : Nature, Homme et Technologie. New York : Knopf, 1971.

### 11.1.2 Général

Venables et Ripley (2002) [24] est un *must-have* si vous êtes intéressé au travail avec **R** ou **Splus**. Les trois éditions précédentes sont désormais légendaire dans la communauté **R/S** pour leur explication approfondie de la pratique statistique moderne, avec de généreux exemples et leur code. La communauté **R** a également produit d'excellents documents de démarrage. Ceux-ci sont librement disponibles au téléchargement sur le site du projet **R** : `http://www.r-project.org`. Télécharger et lire tout ou partie d'entre eux, en écrivant le code au passage. Si vous êtes intéressé par une exploration plus profonde des possibilités de programmation en **R** ou **S** alors Venables et Ripley (2000) est très utile. Certains projets de grande envergure où j'ai été impliqué ont nécessité l'appel à **C** depuis **R** ; cette référence a été alors très utile.

## 11.2 De la théorie

Les modèles à effets mixtes comprennent à la fois des effets fixes et des effets aléatoires. La structure du modèle est habituellement proposée par le plan d'expérience sous jacent ou la structure des données. J'aime prétendre que les effets aléatoires sont proposés par la conception d'une étude, et que les effets fixes sont proposés par les hypothèses, mais ce n'est pas toujours vrai.

### 11.2.1 Les effets

Les « effets » sont des variables prédictives dans un modèle linéaire ou non linéaire [2]. La discussion sur les effets fixes et aléatoires peuvent sembler un peu de confuse. « Aléatoire » et « fixe » ne sont pas normalement censés être opposés l'un à l'autre, ou même mutuellement exclusifs (sauf par la simple force de l'habitude !). Pourquoi pas « stochastique » et « déterministe » ? Ou, « échantillon » et « population » ? Ou, « locale » et « mondial » ? Ces labels peuvent mieux raconter l'histoire. Gelman et Hill (2007) déclinent complètement l'utilisation d'aléatoire et fixe.

Il existe différentes façons de voir ces deux propriétés. Malheureusement, cela affecte l'analyse des données et les conclusions qui peuvent en être tirées. Modélisateurs peuvent être en désaccord sur le fait de savoir si les effets doivent être fixes ou aléatoires, et le même effet peut changer en fonction des circonstances. Certes, les statisticiens ne se sont pas entendus sur une stratégie. Certains prétendent que cela dépend entièrement de l'inférence, et certains que cela dépend entièrement de la conception de l'expérience.

Comme les outils statistiques qui sont utilisés pour analyser de telles données deviennent plus sophistiqués, et que des modèles précédemment impensable devenir dominant, les insuffisances de l'ancien vocabulaires sont de plus en plus évidentes.

**Effets aléatoires**
Les effets aléatoires sont ceux dont le niveau est supposé aléatoirement échantillonné dans un ensemble de niveaux possibles. Généralement, bien que pas toujours, quand des effets aléatoires sont considérés c'est dans l'intérêt de relier les résultats d'une population plus étendue. C'est-à-dire que les niveaux sont supposés être représentatifs collectivement d'une classe plus étendue de niveaux potentiels, pour laquelle nous souhaitons dire quelque chose. Autrement dit, on pourrait dire qu'un effet aléatoire est simplement un de ceux pour lesquels la localisation n'est pas le principal intérêt. Une autre alternative est qu'on pourrait dire qu'un effet aléatoire en est un que l'on souhaite marginaliser, pour quelque raison que ce soit.

**Effets fixes**
Les effets fixes sont généralement supposés être choisis dans un but précis et ne représentent rien d'autre qu'eux-mêmes. Si une expérience devait être répétée, avec le même niveau d'effet expérimental produit, alors l'effet est fixe. Toutefois, certains effets qui peuvent varier en reconduction de l'expérience peuvent aussi être fixe, aussi n'est-ce pas tranché. Autrement, on peut dire que un effet fixe est tout simplement celui pour lequel les estimations de localisation sont de principal intérêt. Une autre alternative est qu'on peut dire qu'un effet fixe en est un que vous souhaitez prendre comme condition, pour quelque raison que ce soit.

**Effets mixtes**
Certaines variables n'occupent pas d'elles-mêmes facilement une classification et, soit la connaissance du processus et/ou une orientation épistémologique est nécessaire. C'est commun pour les ressources naturelles. Par exemple, si une expérience qui, selon nous, est susceptibles d'être affectée par le climat est répétée sur un certain nombre d'années, l'*année* sera t'elle un effet fixe ou un effet aléatoire ? Ce n'est pas un échantillon aléatoire d'années possibles, mais les mêmes années ne se reproduiraient pas si l'expérience avait été répétée. De même pour la reproduction d'une expérience à des endroits connus : certains prétendent que ceux-ci devraient être un effet fixe, d'autres qu'elles représentent des variations environnementales, et, par conséquent, peuvent être considérés comme un effet aléatoire.

2. Le nom de ce label est suspendu au plan d'expérience et n'est plus réellement adapté à l'application, mais ainsi va l'inertie.

### 11.2.2 Construction du modèle

Le processus de construction de modèle devient bien plus complexe maintenant. Nous devons équilibrer différentes démarches et hypothèses, dont chacune comporte des implications différentes pour le modèle et son service. Si nous pensons au processus d'ajustement ordinaire d'une régression comme semblable à un diagramme, alors ajouter des effets aléatoires ajoute aussi une nouvelle dimension au diagramme. Par conséquent, il est très important de planifier la démarche avant de commencer.

Le nombre de stratégies potentielles est aussi varié que le nombre de modèles que nous pouvons ajuster. En voici un que nous allons relier à nos prochains exemples.

1. Choisir le jeu minimal d'effets fixes et aléatoires pour le modèle.
   (a) Choisir les effets fixes qui doivent être présents. Ces effets doivent être tels que, s'ils ne sont pas dans le modèle, celui-ci ne veut rien dire.
   (b) Choisir les effets aléatoires qui doivent être présents. Ces effets doivent être tels que, s'ils ne sont pas dans le modèle, celui-ci ne reflétera pas le plan d'expérience.

   C'est le modèle de base auquel les autres seront comparés
2. Ajuster ce modèle aux données avec des outils encore à examiner, et vérifier les diagnostics d'hypothèse. Itérer le processus d'amélioration des effets aléatoires, notamment :
   (a) une structure de variance hétéroscedastique (plusieurs possibilités)
   (b) une structure de corrélation (plusieurs possibilités)
   (c) d'autres effets aléatoires (e.g. pentes aléatoires).
3. Lorsque les diagnostics donnent à penser que l'ajustement est raisonnable, envisager d'ajouter des effets fixes. À chaque étape, réexaminer les diagnostics pour être sûr que les estimations que que vous utiliserez pour évaluer les effets fixes sont fondées sur une bonne adéquation entre les données, le modèle et les hypothèses.

Un autre niveau de complexité est qu'il est possible que les hypothèses ne correspondent pas à l'absence de certains effets fixes ou effets aléatoires. Dans ce cas, un certain nombre d'itérations est inévitable.

Il est important de garder à l'esprit que les rôles des effets fixes et des effets aléatoires sont distincts. Correction Les effets fixes *expliquent* la variation. Les effets aléatoires *organisent* la variation inexpliquée. À la fin de la journée, vous obtiendrez un modèle qui semble superficiellement pire qu'une régression linéaire simple, par les métriques et la qualité du modèle. Notre but est de trouver une combinaison modèle/hypothèse qui corresponde aux diagnostics que nous examinons. Ajouter des effets aléatoires ajoute de l'information et améliore le diagnostic et la compatibilité, mais n'explique pas plus de variations !

L'essentiel est que l'objectif de l'analyste soit de trouver le modèle le plus simple qui satisfasse les hypothèses de régression nécessaires et réponde aux questions d'intérêt. C'est tentant d'aller à la chasse à la structure la plus complexe d'effets aléatoires, qui peut fournir un plus grand maximum de vraisemblance, mais si un simple modèle satisfait aux hypothèses et répond au maximum de questions, alors la probabilité est plus un exercice mathématique – pas une statistique.

**Exemple**
Pour éclairer certaines de ces questions, considérez les données d'analyse du tronc Grand Fir.

Ces données sont tracées en figures 11.1 et 11.2.

```
> rm(list = ls())
> stage <- read.csv("../data/stage.csv")
> stage$Tree.ID <- factor(stage$Tree.ID)
> stage$Forest.ID <- factor(stage$Forest, labels = c("Kaniksu",
+      "Coeur d'Alene", "St. Joe", "Clearwater", "Nez Perce", "Clark Fork",
+      "Umatilla", "Wallowa", "Payette"))
> stage$HabType.ID <- factor(stage$HabType, labels = c("Ts/Pach",
+      "Ts/Op", "Th/Pach", "AG/Pach", "PA/Pach"))
> stage$dbhib.cm <- stage$Dbhib * 2.54
> stage$height.m <- stage$Height/3.2808399
```

Les codes HabType renvoient aux espèces d'arbres du point culminant, qui est celui des espèces les plus tolérantes à l'ombre qui peuvent croître sur le site, et à dominante invendable, respectivement. Ts indique *Thuja plicata* et *Tsuga heterophylla*, Th. indique le seul *Thuja plicata*, AG est l'*Abies grandis*, PA le *Picea engelmanii* et l'*Abies lasiocarpa*, Pach est le *Pachistima myrsinites* et Op est l'*Oplopanaz horridurn* qui sent mauvais. Le Grand sapin(fir) est considérée comme l'espèce culminante pour l'AG/Pach, l'une des principales espèces caduque pour TH/Pach et PA/Pach, et une espèce caduque mineure pour Ts/Pach et Ts/OP. Grosso modo, une communauté est caduque s'il existe des preuves que, au moins quelques-unes des espèces sont temporaires, culminante et si la collectivité est auto-régénérante Daubenmire (1952) [5].

```
> opar <- par(las = 1)
> plot(stage$dbhib.cm, stage$height.m, xlab = "Diam. (cm)", ylab = "Haut. (m)")
> par(opar)
```

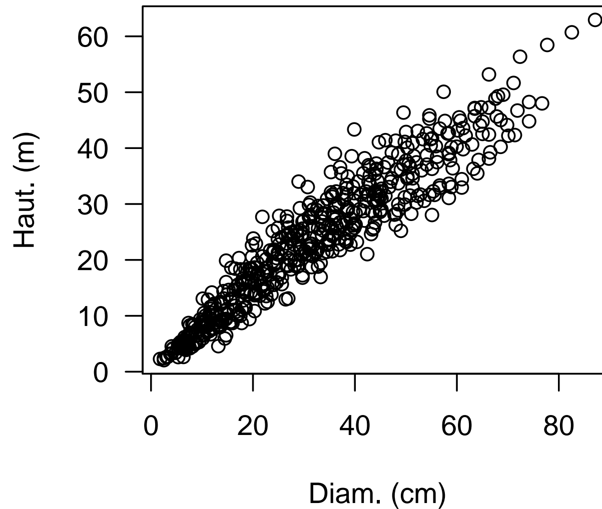


Figure 11.1 – Tracé de la hauteur en fonction du diamètre pour l'analyse des données de tronc à Grand Fir.

Il y avait des arbres dominants et co-dominants.

```
> colours <- c("deepskyblue", "goldenrod", "purple", "orangered2",
+     "seagreen")
> par(mfrow = c(3, 3), pty = "m", mar = c(2, 2, 3, 1) + 0.1, las = 1)
> for (i in 1:length(levels(stage$Forest.ID))) {
+     thisForest <- levels(stage$Forest.ID)[i]
+     forestData <- stage[stage$Forest.ID == thisForest, ]
+     plot(stage$dbhib.cm, stage$height.m, xlab = "", ylab = "",
+         main = thisForest, type = "n")
+     theseTrees <- factor(forestData$Tree.ID)
+     legend("topleft", unique(as.character(forestData$HabType.ID)),
+         xjust = 0, yjust = 1, bty = "n", col = colours[unique(forestData$HabType)],
+         lty = unique(forestData$HabType) + 1)
+     for (j in 1:length(levels(theseTrees))) {
+         thisTree <- levels(theseTrees)[j]
+         lines(forestData$dbhib.cm[forestData$Tree.ID == thisTree],
+             forestData$height.m[forestData$Tree.ID == thisTree],
+             col = colours[forestData$HabType[forestData$Tree.ID ==
+                 thisTree]], lty = forestData$HabType[forestData$Tree.ID ==
+                 thisTree] + 1)
+     }
+ }
> mtext("Height (m)", outer=T, side=2, line=2)
> mtext("Diameter (cm)", outer=T, side=1, line=2)
```

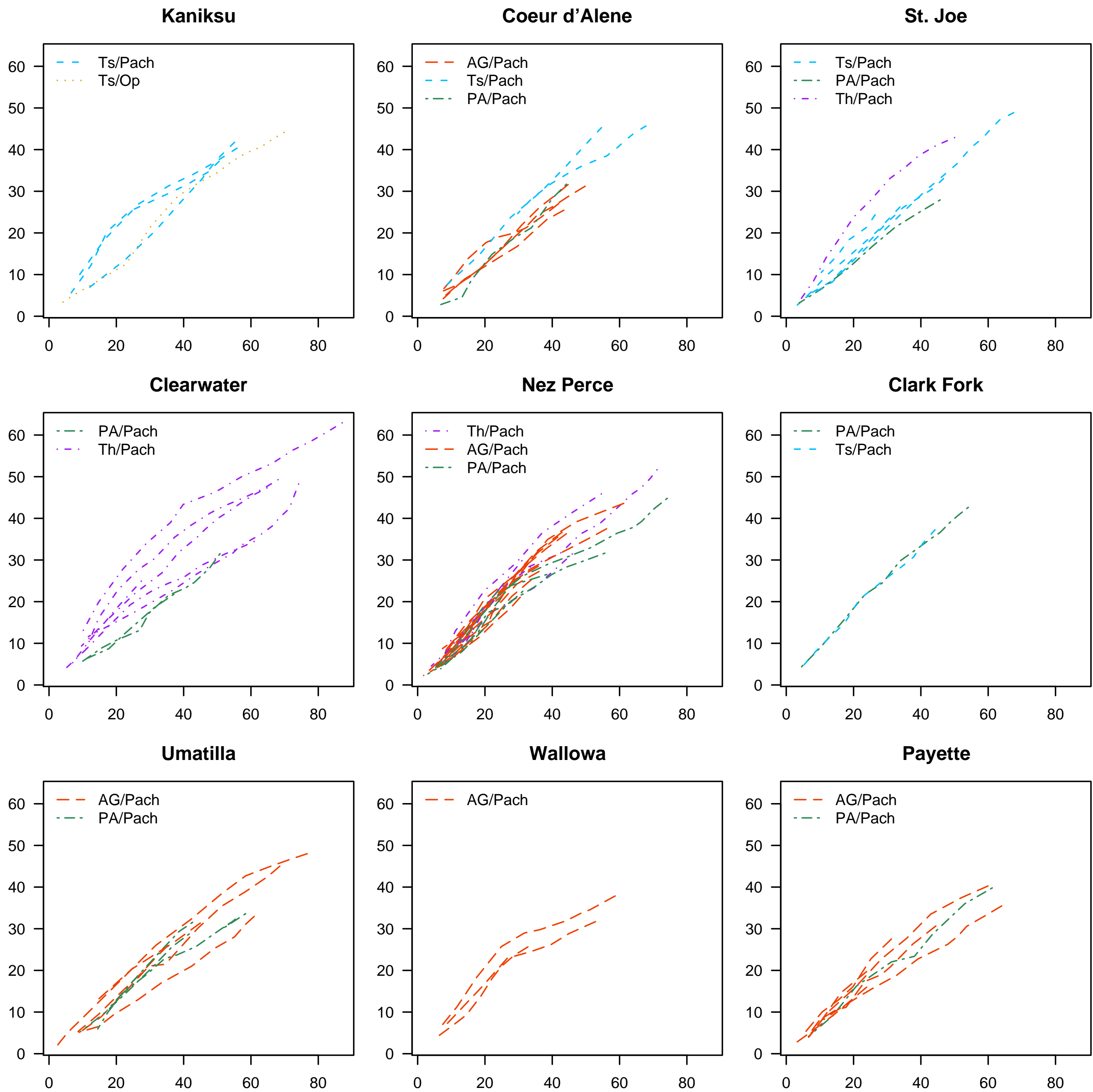


Figure 11.2 – Tracés par forêt nationale de la hauteur en fonction du diamètre pour l'analyse Grand Fir. Il y avait des arbres dominants et co-dominants

### 11.2.3 Dilemme

Un moyen facile d'aborder les avantages de la modélisation qui sont offerts par un modèle à d'effets mixtes est de penser à un exemple simple. Imaginons que nous soyons intéressés à la construction une relation hauteur-diamètre avec deux parcelles choisies au hasard dans une forêt, et que nous ayons mesuré trois arbres sur chacune d'elles. Il s'avère que les conditions de croissance sont tout à fait différentes selon les parcelles conduisant à une différence systématique sur la relation hauteur-diamètre (cf. figure 11.3). Le modèle est :

$$y_i = \beta_0 + \beta_1 \times x_i + \varepsilon_i \tag{11.1}$$

où $\beta_0$ et $\beta_1$ sont des paramètres fixes mais de population inconnue et $\varepsilon_i$ les résidus.

Les hypothèses suivantes sont requises :

- $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$
- Les $\varepsilon_i$ sont indépendants.

```
> trees <- data.frame(plot = factor(c(1, 1, 1, 2, 2, 2)), dbh.cm = c(30,
+     32, 35, 30, 33, 35), ht.m = c(25, 30, 40, 30, 40, 50))
> plot(trees$dbh.cm, trees$ht.m, pch = c(1, 19)[trees$plot], xlim = c(29,
+     36), xlab = "Diamètre (cm)", ylab = "Hauteur (m)")
> abline(lm(ht.m ~ dbh.cm, data = trees), col = "darkgrey")
```

Si nous ajustons une simple régression avec les arbres en entier nous obtenons un tracé des valeurs ajustées/résiduelles comme affiché en figure 11.4.

```
> case.model.1 <- lm(ht.m ~ dbh.cm, data = trees)
> plot(fitted(case.model.1), residuals(case.model.1), ylab = "Résidus",
+     xlab = "Valeurs ajustées", pch = c(1, 19)[trees$plot])
> abline(h = 0, col = "darkgrey")
```

Si nous ajustons une simple régression avec les arbres pour chaque parcelle nous obtenons un tracé des valeurs ajustées/résiduelles comme affiché en figure 11.5.

```
> case.model.2 <- lm(ht.m ~ dbh.cm * plot, data = trees)
> plot(fitted(case.model.2), residuals(case.model.2), xlab = "Valeurs ajustées",
+     ylab = "Résidus", pch = c(1, 19)[trees$plot])
> abline(h = 0, col = "darkgrey")
```

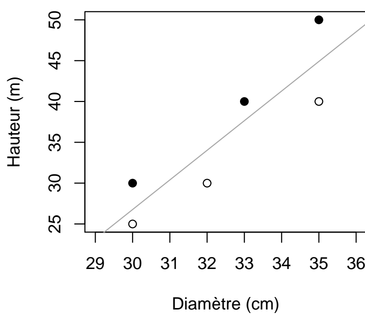


Figure 11.3 – Tracé de la hauteur-diamètre pour trois arbres sur deux parcelles avec la droite de régression.

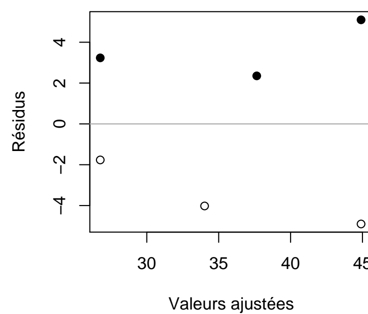


Figure 11.4 – Tracé des résidus du modèle hauteur-diamètre pour trois arbres sur deux parcelles.

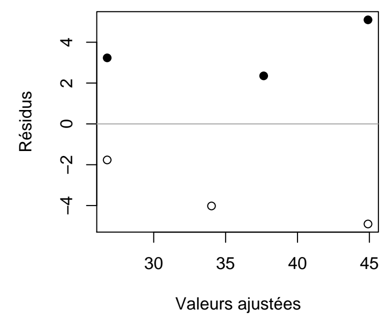


Figure 11.5 – Tracé des résidus du modèle hauteur-diamètre pour trois arbres pour chaque parcelle.

### 11.2.4 Décomposition

Le dilemme documenté dans la section 11.2.3 a au moins deux solutions. La première est l'utilisation de modèles à effets mixtes, l'autre, que nous n'avons pas encore vu ici, est explicite en modélisant une structure de corrélation à l'aide des moindres carrés généralisés.

L'approche des modèles à effets mixtes consiste à décomposer la variation inconnue en plus petits morceaux, chacun desquels satisfait aux hypothèses nécessaires. Imaginez que nous pourrions prendre les six valeurs résiduelles présentées à la figure 11.4, qui ont une structure de corrélation au niveau parcelle, et se décomposent en deux erreurs de niveau parcelle et des erreurs de taille à l'intérieur des parcelles. C'est-à-dire qu'au lieu de :

$$y_{ij} - \hat{y}_{ij} = \hat{\varepsilon}_{ij} \tag{11.2}$$

on pourrait essayer :

$$y_{ij} - \hat{y}_{ij} = \hat{b}_i + \hat{\varepsilon}_{ij} \tag{11.3}$$

Alors nous avons simplement besoin de supposer que :
- La vraie relation est linéaire.
- $b_i \sim \mathcal{N}(0, \sigma_b^2)$
- $\varepsilon_{ij} \sim \mathcal{N}(0, \sigma^2)$
- Les $\varepsilon_{ij}$ sont indépendants.

Toutefois, au moment d'utiliser le modèle pour la prédiction, nous n'avons pas besoin de savoir l'identité de la parcelle, puisque les effets fixes ne l'exigent pas.

Cet exemple illustre l'utilisation d'effets aléatoires. Les effets aléatoires n'expliquent pas la variation, ce qui est le rôle des effets fixes. Les effets aléatoires organisent la variation ou de font respecter une structure plus complexe sur celle-ci, de telle manière qu'une correspondance est possible entre les hypothèses du modèle et les diagnostics. En fait, nous espérons que l'incertitude globale, mesurée en tant que racine de l'erreur quadratique moyenne, augmente chaque fois que nous ajustons autrement qu'avec les moindres carrés.

## 11.3 Un exemple simple

Nous commençons avec un exemple abstrait très simple. D'abord il faut charger le paquetage qui gère le code des effets mixtes,`nlme`.

```
> require(nlme)
> require(lattice)
```

Maintenant générons un jeu de données simple.

```
> straw <- data.frame(y = c(10.1, 14.9, 15.9, 13.1, 4.2, 4.8, 5.8,
+     1.2), x = c(1, 2, 3, 4, 1, 2, 3, 4), group = factor(c(1,
+     1, 1, 1, 2, 2, 2, 2)))
```

Traçons ces données (cf. figure 11.6).

```
> colours = c("red", "blue")
> plot(straw$x, straw$y, col = colours[straw$group])
```

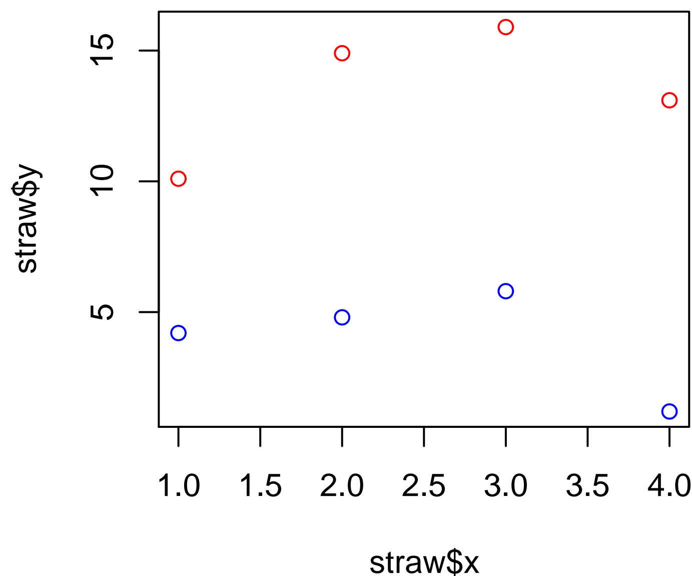


Figure 11.6 – Un simple jeu de données pour montrer l'utilisation des modèles à effets mixtes.

Pour chaque modèle ci-dessous, explorer les sorties en utilisant les commandes `summary()` de chaque modèle, essayer de déterminer quelles sont les différences entre les modèles et si la complexité semble être de plus en plus utile. Utilisez les commandes `anova()` dans ce dernier but.

**Moindres carrés ordinaires**
Ce modèle se contente d'essayer de prédire $y$ avec $x$. Avec l'algèbre nous écririons

$$y_i = \beta_0 + \beta_1 \times x_i + \varepsilon_i \tag{11.4}$$

où $\beta_0$ et $\beta_1$ sont des paramètres de population inconnus et $\varepsilon_i$ sont les résidus.

Les hypothèses suivantes sont requises :

- La vraie relation est linéaire.
- $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$
- Les $\varepsilon_i$ sont indépendants.

Il est à noter que les valeurs $\beta_0$ et de $\beta_1$ qui minimisent la somme de carrés résiduelle sont les estimations des moindres carrés dans tous les cas et sont non biaisées et si la première hypothèse est vraie. Si les hypothèses 2 et 3 sont aussi vraies alors les estimations ont également d'autres propriétés intéressantes.

Le modèle est ajusté en **R** avec le code suivant :

```
> basic.1 <- lm(y ~ x, data = straw)
```

Donnons à chaque groupe son ordonnée à l'origine. Algébriquement,

$$y_i = \beta_{01} + \beta_{02} \times g_i + \beta_1 \times x_i + \varepsilon_i \tag{11.5}$$

où :
- $\beta_{01}$, $\beta_{02}$ et $\beta_1$ sont des paramètres fixes mais de population inconnue,
- $g_i$ une variable indicatrice de valeur 0 pour le premier groupe et 1 pour le second et
- $\varepsilon_i$ les résidus.

Les mêmes hypothèses sont requises que pour le modèle (11.4).

Le modèle est ajusté en **R** avec le code suivant :

```
> basic.1 <- lm(y ~ x + group, data = straw)
```

Donnons à chaque groupe son ordonnée à l'origine *et* sa pente.

$$y_i = \beta_{01} + \beta_{02} \times g_i + (\beta_{11} + \beta_{12} \times g_i) \times x_i + \varepsilon_i \tag{11.6}$$

où :
- $\beta_{01}$, $\beta_{02}$, $\beta_{11}$ et $\beta_{12}$ sont des paramètres fixes mais de population inconnue,
- $g_i$ une variable indicatrice de valeur 0 pour le premier groupe et 1 pour le second et
- $\varepsilon_i$ les résidus.

Les mêmes hypothèses sont requises que pour le modèle (11.4).

Le modèle est ajusté en **R** avec le code suivant :

```
> basic.1 <- lm(y ~ x * group, data = straw)
```

**Effets mixtes**
Nous devons maintenant convertir les données en objet groupé – une sorte spéciale de data-frame qui permet des commandes spéciales de `nlme`. Le `group` sera, par la présente, un effet aléatoire. Une commande que l'on peut maintenant utiliser est `augPred`, comme nous le verrons plus loin. Essayer ceci sur quelques modèles.

```
> straw.mixed <- groupedData(y ~ x | group, data = straw)
```

Ajustons maintenant le modèle à effets mixtes de base qui permet aux ordonnées à l'origine de varier de façon aléatoire entre les groupes. Nous ajouterons un indice pour clarifier les choses.

$$y_{ij} = \beta_0 + b_{0i} + \beta_1 \times x_{ij} + \varepsilon_{ij} \tag{11.7}$$

où :
- $\beta_0$ et $\beta_1$ sont des paramètres fixes mais de population inconnue,
- les $b_{0i}$ sont deux ordonnées à l'origine aléatoires spécifiques du groupe et
- $\varepsilon_i$ les résidus.

Les hypothèses suivantes sont requises :

- La vraie relation est linéaire.
- $b_{0i} \sim \mathcal{N}(0, \sigma^2_{b_{0i}})$
- $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$
- Les $\varepsilon_i$ sont indépendants.

Le modèle est ajusté en **R** avec le code suivant :

```
> basic.4 <- lme(y ~ x, random = ~1 | group, data = straw.mixed)
```

La syntaxe de `random` peut être un peu confuse. Ici, nous disons à **R** de permettre à chaque groupe d'avoir sa propre ordonnée à l'origine aléatoire. Si nous avions voulu laisser chaque groupe avoir ses propres pente et ordonnée à l'origine, nous aurions écrit `random = x`. Si nous voulions laisser chaque groupe a sa propre pente, mais pas l'ordonnée à l'origine, nous aurions écrit `random = x - 1`.

Nous pouvons regarder le modèle dans un graphique utile nommé *tracé de prédiction améliorée*. Ce graphique fournit un diagramme de dispersion, par groupe avec une droite d'ajustement qui représente les prévisions du modèle (cf. figure 11.7). Nous devons aussi vérifier la pertinence à nos hypothèses des diagnostics de régression, mais nous avons si peu de données ici que les exemples ne sont pas utiles. Nous allons développer ces idées au cours de l'étude de cas qui va suivre.

```
> plot(augPred(basic.4))
```

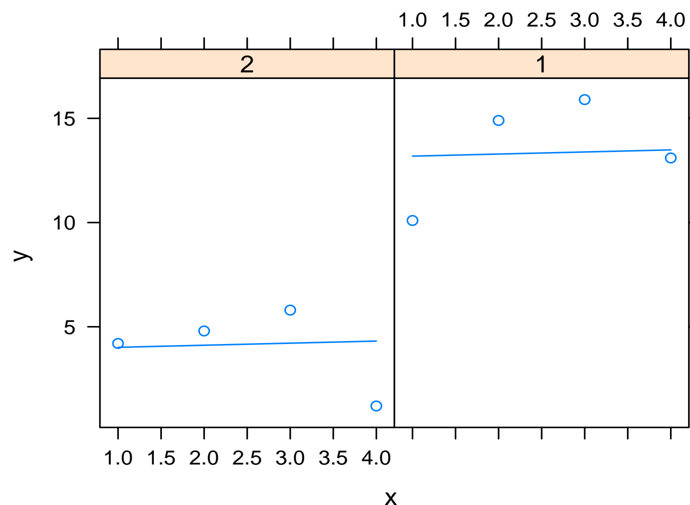


Figure 11.7 – Un tracé de prédiction améliorée du modèle à effets mixtes de base avec les ordonnées à l'origine aléatoires ajustées au jeu de données exemple.

Si nous sommes satisfaits des diagnostics du modèle, nous pouvons alors examiner sa structure et les estimations, en utilisant la fonction `summary()`. Cette fonction présente une collection d'informations utiles sur le modèle. Ici, nous obtenons la structure par défaut d'un objet `summary.lme`. La structure peut changer dans le futur, mais l'information essentielle devrait demeurer la même.

```
> summary(basic.4)
```

En premier, l'objet `data.frame` est identifié, et les statistiques d'ajustement rapportées, y compris l'« An Information Criterion » d'Akaike, le « Bayesian Information Criterion » de Schwartz et la log-vraisemblance

```
Linear mixed-effects model fit by REML
 Data: straw.mixed
       AIC      BIC    logLik
  43.74387 42.91091 -17.87193
```

La structure des effets aléatoires est ensuite décrite et des estimations sont fournies pour les paramètres. Ici nous avons une ordonnée à l'origine pour chaque groupe et l'écart-type est proposé avec celui des résidus intra groupe.

```
Random effects:
 Formula: ~1 | group
        (Intercept) Residual
StdDev:    6.600767 2.493992
```

La structure des effets fixes est ensuite décrite dans une présentation de tableau $t$. Les corrélations estimées entre les effets fixes suivent.

```
Fixed effects: y ~ x
            Value Std.Error DF   t-value p-value
(Intercept)   8.5  5.142962  5 1.6527441  0.1593
x             0.1  0.788670  5 0.1267958  0.9040
 Correlation:
  (Intr)
x -0.383
```

La distribution des résidus intra groupe, que l'on nomme aussi *résidus les plus internes* dans le contexte des modèles strictement hiérarchiques de Pinheiro et Bates (2000) [15], est ensuite décrite.

```
Standardized Within-Group Residuals:
       Min         Q1        Med         Q3        Max
-1.2484737 -0.4255492  0.1749470  0.6387984  1.0078956
```

Enfin la structure hiérarchique du modèle et des données est présentée.

```
Number of Observations: 8
Number of Groups: 2
```

Ajustons ensuite une unique variance à chaque groupe. Le modèle aura la même forme que dans l'équation 11.7, mais les hypothèses seront différentes. On suppose maintenant que :

- La vraie relation est linéaire.
- $b_{0i} \sim \mathcal{N}(0, \sigma^2_{b_{0i}})$
- $\varepsilon_{1j} \sim \mathcal{N}(0, \sigma^2_{b_{01}})$
- $\varepsilon_{2j} \sim \mathcal{N}(0, \sigma^2_{b_{02}})$
- $Cov(\varepsilon_{ab}, \varepsilon_{cd}) = 0$ pour $a \neq c$ ou $b \neq d$.

Le modèle est ajusté en **R** avec le code suivant :

```
> basic.5 <- lme(y ~ x, random = ~1 | group, weights = varIdent(form = ~1 |
+     group), data = straw.mixed)
```

La sortie de la fonction `summary()` est pratiquement identique à la précédente en structure, avec l'ajout d'une nouvelle section qui récapitule le modèle de variance nouvellement ajouté. Nous ne montrons que la nouvelle portion.

```
> summary(basic.5)

Variance function:
 Structure: Different standard deviations per stratum
 Formula: ~1 | group
 Parameter estimates:
       2        1
1.000000 1.327843
```

Enfin, permettons une auto-corrélation temporelle à l'intérieur de chaque groupe. Là encore, le modèle aura la même forme que dans l'équation 11.7, mais les hypothèses seront différentes. On doit supposer maintenant que :

- La vraie relation est linéaire.
- $b_{0i} \sim \mathcal{N}(0, \sigma^2_{b_1})$
- $\varepsilon_{1j} \sim \mathcal{N}(0, \sigma^2_{b_{01}})$
- $\varepsilon_{2j} \sim \mathcal{N}(0, \sigma^2_{b_{02}})$

- $Cov(\varepsilon_{ab}, \varepsilon_{cd}) = \rho$ pour $b \neq d$
- Les $\varepsilon_{ij}$ sont indépendants sinon.

Le modèle est ajusté en **R** avec le code suivant :

```
> basic.6 <- lme(y ~ x, random = ~1 | group, weights = varIdent(form = ~1 |
+     group), correlation = corAR1(), data = straw.mixed)
```

La sortie de la fonction `summary()` est encore pratiquement identique à la précédente en structure, avec l'ajout d'une nouvelle section qui récapitule le modèle de corrélation nouvellement ajouté. Nous ne montrons que la nouvelle portion.

```
> summary(basic.6)

Correlation Structure: AR(1)
 Formula: ~1 | group
 Parameter estimate(s):
      Phi
0.8107325
```

On peut récapituler les quelques différences dans un graphique (cf. figure 11.8).

```
> opar <- par(las = 1)
> colours <- c("blue", "darkgreen", "plum")
> plot(straw$x, straw$y)
> for (g in 1:2) lines(straw$x[straw$group == levels(straw$group)[g]],
+     fitted(basic.1)[straw$group == levels(straw$group)[g]], col = colours[1])
> for (g in 1:2) lines(straw$x[straw$group == levels(straw$group)[g]],
+     fitted(basic.2)[straw$group == levels(straw$group)[g]], col = colours[2])
> for (g in 1:2) lines(straw$x[straw$group == levels(straw$group)[g]],
+     fitted(basic.4)[straw$group == levels(straw$group)[g]], col = colours[3])
> legend(2.5, 13, lty = rep(1, 3), col = colours, legend = c("Moyenne seule",
+     "Ordon. origine Fixes", "Ordon. origine Aléatoires"))
> par(opar)
```

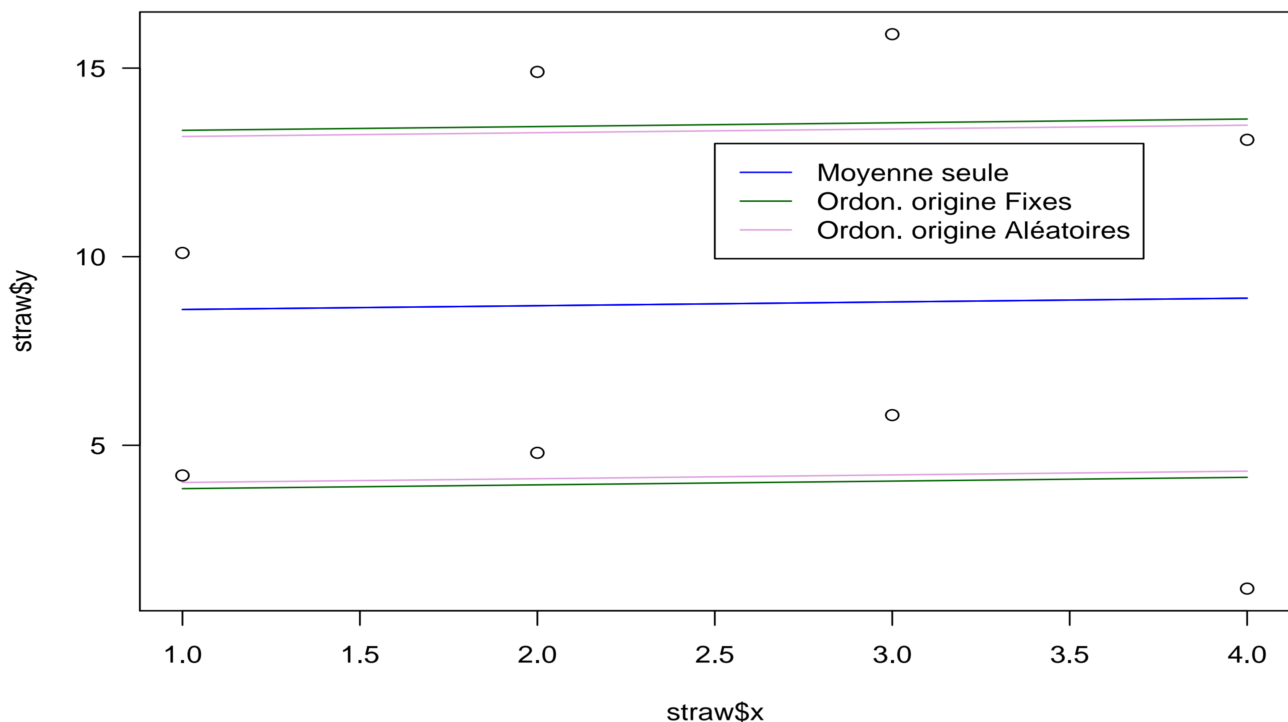


Figure 11.8 – Un tracé exemple montrant la différence entre basic.1 (simple droite), basic.2 (ordonnées à l'origine fixes) et basic.4 (ordonnées à l'origine aléatoires).

### 11.3.1 La toute fin

Il existe de nombreuses représentations différentes du modèle linéaire à effets mixte. Nous adopterons celle suggérée par Laird et Ware (1982)[13] :

$$\begin{aligned}\mathbf{Y} &= \mathbf{X}\boldsymbol{\beta} + \mathbf{Zb} + \boldsymbol{\epsilon}\\ \mathbf{b} &\sim \mathcal{N}(\mathbf{0}, \mathbf{D})\\ \boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, \mathbf{R})\end{aligned}$$

Ici, $\mathbf{D}$ et $\mathbf{R}$ sont construites de préférence avec un petit nombre de paramètres à estimer à partir des données. Voyons d'abord une estimation avec le maximum de vraisemblance.

### 11.3.2 Maximum de vraisemblance

Rappelons que derrière le principe du maximum de vraisemblance était de trouver la suite d'estimations des paramètres qui ont été le mieux soutenu par les données. Cela a commencé par écrire la densité conditionnelle des observations. Par exemple, la distribution pour une simple observation de la loi normale est la suivante :

$$f(y_i|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(y_i-\mu)^2}{2\sigma^2}}$$

Aussi, si $\mathbf{Y} \stackrel{d}{=} \mathcal{N}(\mu, \mathbf{V})$, alors par définition :

$$f(\mathbf{Y}|\mu, \mathbf{V}) = \frac{|\mathbf{V}|^{-\frac{1}{2}}}{(2\pi)^{\frac{n}{2}}} e^{-\frac{1}{2}(\mathbf{Y}-\mu)'\mathbf{V}^{-1}(\mathbf{Y}-\mu)}$$

Aussi en termes du modèle linéaire $\mathbf{Y} = \mathbf{X}\boldsymbol{\beta}$, la densité conditionnelle est

$$f(\mathbf{Y}|\mathbf{X}, \boldsymbol{\beta}, \mathbf{V}) = \frac{|\mathbf{V}|^{-\frac{1}{2}}}{(2\pi)^{\frac{n}{2}}} e^{-\frac{1}{2}(\mathbf{Y}|\mathbf{X}\boldsymbol{\beta})'\mathbf{V}^{-1}(\mathbf{Y}-\mathbf{X}\boldsymbol{\beta})}$$

En inversant les conditions et en prenant le logarithme, il vient :

$$\mathcal{L}(\boldsymbol{\beta}, \mathbf{V}|\mathbf{Y}, \mathbf{X}) = -\frac{1}{2}ln(|\mathbf{V}|) - \frac{n}{2}ln(2\pi) - \frac{1}{2}(\mathbf{Y}|\mathbf{X}\boldsymbol{\beta})'\mathbf{V}^{-1}(\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}) \tag{11.8}$$

Notez que les paramètres qui nous intéressent sont maintenant intégrés à la vraisemblance. Résoudre pour ces paramètres ne devrait pas être plus difficile que de maximiser la vraisemblance, en théorie. Maintenant pour trouver $\hat{\beta}$ nous prenons la dérivée de $\mathcal{L}(\boldsymbol{\beta}, \mathbf{V}|\mathbf{Y}, \mathbf{X})$ par rapport à $\beta$ :

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\beta} = \frac{\mathrm{d}}{\mathrm{d}\beta}\left[-\frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})'\mathbf{V}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\right]$$

ce qui mène à :

$$\beta_{EMV} = (\mathbf{X}'\mathbf{V}^{-1}\mathbf{X})^{-1}\mathbf{X}'\mathbf{V}^{-1}\mathbf{Y}$$

mais cela ne fonctionne que *si l'on connaît* $\mathbf{V}$ !

Autrement, il faut maximiser la vraisemblance comme suit. D'abord substituer

$$(\mathbf{X}'\mathbf{V}^{-1}\mathbf{X})^{-1}\mathbf{X}'\mathbf{V}^{-1}\mathbf{Y}$$

à $\beta$ dans la vraisemblance. C'est-à-dire retirer toutes les instances de $\beta$ et les remplacer par cette valeur. Par ce moyen, $\beta$ est *sorti* de la vraisemblance qui devient, ainsi, une fonction des seules données et de la matrice de covariance $\mathbf{V}$. Cette matrice de covariance est elle-même une fonction des matrices de covariance des effets aléatoires, dont la structure comporte espérons-le seulement peu de paramètres inconnus organisés par les hypothèses du modèle.

Maximiser la vraisemblance résultante revient à estimer $\hat{(}V$ et à calculer les estimations des effets fixes via :

$$\hat{\beta}_{EMV} = (\mathbf{X}'\hat{\mathbf{V}}^{-1}\mathbf{X})^{-1}\mathbf{X}'\hat{\mathbf{V}}^{-1}\mathbf{Y} \tag{11.9}$$

Après de l'algèbre fastidieuse, largement documentée dans Schabenberger et Pierce (2002)[20], nous obtenons aussi les coefficients BLUP (Best Linear Unbiaised Predictor)

$$\hat{b}_{EMV} = \mathbf{DZ}'\hat{\mathbf{V}}(\mathbf{Y} - \mathbf{X}\hat{\beta}) \tag{11.10}$$

où $\mathbf{D}$ est la matrice de covariance des effets aléatoires.

### 11.3.3 Maximum de vraisemblance restreint

Il a été précédemment noté que les estimateurs de maximum de vraisemblance des paramètres de covariance sont généralement négativement biaisés. C'est parce que dans la détermination des effets fixes, nous sommes effectivement en mesure de prétendre que nous les connaissons, et, par conséquent, nous ne pouvons pas réduire les degrés de liberté de façon appropriée. Le maximum de vraisemblance *restreint* ou *résiduel* va pénaliser l'estimation basée sur la taille du modèle, et est donc une stratégie privilégiée. La méthode MVR n'est pas non biaisée, sauf dans certaines circonstances, mais il est moins biaisé que le MV seul.

Au lieu de maximiser la vraisemblance conditionnelle conjointe de **Y** nous le faisons pour une transformation linéaire presque arbitraire de **Y**, que nous désignons **K**. Elle est presque arbitraire dans la mesure où il n'existe que deux contraintes : **K** doit être de plein rang, ou ce serait la création d'observations par magie, et K doit être choisie de telle sorte que E[**K'Y**] = 0.

La meilleure façon de garantir que c'est le cas, c'est de faire en sorte que E[**K'X**] = 0, et que **K** ne compte pas plus de $n-p$ colonnes indépendantes, où p est le nombre de paramètres indépendants du modèle. Notez que nous aimerions que **K** ait autant de colonnes que possible parce que cela se traduira par plus de réalisations d'ajustement du modèle. Cela supprime les effets fixes de l'épreuve et, de ce fait, pénalise également l'estimation de la taille du modèle. Ainsi, la vraisemblance est restreinte par les effets fixes mis à **0**, donc, le maximum de vraisemblance est restreint. Enfin, noter que le fait d'avoir une colonne de 0 dans **K** n'ajoute pas du tout d'information à ce problème.

Ainsi, brièvement, le MVR contient l'application du MV, mais en remplaçant**Y** par **KY**, **X** par **0**, **Z** par **K'Z** et **V** par **K'VK**.

## 11.4 Étude de cas

Reprenons notre exposé brutal du modèle à effets mixtes :

$$\begin{aligned} \mathbf{Y} &= \mathbf{X}\boldsymbol{\beta} + \mathbf{Z}\mathbf{b} + \boldsymbol{\epsilon} \\ \mathbf{b} &\sim \mathcal{N}(\mathbf{0}, \mathbf{D}) \\ \boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \end{aligned}$$

**D** et **R** sont les matrices de covariance, construites avec un petit nombre de paramètres et dont la structure est *suggérée* par ce que l'on sait sur les données et peut être *testé* en comparant les modèles imbriqués.

### 11.4.1 Données stage

Un bref résumé : un échantillon de 66 arbres ont été sélectionnés dans les forêts nationales dans le nord et le centre de l'Idaho. Conformément à Stage (comm. pers. 2003), les arbres ont été sélectionnés dans un but précis plutôt que de manière aléatoire. Stage notait (1963) que les arbres sélectionnés « . . . semblent avoir été dominants tout au long de leur vie » et « . . . n'ont montré aucun signe visible de dommages en couronne, de fourches, de cime brisée, etc. » Le type d'habitat et le diamètre à 4'6" ont également été enregistrés pour chaque arbre, comme la forêt d'où elle proviennent. Chaque arbre a été ensuite découpé, et des mesures décennales ont été faites de la hauteur et du diamètre intérieur à l'écorce à 4'6". D'abord, un coup d'œil aux données dans la feuille de calcul de votre choix. Puis import des données comme suit :

```
> rm(list = ls())
> stage <- read.csv("../data/stage.csv")
> dim(stage)

[1] 542   7

> names(stage)

[1] "Tree.ID" "Forest"  "HabType" "Decade"  "Dbhib"   "Height"  "Age"

> sapply(stage, class)

  Tree.ID    Forest   HabType    Decade     Dbhib    Height       Age
"integer" "integer" "integer" "integer" "numeric" "numeric" "integer"
```

Du ménage est nécessaire. Commençons avec les facteurs.

```
> stage$Tree.ID <- factor(stage$Tree.ID)
> stage$Forest.ID <- factor(stage$Forest, labels = c("Kan", "Cd'A",
+     "StJoe", "Cw", "NP", "CF", "Uma", "Wall", "Ptte"))
> stage$HabType.ID <- factor(stage$HabType, labels = c("Ts/Pach",
+     "Ts/Op", "Th/Pach", "AG/Pach", "PA/Pach"))
```

Les mesures sont toutes impériales (c'était à peu près en 1960 !).

```
> stage$dbhib.cm <- stage$Dbhib * 2.54
> stage$height.m <- stage$Height/3.2808399
> str(stage)
```

```
'data.frame': 542 obs. of  11 variables:
 $ Tree.ID   : Factor w/ 66 levels "1","2","3","4",..: 1 1 1 1 1 2 2 2 2 2 ...
 $ Forest    : int  4 4 4 4 4 4 4 4 4 4 ...
 $ HabType   : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Decade    : int  0 1 2 3 4 0 1 2 3 4 ...
 $ Dbhib     : num  14.6 12.4 8.8 7 4 20 18.8 17 15.9 14 ...
 $ Height    : num  71.4 61.4 40.1 28.6 19.6 ...
 $ Age       : int  55 45 35 25 15 107 97 87 77 67 ...
 $ Forest.ID : Factor w/ 9 levels "Kan","Cd'A","StJoe",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ HabType.ID: Factor w/ 5 levels "Ts/Pach","Ts/Op",..: 5 5 5 5 5 5 5 5 5 5 ...
 $ dbhib.cm  : num  37.1 31.5 22.4 17.8 10.2 ...
 $ height.m  : num  21.76 18.71 12.22  8.72  5.97 ...
```

**Hauteur à partir du diamètre**
La prévision de la hauteur à partir du diamètre fournit des informations utiles et peu coûteuses. Il se peut que la relation hauteur–diamètre diffère entre les types d'habitat, les zones de climat, ou l'âge des arbres. Nous allons examiner le modèle hauteur–diamètre des arbres en utilisant un modèle à effets mixtes. Commençons par un cas simple, en utilisant seulement la plus ancienne mesure de chaque arbre qui fournit une.

```
> stage.old <- stage[stage$Decade == 0, ]
```

Notez que ce code oublie en fait un arbre, mais nous pouvons nous permettre de laisser aller pour cette démonstration.

Pour établir une ligne de base de normalité, adaptons d'abord le modèle en utilisant des moindres carrés ordinaires. Nous ôtons la seule observation de type d'habitat TS/Op. Cela causerait des problèmes autrement (l'effet de levier se trouverait très élevé).

```
> hd.lm.1 <- lm(height.m ~ dbhib.cm * HabType.ID, data = stage.old,
+     subset = HabType.ID != "Ts/Op")
```

Formellement, je pense qu'il est bon d'examiner les diagnostics du problème sur lequel le modèle repose, avant d'examiner le modèle lui-même, agissant ainsi dès que possible, donc en regardant la figure 11.9. Le graphique des résidus en fonction des valeurs ajustées (en haut à gauche) semble bon. Il n'y a pas de suggestion d'hétéroscédasticité. Le tracé Quantile-Quantile normal, s'il est un peu tortillé, semble raisonnablement droit. Il semble y avoir aucun point d'influence marquée (en bas à gauche ; toutes les distances de Cook sont <1).

```
> opar <- par(mfrow = c(2, 2), mar = c(4, 4, 4, 1))
> plot(hd.lm.1)
> par(opar)
```

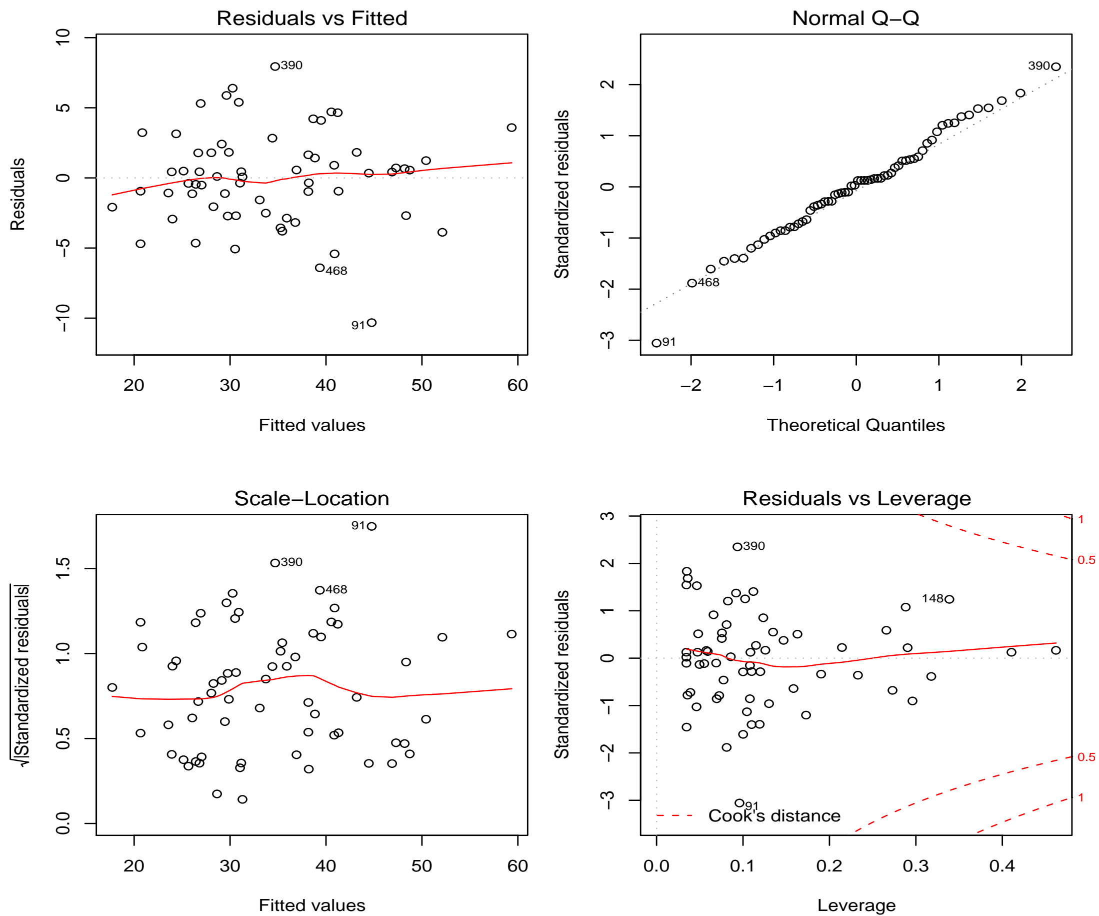


Figure 11.9 – Diagnostics de régression pour l'ajustement aux moindres carrés ordinaires du modèle hauteur–diamètre avec les types d'habitat sur les données Stage.

Bon, après avoir été jusqu'ici, nous devrions examiner le récapitulé du modèle.

```
> summary(hd.lm.1)

Call:
lm(formula = height.m ~ dbhib.cm * HabType.ID, data = stage.old,
    subset = HabType.ID != "Ts/Op")

Residuals:
     Min       1Q   Median       3Q      Max
-10.3210  -2.1942   0.2218   1.7992   7.9437

Coefficients:
                           Estimate Std. Error t value Pr(>|t|)
(Intercept)                 8.33840    4.64118   1.797   0.0778 .
dbhib.cm                    0.58995    0.08959   6.585 1.67e-08 ***
HabType.IDTh/Pach           2.42652    5.78392   0.420   0.6764
HabType.IDAG/Pach           0.29582    5.13564   0.058   0.9543
HabType.IDPA/Pach           0.02604    5.96275   0.004   0.9965
dbhib.cm:HabType.IDTh/Pach -0.03224    0.10670  -0.302   0.7637
dbhib.cm:HabType.IDAG/Pach -0.08594    0.10116  -0.850   0.3992
dbhib.cm:HabType.IDPA/Pach -0.10322    0.11794  -0.875   0.3852
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.551 on 56 degrees of freedom
Multiple R-squared: 0.8748,        Adjusted R-squared: 0.8591
F-statistic: 55.89 on 7 and 56 DF,  p-value: < 2.2e-16
```

Par comparaison : les quantitézs suivantes sont en mètres. La première est l'écart-type des mesures de hauteur. La seconde est l'écart-type des mesures de hauteur conditionnées par les mesures de diamètre *et* le modèle.

```
> sd(stage.old$height.m)

[1] 9.468042

> summary(hd.lm.1)$sigma

[1] 3.551062
```

Il est également intéressant de savoir la quantité de variation expliquée par l'information « type d'habitat ». Nous pouvons l'évaluer de la même manière. Ici nous ne nous inquiéterons pas des diagnostics, bien que cela dusse être fait.

```
> summary(lm(height.m ~ dbhib.cm, data = stage.old, subset = HabType.ID !=
+     "Ts/Op"))$sigma

[1] 4.101350
```

Pas plus ?

**Effets mixtes**

Sur la base de notre connaissance de l'emplacement de forêts nationales, il est raisonnable de penser qu'il y aura des similitudes entre les arbres qui poussent dans la même forêt *relativement à la la population des arbres.*

Toutefois, nous aimerions créer un modèle qui ne se fonde pas sur la connaissance de la forêt, c'est-à-dire un modèle qui puisse raisonnablement être utilisé pour les arbres d'autres forêts. Ceci est acceptable pour autant que nous soyons disposés à croire que l'échantillon d'arbres que nous utilisons est représentatif des conditions pour lesquelles nous souhaitons appliquer le modèle. En l'absence d'autre information, il s'agit d'une question de jugement. Supposons-le pour le moment.

Ensuite, en se fondant sur l'information ci-dessus, la forêt nationale sera un effet aléatoire, et le type d'habitat un effet fixe. C'est-à-dire que nous tenons à construire un modèle qui peut être utilisé pour toute forêt, qui pourrait être plus précis s'il est correctement utilisé dans une forêt nationale particulière et fournit des estimations uniques pour le type d'habitat. Nous pourrons nous inquiéter plus tard de l'utilité de la connaissance du type d'habitat et si nous voulons l'inclure dans le modèle.

Donc, nous avons deux effets aléatoires : les forêts nationales et les arbres dans les forêts nationales. Nous avons un effet fixe de base : diamètre à hauteur de poitrine à l'intérieur de l'écorce, avec d'éventuels ajouts : l'âge et le type d'habitat. Le plus bas niveau d'unité dans l'échantillonnage sera l'arbre, imbriqué dans la forêt nationale.

Il convient de fournir une structure de base à **R**. La structure permettra à **R** de créer des diagnostics graphiques utiles plus tard dans l'analyse. Notez qu'il vous suffit d'un `require()` d'un paquetage une fois par session, mais ce ne serait pas de mal de le faire plus souvent. Ici nous les disperserons de façon libérale pour vous rappeler quels paquetages vous devriez utiliser.

```
> require(nlme)
> stage.old <- groupedData(height.m ~ dbhib.cm | Forest.ID, data = stage.old)
```

Maintenant regardons notre modèle.

$$y_{ij} = \beta_0 + b_{0i} + \beta_1 \times x_{ij} + \varepsilon_{ij} \tag{11.11}$$

$y_{ij}$ est la hauteur de l'arbre $j$ de la forêt $i$, $x_{ij}$ est le diamètre du même arbre. $\beta_0$ et $\beta_1$ sont des paramètres fixes mais inconnus et $\beta_1$ sont les ordonnées à l'origine aléatoires et inconnues. Plus tard nous aimerions savoir si la pente varie aussi avec la forêt. Ainsi, sous forme matricielle,

$$\mathbf{Y} = \boldsymbol{\beta}\mathbf{X} + \mathbf{b}\mathbf{Z} + \boldsymbol{\epsilon} \tag{11.12}$$

**Y** est la colonne des hauteurs d'arbre, **X** celle des diamètres avec une matrice de 0 et de 1 pour affecter les observations aux différents types d'habitat, à côté d'une colonne pour les âges, si nécessaire. $\boldsymbol{\beta}$ sera le vecteur des estimations de paramètres. **Z** sera une matrice de 0 et de 1 pour affecter les observations des différentes forêts. **b** sera un vecteur de moyennes pour les forêts et les arbres dans les forêts. Enfin, construisons **D** comme matrice identité $9 \times 9$ multipliée par une constante $\sigma_h^2$ et **R** comme matrice identité $66 \times 66$ multipliée par une constante $\sigma^2$.

```
> hd.lme.1 <- lme(height.m ~ dbhib.cm, random = ~1 | Forest.ID,
+     data = stage.old)
```

Des fonctions automatiques sont disponibles pour extraire et tracer les différentes pièces du modèle. Je préfère les extraire et choisir mes propres méthodes de traçage. Je vous recommande de faire la même chose. Pour les versions pré-programmées voir Pinheiro et Bates (2000)[15]. Une rapide rafale de jargon : pour les modèles hiérarchiques il y a plusieurs niveaux de valeurs ajustées et de résidus. Pinheiro et Bates (2000)[15] adoptent l'approche suivante : les résidus et les valeurs ajustées les plus externes sont conditionnés seulement sur les effets fixes, les résidus et les valeurs ajustées les plus internes sont conditionnés sur tous les effets, fixes et aléatoires, et il y a autant de niveaux entre ces deux extrêmes qui sont nécessaire. Ainsi, dans un modèle à deux niveaux comme celui-ci,

- Les résidus les plus externes sont ceux calculés à partir des valeurs ajustées les plus externes, celles qui sont calculées avec les seuls effets fixes. On y fait référence comme $r_0$.

$$r_0 = y_{ij} - \hat{\beta}_0 - \hat{\beta}_1 \times x_{ij} \tag{11.13}$$

- Les résidus les plus internes sont ceux calculés à partir des valeurs ajustées les plus internes, celles qui sont calculées avec les effets fixes et aléatoires. On y fait référence comme $r_1$.

$$r_1 = y_{ij} - \hat{\beta}_0 - \hat{b}_{0i} - \hat{\beta}_1 \times x_{ij} \tag{11.14}$$

En outre, les effets mixtes nous offrent trois types de résidus les plus internes et les plus externes :

1. résidus *habituels*, différence entre observation et prédiction ,
2. résidus *de Pearson*, qui sont les résidus habituels normés par l'écart type et
3. résidus *normalisés*, qui sont les résidus de Pearson pré-multipliés par l'inverse de la racine carrée de la matrice de corrélation estimée du modèle.

Les hypothèses clefs que nous faisons pour notre modèle sont que :

1. la structure du modèle est correctement spécifiée ;
2. les effets aléatoires sont normalement distribués ;
3. les résidus les plus internes sont normalement distribués ;
4. les résidus les plus internes sont homoscédastiques intra et inter groupes et
5. les résidus les plus internes sont indépendants intra groupes.

Notez que nous ne faisons aucune hypothèse sur les résidus les plus externes. Cependant, ils sont utiles pour récapituler les éléments de performance du modèle.

Nous devrions construire des graphiques diagnostiques pour vérifier ces hypothèses. Notez que dans certains cas, les hypothèses sont énoncées de façon largement indéfendable. Par conséquent, la stratégie raisonnable est de vérifier les conditions qui peuvent être interprétées dans le contexte du plan d'expérience, des données, et du modèle. Par exemple, il existe une infinité de façons où les résidus les plus internes pourraient être hétéroscédastiques. Quelles sont les façons importantes ? La situation la plus susceptible de conduire à des problèmes si la variance des résidus est une fonction de quelque chose, qu'il s'agisse d'un effet fixe ou aléatoire.

Plutôt que de faire confiance à ma capacité d'anticiper ce que les programmeurs entendent par les étiquettes, etc., je veux savoir ce qui se passe dans chacun de mes tracés. La meilleure façon de le faire est de les montrer ici moi-même. Pour examiner chacune des hypothèses, à la suite, j'ai construit bloc de graphiques ci-après. Ils sont présentés dans la figure 11.10.

1. Un tracé des valeurs ajustées les plus externes en fonction des valeurs observées de la variable réponse. Ce graphique permet un récapitulatif général du pouvoir explicatif du modèle.
   (a) Quelle est la quantité de variation expliquée ?
   (b) Qu'est-ce qui reste ?
   (c) Existe t'il un manque d'ajustement quelque part ?
2. Un tracé des valeurs ajustées les plus internes en fonction des résidus de Pearson les plus internes. Ce graphique permet un contrôle de l'hypothèse d'une structure de modèle correcte.

(a) Y a t'il une courbure ?

(b) Est-ce que les résidus sont ventilés ?

3. Un tracé quantile-quantile des effets aléatoires estimés, pour contrôler s'ils sont normalement distribués avec une variance constante.

   (a) Les points suivent-ils une ligne droite, ou montrent-ils une asymétrie ou un aplatissement ?

   (b) Est-ce que les valeurs aberrantes sont évidentes ?

4. Un tracé quantile-quantile des résidus de Pearson pour contrôler s'ils sont normalement distribués avec une variance constante.

   (a) Les points suivent-ils une ligne droite, ou montrent-ils une asymétrie ou un aplatissement ?

   (b) Est-ce que les valeurs aberrantes sont évidentes ?

5. Un tracé de boîtes à moustaches crantées des résidus de Pearson les plus internes en fonction de la variable de groupe, pour voir l'aspect de la distribution intra-groupe.

   (a) Les crans coupent-ils l'axe horizontal ?

   (b) Y-a t'il une direction marquée entre les médianes des résidus intra-groupes et l'effet aléatoire estimé ?

6. Un diagramme de dispersion de la variance des résidus de Pearson intra-forêt en fonction des effets aléatoires de forêt.

   (a) Y-a t'il une tendance positive ou négative ?

**NB :** Je n'ai pas tendance à utiliser toutes les possibilités des tests statistiques disponibles pour l'homoscédasticité, la normalité, etc., pour les diagnostics. J'aime (Box, 1953)[2] : « ... faire des tests préliminaires sur les écarts est un peu comme une mise à la mer d'un bateau à rames pour savoir si les conditions sont suffisamment calme pour qu'un paquebot quitte le port ».

Nous utilisons le code suivant à produire figure 11.10. Bien sûr, il n'est pas nécessaire de laisser tous les graphiques de diagnostic sur la même figure.

```
> opar <- par(mfrow = c(3, 2), mar = c(4, 4, 3, 1), las = 1, cex.axis = 0.9)
> #### Tracé 1
> plot(fitted(hd.lme.1, level = 0), stage.old$height.m, xlab = "Valeurs ajustées (hauteur m)",
+     ylab = "Valeurs observées (hauteur m)", main = "Structure du modèle (I)")
> abline(0, 1, col = "blue")
> #### Tracé 2
> scatter.smooth(fitted(hd.lme.1), residuals(hd.lme.1, type = "pearson"),
+     xlab = "Valeurs ajustées", ylab = "Résidus les plus internes",
+     main = "Structure du modèle (II)")
> abline(h = 0, col = "red")
> #### Tracé 3
> ref.forest <- ranef(hd.lme.1)[[1]]
> ref.var.forest <- tapply(residuals(hd.lme.1, type = "pearson",
+     level = 1), stage.old$Forest.ID, var)
> qqnorm(ref.forest, main = "Q-Q Normal - Effets aléatoires de forêt")
> qqline(ref.forest, col = "red")
> #### Tracé 4
> qqnorm(residuals(hd.lme.1, type = "pearson"), main = "Q-Q Normal - Résidus")
> qqline(residuals(hd.lme.1, type = "pearson"), col = "red")
> #### Tracé 5
> boxplot(residuals(hd.lme.1, type = "pearson", level = 1) ~ stage.old$Forest.ID,
+     ylab = "Résidus les plus internes", xlab = "Forêt nationale",
+     notch = T, varwidth = T, at = rank(ref.forest))
> axis(3, labels = format(ref.forest, dig = 2), cex.axis = 0.8,
+     at = rank(ref.forest))
> abline(h = 0, col = "darkgreen")
> #### Tracé 6
> plot(ref.forest, ref.var.forest, xlab = "Effets aléatoires de forêt",
+     ylab = "Variance des résidus intra-forêt")
> abline(lm(ref.var.forest ~ ref.forest), col = "purple")
> par(opar)
```

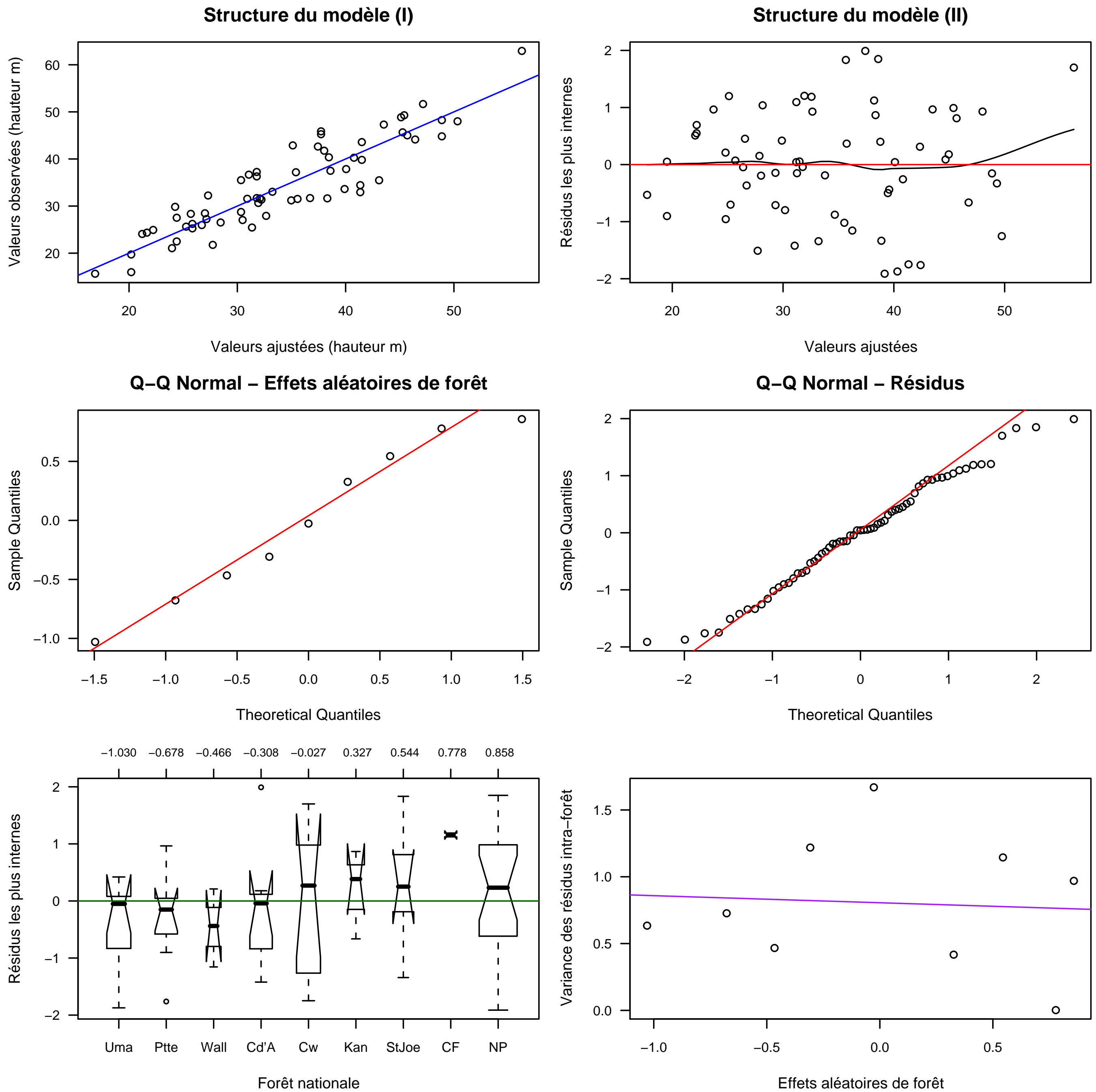


Figure 11.10 – Diagnostics sélectionnés pour l'ajustement en effets mixtes du rapport hauteur/diamètre selon le type d'habitat et la forêt nationale sur les données Stage.

Tous les les diagnostics de résidus semblent bons.

Acceptons le modèle tel qu'il est pour le moment et examinons le récapitulatif numérique (cf. section 11.3 pour les détails de ce qui suit).

```
> summary(hd.lme.1)

Linear mixed-effects model fit by REML
 Data: stage.old
       AIC      BIC    logLik
  376.6805 385.2530 -184.3403

Random effects:
 Formula: ~1 | Forest.ID
        (Intercept) Residual
StdDev:    1.151405 3.937486
```

```
Fixed effects: height.m ~ dbhib.cm
               Value Std.Error DF   t-value p-value
(Intercept) 6.58239 1.7763571 55  3.705556   5e-04
dbhib.cm    0.57036 0.0335347 55 17.008062   0e+00
 Correlation:
         (Intr)
dbhib.cm -0.931

Standardized Within-Group Residuals:
        Min          Q1         Med          Q3         Max
-1.91215622 -0.70233393  0.04308139  0.81189065  1.99133843

Number of Observations: 65
Number of Groups: 9
```

1. Ici, nous avons l'ensemble des paramètres d'ajustement du modèle, y compris la log-vraisemblance (on se souvient que c'est la quantité à maximiser pour obtenir l'ajustement), et les statistiques AIC et BIC. Les effets fixes sont déterminés par la log-vraisemblance, de sorte que cette dernière n'est fonction que des données et de deux paramètres : $\sigma_h^2$ et $\sigma^2$.
2. La formule nous rappelle ce que nous avons demandé : que la forêt est un effet aléatoire, et qu'une unique ordonnée à l'origine doit être ajustée pour chaque niveau de forêt. Les racines carrées des estimations des deux paramètres sont également présentes.
3. Une autre mesure de qualité du modèle est la RMSE, qui est l'estimation de la déviation standard des résidus de la réponse subordonnés aux seuls effets fixes. Notez que 3.94 n'est pas la RMSE, c'est plutôt une estimation de l'écart-type des résidus de la réponse subordonnés aux effets fixes et aléatoires. L'obtention d'une RMSE est relativement facile parce que les effets aléatoires et les résidus sont supposés être indépendants.

$$\text{RMSE} = \sqrt{\sigma_h^2 + \sigma^2} = 4.1$$

   La dernière mesure de qualité du modèle que l'on peut obtenir ici est la corrélation intra-classe. C'est la variance de l'effet aléatoire divisée par la somme des variances des effets aléatoires et des résidus.

$$\rho = \frac{\sigma_h^2}{\sigma_h^2 + \sigma^2} = 0.0788$$

   ainsi environ 7,9% de la variation en hauteur (celle qui n'est pas expliquée par le diamètre) est expliquée par la forêt. C'est pas beaucoup.
4. On a maintenant un rappel du modèle à effets fixes et leur estimation. Plusieurs colonnes donnent :
   (a) la valeur de l'estimation,
   (b) son erreur standard (non identique ici à cause du manque d'équilibrage),
   (c) le degré de liberté (simplement mystérieux pour de multiples raisons),
   (d) la valeur de $t$ associée au test de significativité de l'hypothèse nulle contre l'hypothèse alternative (non nulle), qui ne veut pas dire grand chose pour ce modèle et
   (e) la valeur p associée à ce test déraisonnable.
5. La matrice de corrélation des estimations des effets fixes. Elle est estimée à partir de la matrice du plan d'expérience. Elle est issuede la matrice de covariance des effets fixes qui peut être estimée par

$$(\mathbf{X}'\mathbf{V}^{-1}\mathbf{X})^{-1}$$

6. Information sur les résidus intra-groupe. Sont-ils symétriques ? Y-a t'il des valeurs aberrantes évidentes ? Ces valeurs sont à comparer avec ce que l'on sait d'une distribution normale standard, avec une médiane autour de 0, le premier quartile à -0.674 et le troisième à 0.674.
7. Et enfin, la confirmation que l'on a le nombre correct d'observations et de groupes. C'est une conclusion utile à tirer ; elle nous confirme d'avoir ajusté le modèle que nous pensions !

Un récapitulé explicatif de la puissance du modèle peut être obtenu par :

```
> anova(hd.lme.1)

            numDF denDF   F-value p-value
(Intercept)     1    55 2848.4436  <.0001
dbhib.cm        1    55  289.2742  <.0001
```

### Conception plus poussée

Nous allons maintenant traiter les données hauteur/diamètre du Grand fir de Stage (1963) d'une manière différente. Nous avons en fait de nombreuses mesures de hauteur et diamètre pour chaque arbre. Cela semble du gaspillage d'utiliser seulement le plus grand. Nous allons toujours supposer que les forêts nationales représentent différentes sources sélectionnées pour leurs variations climatiques, et que le type d'habitat représente un traitement choisi au hasard de l'environnement (non, ce n'est probablement pas vrai, mais nous supposerons que ça l'est). Il s'agit d'un plan d'expérience à blocs randomisés, où les effets bloc et traitement sont croisés. Cette fois nous allons utiliser toutes les données. Auparavant, nous n'avions pris que la première mesure.

Comment le modèle change t'il ? Comme toujours, nous commençons par la mise en place des données :

```
> require(nlme)
> stage <- groupedData(height.m ~ dbhib.cm | Forest.ID/Tree.ID,
+     data = stage)
```

Disons que, sur la base des informations ci-dessus, la forêt nationale sera un effet aléatoire, le type d'habitat et un candidat d'effet fixe. Donc, nous avons partout de un à trois effets fixes (`dbhib`, `age` et `habitat`) et de deux effets aléatoires (`forest` et `tree` dans la forêt). La variable réponse va maintenant être la mesure de hauteur, imbriquée dans l'arbre, éventuellement imbriquée dans le type d'habitat. Supposons, pour le moment, que les mesures soient indépendantes à l'intérieur de l'arbre (ce qui n'est certainement pas le cas). Maintenant, examinons notre modèle. Un simple et raisonnable modèle candidat est :

$$y_{ijk} = \beta_0 + b_{0i} + b_{0ij} + \beta_1 \times x_{ijk} + \varepsilon_{ijk} \tag{11.15}$$

$y_{ijk}$ est la hauteur de l'arbre $j$ de la forêt $i$ pour la mesure $k$, $x_{ijk}$ est le diamètre du même arbre. $\beta_0$ et $\beta_1$ sont des paramètres fixes mais inconnus, les $b_{0i}$ sont des ordonnées à l'origine aléatoires spécifiques de la forêt et $b_{0ij}$ des ordonnées à l'origine aléatoires spécifiques des arbres. Plus tard on regardera si la pente varie aussi avec la forêt. Ainsi, sous forme matricielle, on obtient :

$$\mathbf{Y} = \boldsymbol{\beta}\mathbf{X} + \mathbf{b}\mathbf{Z} + \boldsymbol{\epsilon} \tag{11.16}$$

- $\mathbf{Y}$ est le vecteur des mesures de hauteur. L'unité de base de $\mathbf{Y}$ sera une mesure à l'intérieur de l'arbre à l'intérieur d'une forêt. Cela fait 542 observations.
- $\mathbf{X}$ sera une matrice de 0, 1 et de diamètres pour affecter les observations aux différentes forêts nationales, aux différents diamètres d'arbre et aux instants de mesure.
- $\boldsymbol{\beta}$ sera le vecteur des estimations de paramètres.
- $\mathbf{Z}$ sera une matrice de 0 et de 1 pour affecter les observations aux différentes forêts et aux arbres dans les forêts.
- $\mathbf{b}$ sera un vecteur des moyennes des forêts et des arbres.
- $\mathbf{D}$ sera une matrice diagonale par bloc comprenant une matrice identité $9 \times 9$ multipliée par une constante $\sigma_f^2$ et ensuite une matrice carrée pour chaque forêt, matrice diagonale avec les variances sur les diagonales.
- $\mathbf{R}$ sera une matrice identité $9 \times 9$ multipliée par une constante $\sigma^2$.

```
> hd.lme.3 <- lme(height.m ~ dbhib.cm, random = ~1 | Forest.ID/Tree.ID,
+     data = stage)
```

Les hypothèses clefs que l'on fait à présent sont que :

1. la structure du modèle est correctement spécifiée,
2. les arbres et forêts sont des effets aléatoires normalement distribués,
3. les effets aléatoires arbres sont homoscédastiques à l'intérieur des effets aléatoires forêts,
4. les résidus les plus internes sont normalement distribués,
5. les résidus les plus internes sont homoscédastiques intra et inter effets aléatoires arbres.
6. les résidus les plus internes sont indépendants intra groupes.

Nous construisons encore des graphiques diagnostiques pour vérifier ces hypothèses. Pour les examiner à la suite, j'ai repris la série précédente avec des graphiques supplémentaires.

1. un tracé quantile-quantile de plus des effets aléatoires au niveau arbre, pour contrôler qu'ils sont normalement distribués et de variance constante.
   (a) Les points suivent-ils une ligne droite, ou montrent-ils une asymétrie ou un aplatissement ?
   (b) Est-ce que les valeurs aberrantes sont évidentes ?

2. des boîtes à moustaches crantées des effets aléatoires de niveau arbre selon les variables de groupe, pour voir l'aspect des distributions intra-groupe
   (a) Les crans coupent-ils l'axe horizontal ?
   (b) Y-a t'il une tendance entre les médianes des résidus intra-groupe et l'effet aléatoire estimé ?
3. un diagramme de dispersion de la variance des effets aléatoires de niveau arbre dans la forêt en fonction de l'effet aléatoire forêt.
   (a) Y-a t'il une tendance positive ou négative ?
4. Un tracé d'auto-corrélation des erreurs intra-abre.

Comme règle d'or, on a besoin de quatre graphiques plus trois pour chaque effet aléatoire. Comparez cela aux figures 11.11, 11.12 et 11.13. Chaque tracé devrait être, dans l'idéal, examiné séparément dans son propre contexte. En voici le code :

```
> opar <- par(mfrow = c(1, 3), mar = c(4, 4, 3, 1), las = 1, cex.axis = 0.9)
> plot(fitted(hd.lme.3, level = 0), stage$height.m, xlab = "Valeurs ajustées",
+     ylab = "Valeurs observées", main = "Structure du modèle (I)")
> abline(0, 1, col = "gray")
> scatter.smooth(fitted(hd.lme.3), residuals(hd.lme.3, type = "pearson"),
+     main = "Structure du modèle (II)", xlab = "Valeurs ajustées",
+     ylab = "Résidus les plus internes")
> abline(h = 0, col = "gray")
> acf.resid <- ACF(hd.lme.3, resType = "normal")
> plot(acf.resid$lag[acf.resid$lag < 10.5], acf.resid$ACF[acf.resid$lag <
+     10.5], type = "b", main = "Auto-corrélation", xlab = "Décalage",
+     ylab = "Corrélation")
> stdv <- qnorm(1 - 0.01/2)/sqrt(attr(acf.resid, "n.used"))
> lines(acf.resid$lag[acf.resid$lag < 10.5], stdv[acf.resid$lag <
+     10.5], col = "darkgray")
> lines(acf.resid$lag[acf.resid$lag < 10.5], -stdv[acf.resid$lag <
+     10.5], col = "darkgray")
> abline(0, 0, col = "gray")
> par(opar)
```

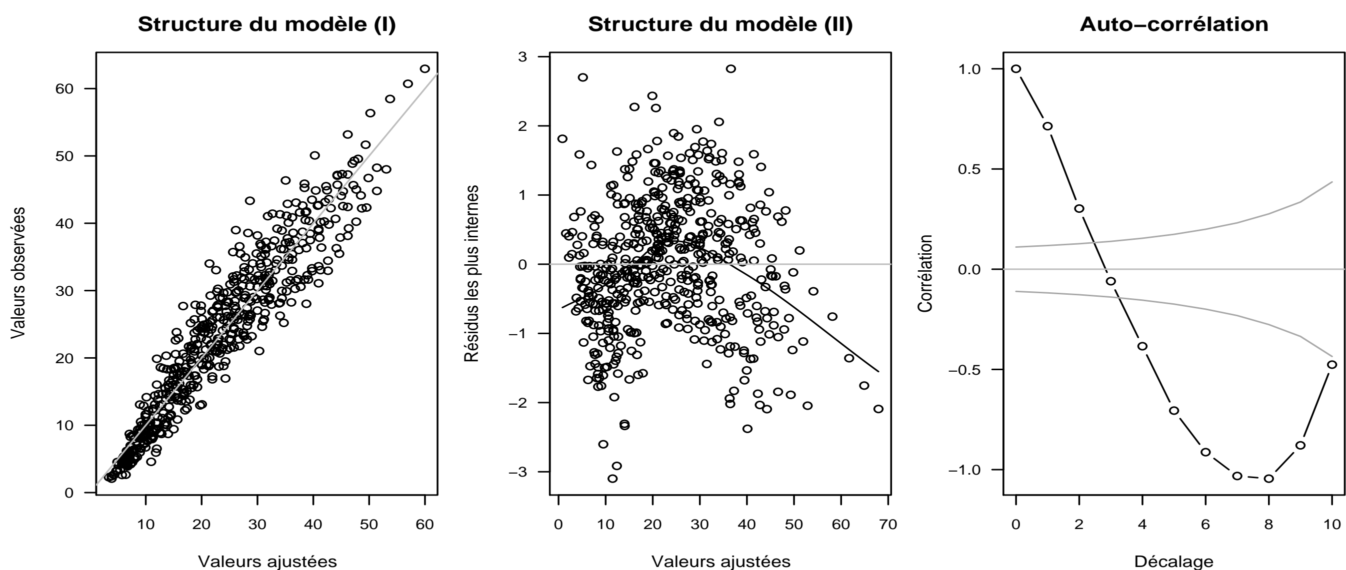


Figure 11.11 – Diagnostics généraux sélectionnés pour l'ajustement en effets mixtes du modèle hauteur et diamètre pour les données Stage.

```
> opar <- par(mfrow = c(1, 3), mar = c(4, 4, 3, 1), las = 1, cex.axis = 0.9)
> ref.forest <- ranef(hd.lme.3, level = 1, standard = T)[[1]]
> ref.tree <- ranef(hd.lme.3, level = 2, standard = T)[[1]]
> ref.tree.frame <- ranef(hd.lme.3, level = 2, augFrame = T, standard = T)
> ref.var.tree <- tapply(residuals(hd.lme.3, type = "pearson",
```

```
+     level = 1), stage$Tree.ID, var)
> ref.var.forest <- tapply(ref.tree, ref.tree.frame$Forest, var)
> qqnorm(ref.forest, main = "QQ plot: Forêt")
> qqline(ref.forest)
> qqnorm(ref.tree, main = "QQ plot: Arbre")
> qqline(ref.tree)
> qqnorm(residuals(hd.lme.3, type = "pearson"), main = "QQ plot: Résidus")
> qqline(residuals(hd.lme.3, type = "pearson"), col = "red")
> par(opar)
```

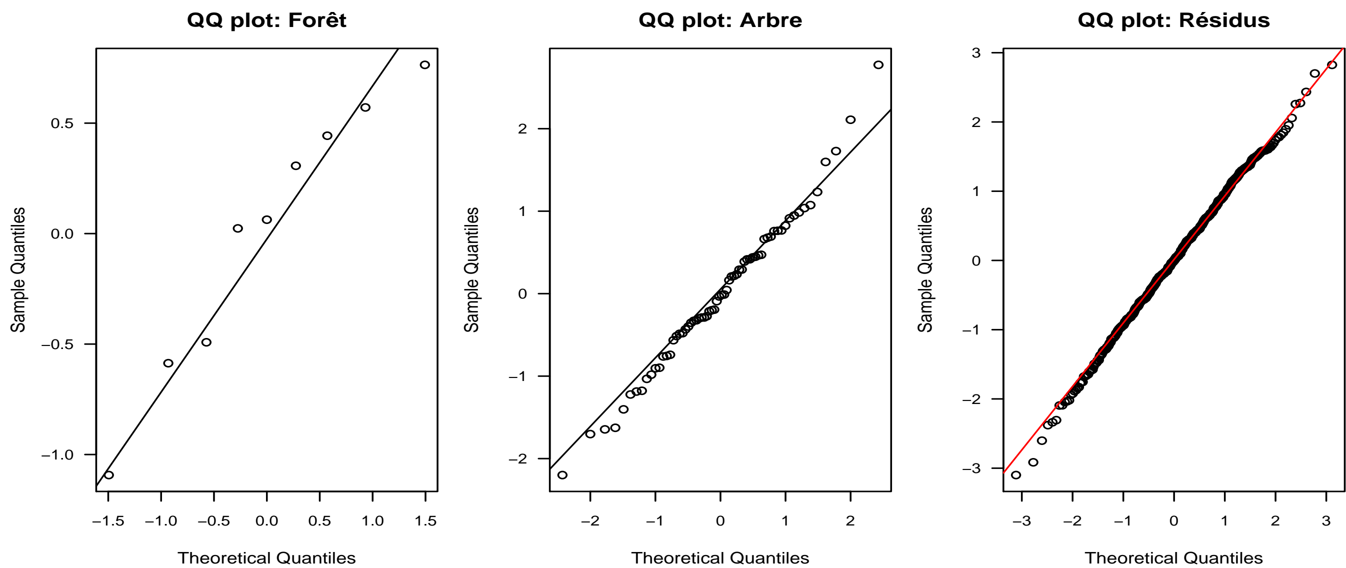


Figure 11.12 – Diagnostics Quantile-Quantile sélectionnés pour l'ajustement en effets mixtes du modèle hauteur et diamètre pour les données Stage.

```
> opar <- par(mfrow = c(2, 2), mar = c(4, 4, 3, 1), las = 1, cex.axis = 0.9)
> boxplot(ref.tree ~ ref.tree.frame$Forest, ylab = "Effets arbre",
+     xlab = "Forêt nationale", notch = T, varwidth = T, at = rank(ref.forest))
> axis(3, labels = format(ref.forest, dig = 2), cex.axis = 0.8,
+     at = rank(ref.forest))
> abline(h = 0, col = "darkgreen")
> boxplot(residuals(hd.lme.3, type = "pearson", level = 1) ~ stage$Tree.ID,
+     ylab = "Résidus les plus internes", xlab = "Arbre", notch = T,
+     varwidth = T, at = rank(ref.tree))
> axis(3, labels = format(ref.tree, dig = 2), cex.axis = 0.8, at = rank(ref.tree))
> abline(h = 0, col = "darkgreen")
> plot(ref.forest, ref.var.forest, xlab = "Effet aléatoire forêt",
+     ylab = "Variance des résidus intra-forêt")
> abline(lm(ref.var.forest ~ ref.forest), col = "purple")
> plot(ref.tree, ref.var.tree, xlab = "Effet aléatoire arbre",
+     ylab = "Variance des résidus intra-arbre")
> abline(lm(ref.var.forest ~ ref.forest), col = "purple")
> par(opar)
```

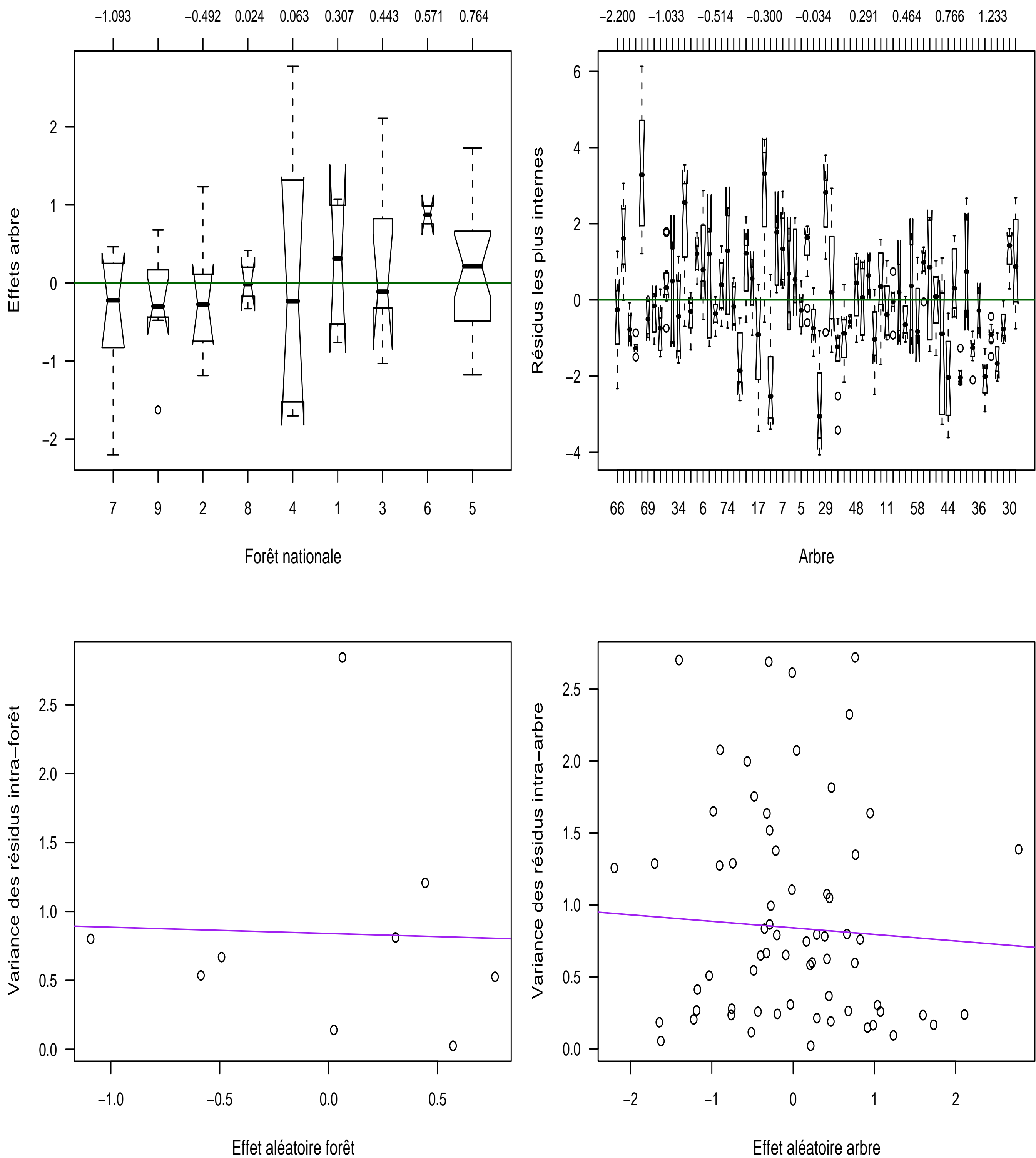


Figure 11.13 – Diagnostics d'effets aléatoires sélectionnés pour l'ajustement en effets mixtes du modèle hauteur et diamètre pour les données Stage.

Tout dans ces figures semblent bon, sauf pour les tracés des résidus et la corrélation des résidus intra-arbre, qui montrent un signal de force inacceptable. À ce moment-là on pourrait penser que la prochaine étape consiste à essayer d'ajuster une fonction d'auto-corrélation des résidus intra-arbre, mais le coude dans le tracé des résidus suggère qu'il semble plus intelligent de jeter un coup d'oeil dabord à un diagnostic différent.

Le tracé des prévisions agrémentées recouvre la modèle ajusté avec les données observées, à un niveau intra, optionnel. Il est construit en utilisant `xyplot()` des graphiques `lattice` et accepte des arguments qui sont utiles à cette fonction, pour aller plus loin dans la personnalisation. Cela permet de trier les arbres par forêt nationale, pour nous aider à extraire tous les effets de groupement.

```
> trees.in.forests <- aggregate(x = list(measures = stage$height.m),
+     by = list(tree = stage$Tree.ID, forest = stage$Forest.ID),
```

```
+     FUN = length)
> panel.order <- rank(as.numeric(as.character(trees.in.forests$tree)))
> print(plot(augPred(hd.lme.3), index.cond = list(panel.order)))
```

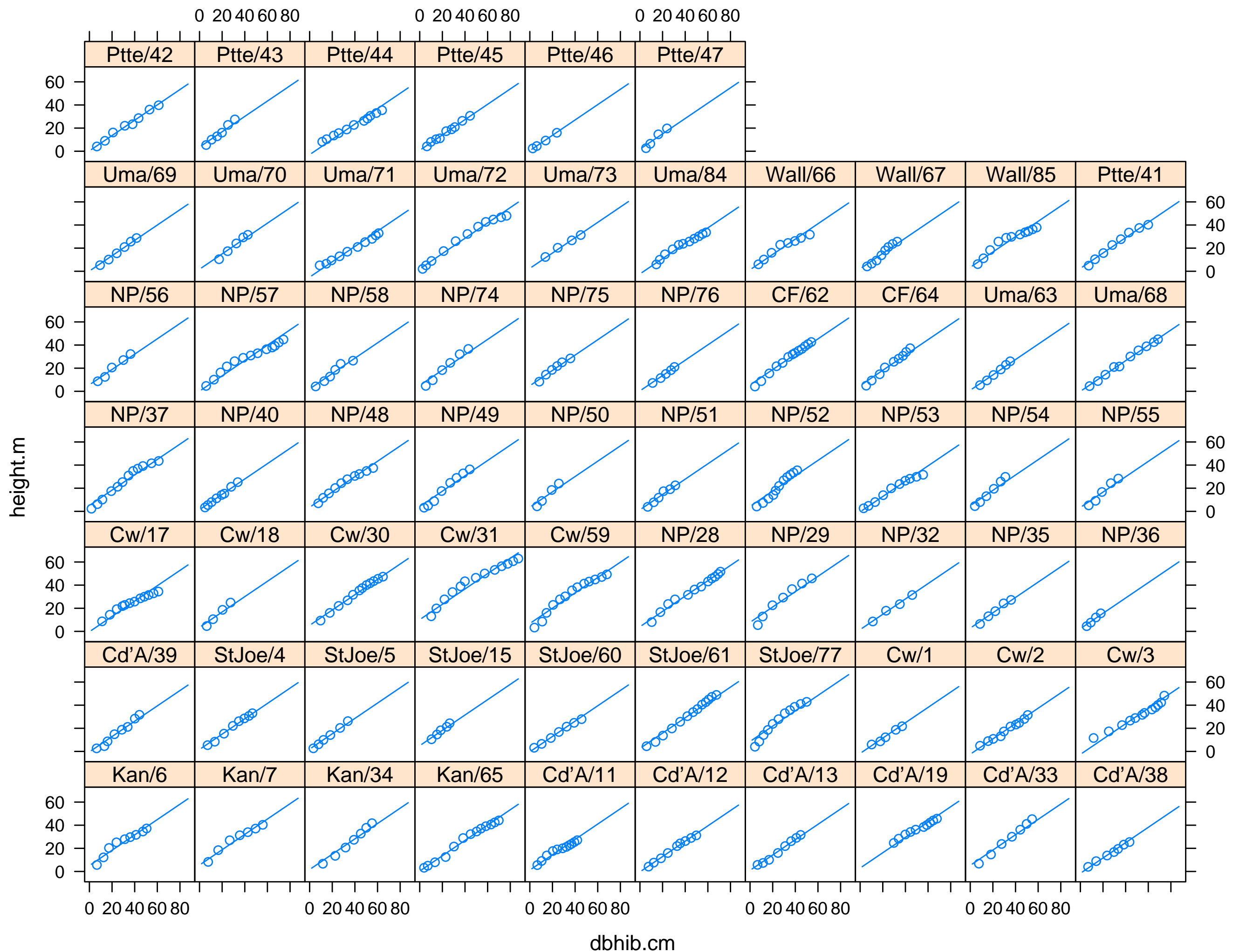


Figure 11.14 – Hauteur en fonction du diamètre par arbre, agrémenté des droites de prédiction.

Le tracé de prédiction agrémenté (figure 11.14) montre que quelques arbres présentent une courbure dans la relation hauteur – diamètre que le modèle ne peut extraire, alors que les autres semblent joliment linéaires. Cela montre aussi que l'omission d'une pente aléatoire apparaît comme problématique.

À ce point, nous avons plusieurs options, chacune conduisant potentiellement à différentes solutions de notre problème, ou, plus probablement, à plusieurs autres approches, et ainsi de suite. Comment nous procéderons dépend de notre objectif. Nous pouvons :

1. ajouter un effet fixe quadratique ;
2. ajouter un effet aléatoire quadratique ;
3. ajouter des effets aléatoires et fixes quadratiques ;
4. corriger le modèle en incluant une corrélation intra-arbre et
5. passer à des modèles à effets mixtes non linéaires et utiliser une forme fonctionnelle plus appropriée.

Puisque nous ne pensons pas que la véritable relation entre la hauteur et le diamètre pourrait raisonnablement être une ligne droite, nous allons ajouter un effet fixe et aléatoire quadratique de diamètre, par arbre, et voir comment les choses se passent. Pour commencer cela fera augmenter le nombre de diagnostics graphiques que nous voulons examiner à environ 22 ! Nous ne montrerons ici qu'un échantillon.

```
> hd.lme.4 <- lme(height.m ~ dbhib.cm + I(dbhib.cm^2), random = ~dbhib.cm +
+     I(dbhib.cm^2) | Tree.ID, data = stage)
```

```
> opar <- par(mfrow = c(1, 3), mar = c(4, 4, 3, 1), las = 1, cex.axis = 0.9)
> plot(fitted(hd.lme.4, level = 0), stage$height.m, xlab = "Valeurs ajustées",
+     ylab = "Valeurs observées", main = "Structure du modèle (I)")
> abline(0, 1, col = "gray")
> scatter.smooth(fitted(hd.lme.4), residuals(hd.lme.4, type = "pearson"),
+     main = "Structure du modèle (II)", xlab = "Valeurs ajustées",
+     ylab = "Résidus les plus internes")
> abline(0, 0, col = "gray")
> acf.resid <- ACF(hd.lme.4, resType = "n")
> plot(acf.resid$lag[acf.resid$lag < 10.5], acf.resid$ACF[acf.resid$lag <
+     10.5], type = "b", main = "Autocorrélation", xlab = "Décalage",
+     ylab = "Corrélation")
> stdv <- qnorm(1 - 0.01/2)/sqrt(attr(acf.resid, "n.used"))
> lines(acf.resid$lag[acf.resid$lag < 10.5], stdv[acf.resid$lag <
+     10.5], col = "darkgray")
> lines(acf.resid$lag[acf.resid$lag < 10.5], -stdv[acf.resid$lag <
+     10.5], col = "darkgray")
> abline(0, 0, col = "gray")
> par(opar)
```

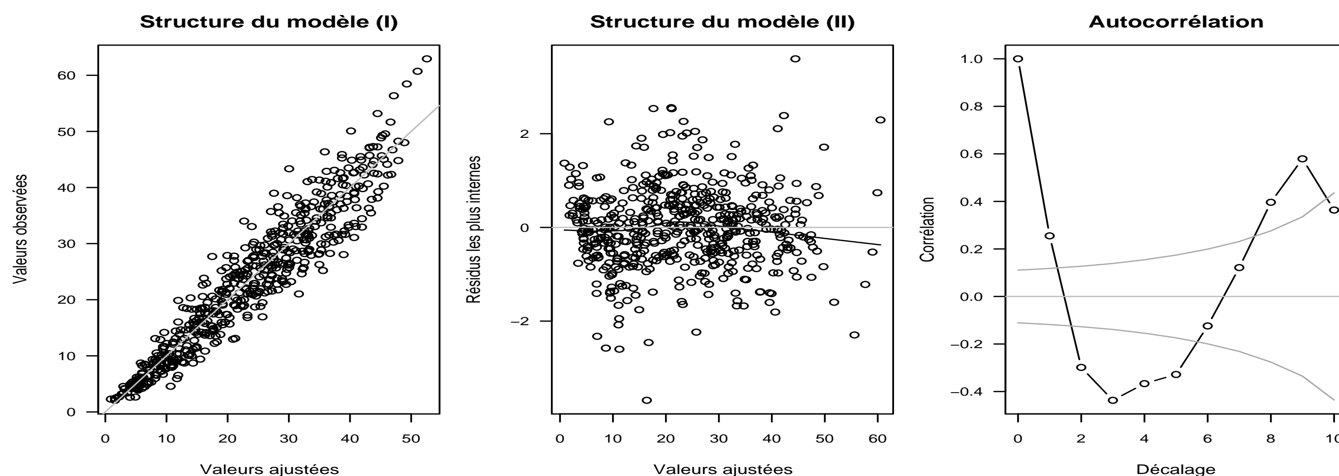


Figure 11.15 – Diagnostics sélectionnés pour l'ajustement en effets mixtes du modèle hauteur et diamètre pour les données Stage.

Cela a un peu amélioré le modèle, mais il semble que nous n'avons pas besoin d'inclure certains pour la comptabilité de corrélation intra-arbre. Pinheiro et Bates (2000)[15] détaille les options qui sont disponibles. Aussi, allons-nous utiliser `update()` parce que cela commence l'ajustement du modèle aux estimations ayant le plus récemment convergé, ce qui accélère considérablement l'ajustement. Enfin, nous devons utiliser un autre moteur d'ajustement, pour une plus grande stabilité.

```
> hd.lme.5 <- update(hd.lme.4, correlation = corCAR1(), control = lmeControl(opt = "optim"))
> opar <- par(mfrow = c(1, 3), mar = c(4, 4, 3, 1), las = 1, cex.axis = 0.9)
> plot(fitted(hd.lme.5, level = 0), stage$height.m, xlab = "Valeurs ajustées",
+     ylab = "Valeurs observées", main = "Structure du modèle (I)")
> abline(0, 1, col = "gray")
> scatter.smooth(fitted(hd.lme.5), residuals(hd.lme.5, type = "pearson"),
+     main = "Structure du modèle (II)", xlab = "Valeurs ajustées",
+     ylab = "Résidus les plus internes")
> abline(0, 0, col = "gray")
> acf.resid <- ACF(hd.lme.5, resType = "n")
> plot(acf.resid$lag[acf.resid$lag < 10.5], acf.resid$ACF[acf.resid$lag <
+     10.5], type = "b", main = "Auto-corrélation", xlab = "Décalage",
+     ylab = "Corrélation")
> stdv <- qnorm(1 - 0.01/2)/sqrt(attr(acf.resid, "n.used"))
> lines(acf.resid$lag[acf.resid$lag < 10.5], stdv[acf.resid$lag <
+     10.5], col = "darkgray")
> lines(acf.resid$lag[acf.resid$lag < 10.5], -stdv[acf.resid$lag <
```

```
+      10.5], col = "darkgray")
> abline(0, 0, col = "gray")
> par(opar)
```

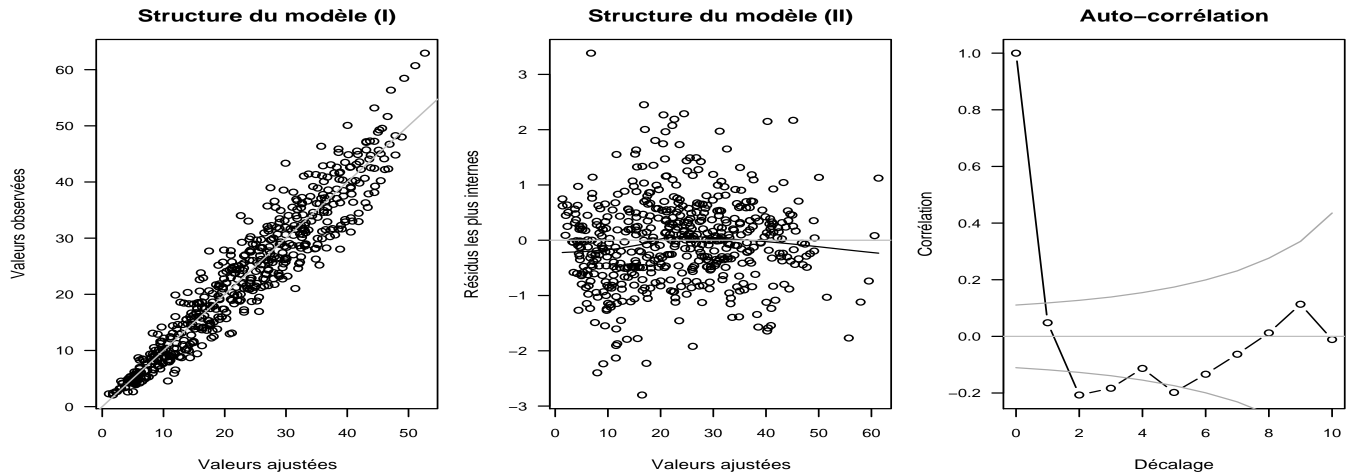


Figure 11.16 – Diagnostics sélectionnés pour l'ajustement en effets mixtes du modèle hauteur et diamètre pour les données Stage.

La corrélation est maintenant faible.

Un autre élément du modèle où nous avons plus de contrôle est la variance des effets aléatoires. Nous n'avons pas vu se lever de drapeaux rouges d'hétéroscédasticité pour les diagnostics de modèle, aussi n'avons-nous pas à nous inquiéter. Toutefois, de telles situations sont suffisamment courantes pour en faire un exemple intéressant.

Deux types d'hétéroscédasticité sont fréquents et digne de préoccupation : premièrement, que le variance de la variable réponse soit liée à la réponse et, deuxièmement, que la variance conditionnelle des observations ait varié dans une ou plusieurs strates. Une combinaison des deux conditions est également possible.

Nous pouvons détecter ces conditions par des diagrammes de dispersion des résidus conditionnels du type suivant. Le premier est un diagramme de dispersion des résidus de Pearson les plus internes fonction des valeurs stratifiées par type d'habitat. Le code pour créer ce graphique fait partie du paquetage `nlme`.

```
> plot(hd.lme.5, resid(.) ~ fitted(.) | HabType.ID,
+ layout = c(1, 5))
```

Le deuxième est un tracé Quantile-Quantile des résidus de Pearson les plus internes en fonction de la distribution normale, stratifiée par type d'habitat. Ce code est fourni par le paquetage `lattice`, et nous avons trouvé un motif sous `?qqmath`.

```
> qqmath(~resid(hd.lme.5) | stage$HabType.ID, prepanel = prepanel.qqmathline,
+     panel = function(x, ...) {
+         panel.qqmathline(x, distribution = qnorm)
+         panel.qqmath(x, ...)
+     })
```

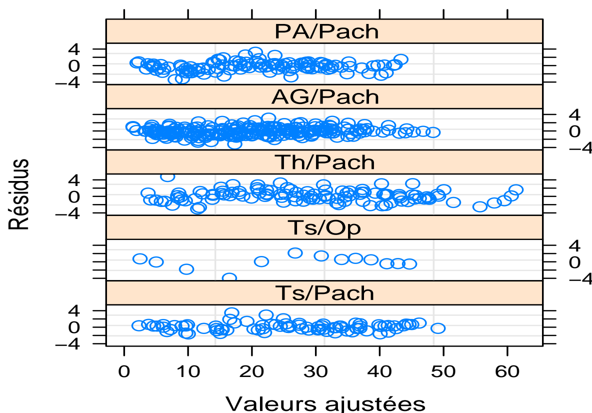


Figure 11.17 – Résidus de Pearson les plus internes en fonction des valeur ajustées par type d'habitat.

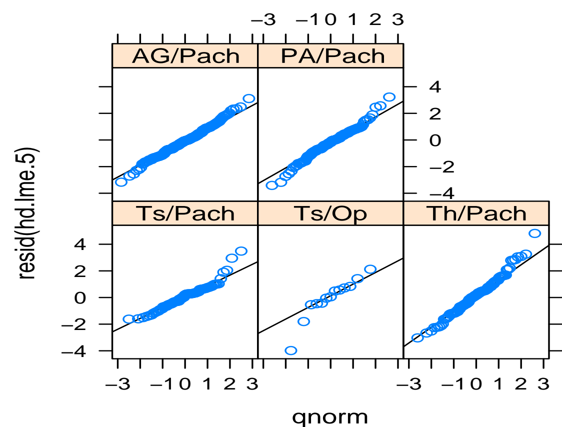


Figure 11.18 – quantiles des résidus de Pearson les plus internes en fonction d'une distribution normale par type d'habitat.

Il semble un peu évident dans les deux figures 11.17 et 11.18 de penser que le modèle de variance est inadéquat.

Dès que le modèle de variance semble inadéquat, on peut utiliser l'argument `weights` dans un appel à `update` avec une des démarches suivantes :

```
> all.betas <- t(all.betas[, -1])
> len.all <- length(unlist(hd.lme.1$coefficients))
> len.fixed <- length(hd.lme.1$coefficients$fixed)
> len.ran <- length(hd.lme.1$coefficients$random$Forest.ID)
> opar <- par(mfrow = c(len.all, 1), oma = c(2, 0, 1, 0), mar = c(0,
+     4, 0, 0), las = 1)
> for (i in 1:len.fixed) {
+     plot(all.betas[, i], type = "l", axes = F, xlab = "", ylab = "")
+     text(length(all.betas[, i]) - 1, max(all.betas[, i], na.rm = T),
+         names(unlist(hd.lme.1$coefficients))[i], adj = c(1, 1),
+         col = "red")
+     axis(2)
+     box()
+ }
> for (i in (len.fixed + 1):(len.all)) {
+     plot(all.betas[, i], type = "l", axes = F, xlab = "", ylab = "")
+     text(length(all.betas[, i]) - 1, max(all.betas[, i], na.rm = T),
+         names(unlist(hd.lme.1$coefficients))[i], adj = c(1, 1),
+         col = "red")
+     axis(2)
+     box()
+ }
> axis(1)
> par(opar)
```

- `weights = varIdent(form = 1 | HabType.ID)` Cette option permettrait aux observations dans chaque type d'habitat d'avoir sa propre variance.
- `weights = varPower()` Cette option ajusterait une fonction puissance pour la relation entre variance et moyenne prédite, et en estimerait l'exposant.
- `weights = varPower(form = dhib.cm | HabType.ID)` Cette option ajusterait une fonction puissance pour la relation entre variance et le diamètre uniquement dans chaque type d'habitat, et en estimerait l'exposant.
- `weights = varConstPower()` Cette option ajusterait une fonction puissance avec une constante pour la relation entre variance et moyenne prédite, et en estimerait l'exposant et la constante.

D'autres options sont disponibles ; la fonction est totalement documentée dans Pinheiro et Bates (2000)[15].

Bon, acceptons le modèle tel qu'il se présente pour le moment. C'est le modèle de base, ainsi il fournit des prévisions de hauteur à partir du diamètre, et satisfait aux hypothèses de régression. D'autres options peuvent justifier plus tard qu'il y a un meilleur ajustement, par exemple, ce peut être le fait d'inclure le type d'habitat ou l'âge dans le modèle pour éviter notre utilisation du terme diamètre quadratique. Que cela en fasse oui ou non un meilleur modèle en termes d'applications réelles varira !

```
> plot(augPred(hd.lme.5), index.cond = list(panel.order))
```

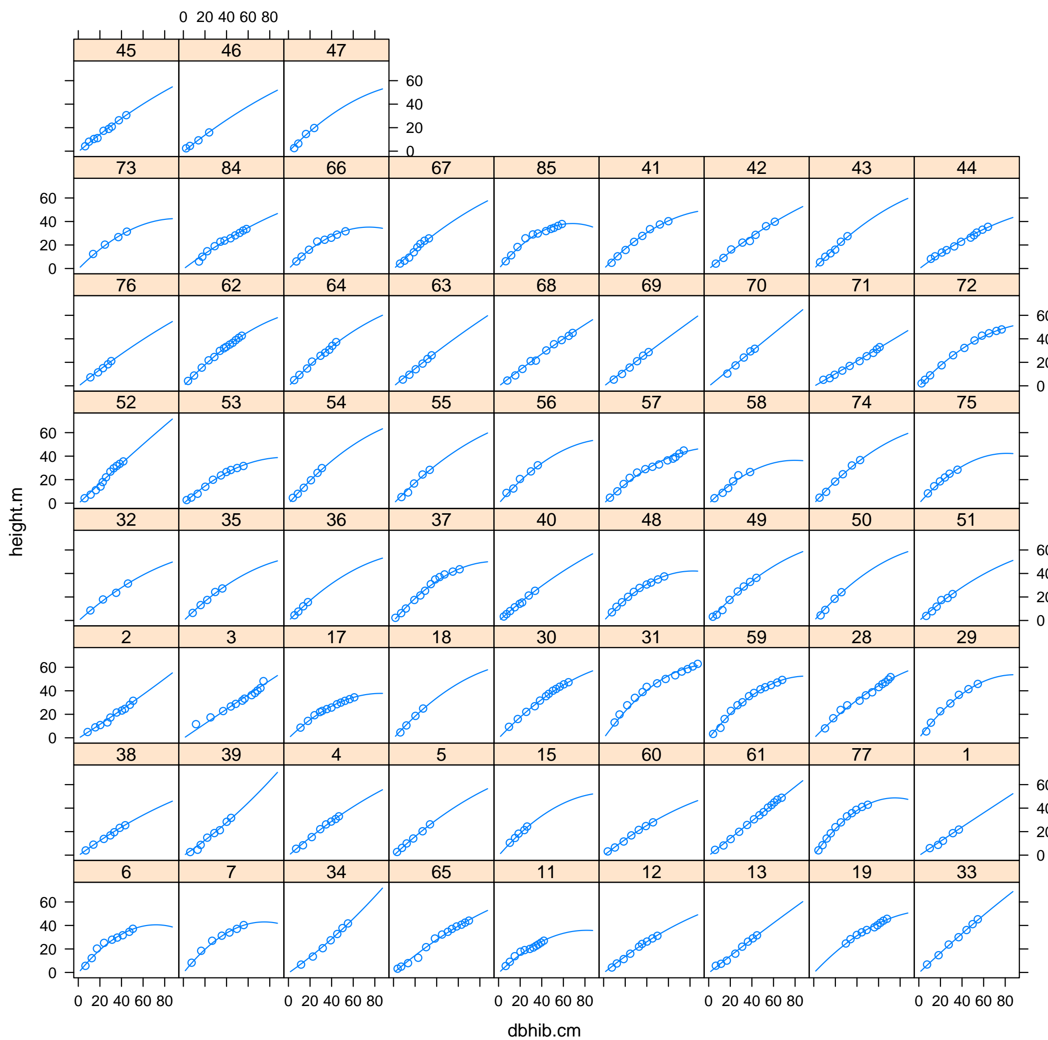


Figure 11.19 – Hauteur en fonction du diamètre par arbre, agrémenté des courbes prédites.

```
> summary(hd.lme.5)

Linear mixed-effects model fit by REML
 Data: stage
       AIC      BIC    logLik
  1945.521 1992.708 -961.7604

Random effects:
 Formula: ~dbhib.cm + I(dbhib.cm^2) | Tree.ID
 Structure: General positive-definite, Log-Cholesky parametrization
              StdDev       Corr
(Intercept)   0.0007992449 (Intr) dbhb.c
dbhib.cm      0.1844016123 -0.243
I(dbhib.cm^2) 0.0030927949 -0.175 -0.817
Residual      1.4223449957

Correlation Structure: Continuous AR(1)
 Formula: ~1 | Tree.ID
 Parameter estimate(s):
      Phi
0.6660391
Fixed effects: height.m ~ dbhib.cm + I(dbhib.cm^2)
                   Value  Std.Error  DF   t-value p-value
(Intercept)   -0.4959656 0.25444240 474 -1.949226  0.0519
dbhib.cm       0.8918030 0.02985028 474 29.875871  0.0000
I(dbhib.cm^2) -0.0032310 0.00052633 474 -6.138857  0.0000
```

```
 Correlation:
              (Intr) dbhb.c
dbhib.cm      -0.514
I(dbhib.cm^2)  0.417 -0.867

Standardized Within-Group Residuals:
         Min           Q1          Med           Q3          Max
-2.799939334 -0.482647581 -0.008763721  0.414566863  3.384428017

Number of Observations: 542
Number of Groups: 66
```

### 11.4.2 Extensions au modèle

Nous pouvons essayer d'étendre le modèle de référence pour améliorer ses performances, sur la base de nos connaissances du système. Par exemple, il serait peut-être vrai que l'âge de l'arbre modifie son diamètre – en relation avec la hauteur d'une manière qui n'a pas été prise en compte dans le modèle. Nous pouvons tester formellement cette affirmation, en utilisant la fonction `anova`, nous pouvons l'examiner graphiquement, en utilisant un tracé à variable ajoutée, ou nous pouvons essayer d'adapter le modèle avec le terme inclus et voir quel effet cela a sur la variation résiduelle.

Un tracé à variable ajoutée est une récapitulation graphique de la quantité de variation qui est expliquée de façon unique par une variable prédictive. Il peut être construit en **R** comme suit. Ici, nous devons décider quel niveau des résidus choisir, car il en existe plusieurs. Nous adoptons les résidus les plus externes.

```
> age.lme.1 <- lme(Age ~ dbhib.cm, random = ~1 | Forest.ID/Tree.ID,
+     data = stage)
> res.Age <- residuals(age.lme.1, level = 0)
> res.HD <- residuals(hd.lme.5, level = 0)
> scatter.smooth(res.Age, res.HD, xlab = "Variation avec l'âge seul",
+     ylab = "Variation de hauteur avec tout sauf l'âge")
```

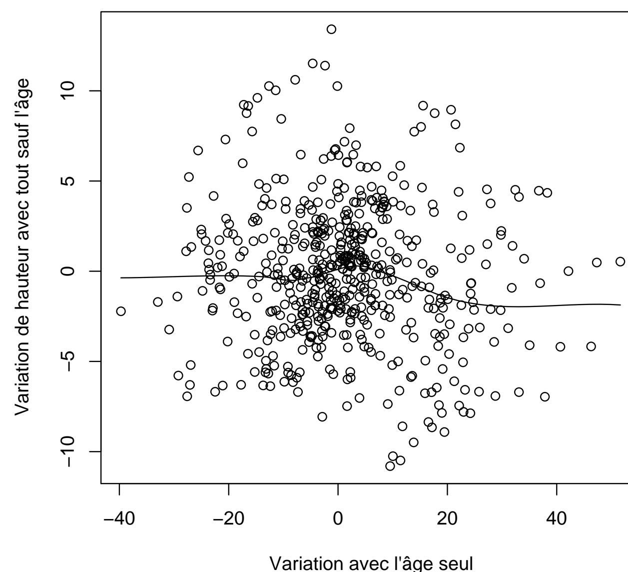


Figure 11.20 – Un tracé à variable ajoutée pour l'âge par rapport à la hauteur en fonction du diamètre.

Afin de déterminer si nous serions mieux servis par l'ajout au modèle du type d'habitat, nous pouvons construire une récapitulation graphique, ainsi :

```
> xyplot(stage$height.m ~ fitted(hd.lme.5, level = 0) | HabType.ID,
+     xlab="Hauteur prédite (m)",
```

```
+     ylab="Hauteur observée (m)",
+     data = stage, panel = function(x, y, subscripts) {
+         panel.xyplot(x, y)
+         panel.abline(0, 1)
+         panel.abline(lm(y ~ x), lty = 3)
+     }
+ )
```

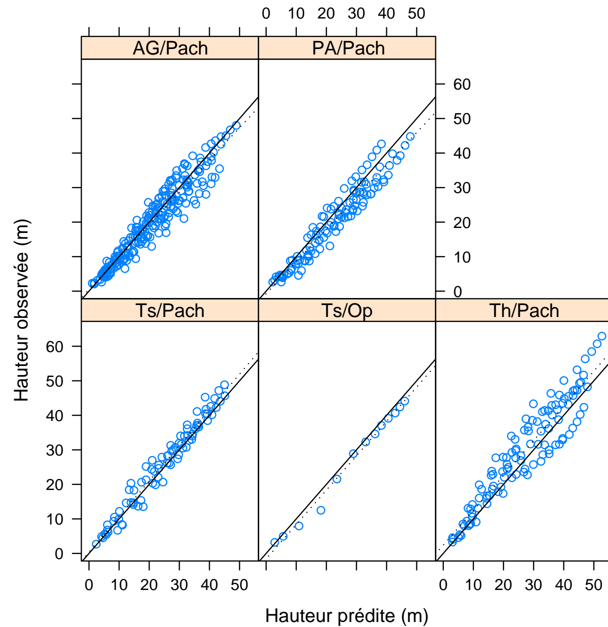


Figure 11.21 – Tracé de la hauteur prédite en fonctioin de la hauteur observée, par type d'habitat. La droite est la première bissectrice comme prédite par le modèle. Celle en pointillé est la droite MCO de meilleur ajustement intra-habitat.

## 11.5 Le modèle

Examinons notre modèle final :

$$\begin{aligned} y_{ijk} &= \beta_0 + b_{0i} + b_{0j} && (11.17)\\ &+(\beta_1 + b_{1i} + b_{1ij}) \times x_{ijk} && (11.18)\\ &+(\beta_2 + b_{2i} + b_{2ij}) \times x_{ijk}^2 && (11.19)\\ &+\varepsilon_{ijk} && (11.20) \end{aligned}$$

En forme matricielle c'est toujours :

$$\begin{aligned} \mathbf{Y} &= \mathbf{X}\boldsymbol{\beta} + \mathbf{Z}\mathbf{b} + \boldsymbol{\epsilon} \\ \mathbf{b} &\sim \mathcal{N}(\mathbf{0}, \mathbf{D}) \\ \boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \end{aligned}$$

En voici la structure :
- $\mathbf{Y}$ est le vecteur de mesures de hauteur. il y a 542 observations.
- $\mathbf{X}$ est une matrice $3 \times 542$ de 1, de diamètres et de diamètres carrés.
- $\boldsymbol{\beta}$ est un vecteur de taille trois : il contient une ordonnée à l'origine, une pente pour le terme de diamètre linéaire et une pente pour le terme de diamètre quadratique.
- $\mathbf{Z}$ est une matrice $225 \times 542$ brute. Voir plus bas.
- $\mathbf{b}$ est est un vecteur d'ordonnées à l'origine, de pentes pour les diamètre et diamètre carré de chaque forêt. Il aura $27 + 198 = 225$ éléments. Voir plus bas. Les prédictions peuvent être obtenues avec `ranef(hd.lme.5)`.
- $\mathbf{D}$ sera une matrice diagonale par bloc comprenant 9 matrices $3 \times 3$ identiques, suivies de 66 matrices $3 \times 3$ identiques. Chaque matrice exprimera la covariance entre les 3 effets aléatoires intra-forêt ou intra-arbre. Voir plus bas.
- $\mathbf{R}$ sera maintenant une matrice $542 \times 542$ symétrique pour laquelle les éléments hors diagonale sont des 0 inter-arbres et une fonction géométrique inter-instants de mesure, intra-arbre.

### 11.5.1 Z

Le seul rôle est de Z est d'affecter les effets aléatoires à l'élément pertinent. Ce peut être un peu compliqué. Notre **Z** peut être divisé en deux sections indépendantes, une matrice $27 \times 542$, $\mathbf{Z_f}$, associée aux niveaux d'effet forêt, et une matrice $198 \times 542$, $\mathbf{Z_a}$, associée aux niveaux d'effet arbre. Dans la nomenclature des matrices :

$$\mathbf{Z} = [\mathbf{Z_f}|\mathbf{Z_a}] \tag{11.21}$$

Maintenant, $\mathbf{Z_f}$ affecte les ordonnées à l'origine aléatoires et deux pentes à chaque observation de chaque forêt. Il y a 9 forêts, de sorte que toute ligne de $\mathbf{Z_f}$ contiendra 24 zéros, un 1, et les valeurs correspondant à *dbhib* et *dbhib*$^2$. Par exemple, pour la ligne correspondant à la mesure 4 sur l'arbre 2 de la forêt 5, nous aurons :

$$Z_f = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, d_{524}, d_{524}^2, 0, 0, 0, \ldots) \tag{11.22}$$

De même, $\mathbf{Z_a}$ affecte les ordonnées à l'origine aléatoires et deux pentes à chaque observation de chaque arbre. Il y a 66 arbres, de sorte que toute ligne de $\mathbf{Z_f}$ contiendra 195 zéros, un 1, et les valeurs correspondant à *dbhib* et *dbhib*$^2$. Elle aura le même aspect fondamental que ci-dessus.

### 11.5.2 b

Le but de **b** est de contenir tous les effets aléatoires prédits. Ainsi, il sera long de 225 unités, ce qui correspond à 3 pour chaque niveau de forêt (une ordonnée à l'origine, une pente de diamètre, et une pente de diamètre au carré) et 3 pour chaque niveau d'arbre (une ordonnée à l'origine, une pente de diamètre, et une pente de diamètre au carré).

$$\mathbf{b} = (b_{f10}, b_{f1d}, b_{f1d2}, b_{f20}, b_{f2d}, b_{f2d2}, \ldots, b_{a10}, b_{a1d}, b_{a1d2}, b_{a20}, b_{a2d}, b_{a2d2}, \ldots)' \tag{11.23}$$

La combinaison des **b** et **Z** sert à affecter chaque effet aléatoire à l'unité et la mesure appropriées.

### 11.5.3 D

Enfin, **D** dicte les relations entre les différents effets aléatoires aux niveaux forêts et arbres. Nous avons supposé que les effets aléatoires seront indépendants entre les types d'habitats et les arbres. Donc, il n'y a que deux sous-matrices à cette matrice, appelé $\mathbf{D_f}$ et $\mathbf{D_a}$.

$$\mathbf{D_f} = \begin{pmatrix} \sigma^2_{bf0} & \sigma_{bf0d} & \sigma_{bf0d2} \\ \sigma_{bf0d} & \sigma^2_{bfd} & \sigma_{bfdd2} \\ \sigma_{bf0d2} & \sigma_{bfdd2} & \sigma^2_{bfd2} \end{pmatrix} \tag{11.24}$$

$$\mathbf{D_a} = \begin{pmatrix} \sigma^2_{ba0} & \sigma_{ba0d} & \sigma_{ba0d2} \\ \sigma_{ba0d} & \sigma^2_{bad} & \sigma_{badd2} \\ \sigma_{ba0d2} & \sigma_{badd2} & \sigma^2_{bad2} \end{pmatrix} \tag{11.25}$$

Ensuite la structure de **D** est simplement 9 répétitions de $\mathbf{D_f}$, placées sur la diagonale, suivies de 66 répétitions de $\mathbf{D_a}$, placées sur la même diagonale, t de zéros partout ailleurs.

## 11.6 Disputes

Nous avons observé plus haut que l'utilisation de l'argument `control` était un outil clef pour le modélisateur. Cet élément peut introduire un petit choc culturel. Provenant nous-mêmes de traditions d'ajustement de modèle pour lequel des solutions précises ont été facilement obtenues, et la convergence sans équivoque, il est surprenant, pour ne pas dire désolant, de constater que des algorithmes parfois s'arrêtent avant la convergence. Nous affichons probablement ainsi notre naïveté.

Les outils statistiques dont nous avons discuté dans ce chapitre sont trop compliqués pour admettre des solutions exactes. En conséquence, nous devons essayer de maximiser la vraisemblance, par exemple, par des moyens itératifs. Il est nécessaire et correct que les auteurs du code auquel nous avons eu recours aient mis des contrôles, pour mettre fin au code dans des situations ininterrompues qu'ils jugent non rentables.

En tout état de cause, l'amère expérience et l'expérimentation impitoyable nous ont appris que les auteurs du code n'ont pas nécessairement exactement notre problème à l'esprit quand ils choisissent les paramètres par défaut de leurs logiciels. Dans de tels cas, il est nécessaire de remonter nos manches et plonger les bras dans les organes de notre analyse. La plupart des outils d'ajustement que nous utilisons ont des arguments de contrôle qui feront un rapport ou modifieront le processus de modélisation de l'ajustement. Faire des expériences avec conduit souvent à des configurations de modèle fiables qui conviennent.

En bref, ne pas hésiter à expérimenter. Tout ou partie des stratégies suivantes pourraient être nécessaires pour obtenir un bon ajustement de votre modèle à vos données.

### 11.6.1 Contrôler

Afin d'être mieux informés sur les progrès de l'ajustement de modèle, nous utilisons l'argument `msVerbose`. Il fournit une brève description des progrès sur l'ajustement de modèle. Il mettra également en évidence des problèmes en cours de route, ce qui aide l'utilisateur à décider de la meilleure chose à faire.

### 11.6.2 Toucher à tout

Cette stratégie implique l'adaptation de l'outil approprié.

Si le modèle ne parvient pas à converger, alors souvent tout ce qui est nécessaire est une augmentation du nombre d'itérations admissibles. Les algorithmes d'ajustement de modèle à effets mixtes dans `lme` utilisent un système d'optimisation hybride qui commence par l'algorithme EM et passe ensuite à Newton-Raphson (Pinheiro et Bates[15], 2000, p. 80). Le dernier algorithme est mis en œuvre avec deux boucles, nous avons donc trois cpossibilités d'itération. Nous avons constaté qu'augmenter les deux arguments `maxIter` et `msMaxIter` est une stratégie utile. Si nous restons patient, nous les augmenterons à environ 10000, et de surveillerons le processus pour voir si l'algorithme reste toujours en recherche. Nous avons parfois vu le compte d'itération dépasser 8000 modèles que pour des modèles qui ont par la suite convergé.

Nous avons également eu du succès en changeant l'algorithme d'optimisation. C'est-à-dire que les modèles qui n'ont pas réussi à converger avec `nlminb`, attraper dans une singulière convergence et converger avec succès en utilisant `Nelder-Mead` dans `optim`. Le défaut est d'utiliser `nlminb`, mais il peut être mieux de passer à `Optim`, et dans `Optim`, le choisir entre `Nelder-Mead`, `BFGS`, et `L-BFGS-B`. Chacun de ces algorithmes a des propriétés différentes, et différentes forces et faiblesses. Tout peut conduire de manière satisfaisante à une solution plus fiable.

### 11.6.3 Modifier

Cette stratégie implique le changement de la relation entre le modèle et les données.

La fonction `update` correspond à un nouveau modèle en utilisant la sortie d'un ancien modèle au départ. C'est un moyen facile de fixer au début les estimations des paramètres, et devrait avoir en commun l'utilisation de la construction itérative de modèles dans tous les cas, en raison de son efficacité. Essayez de retirer du modèle les composants qui compliquent les choses, un petit ajustement, un plus simple modèle, et ensuite utiliser `update` pour ajuster le modèle complet.

Autrement, un certain nombre de composantes du modèle permet la spécification d'un point de départ. Par exemple, si nous donnons un nombre approprié à la fonction `corAR1` l'algorithme utilise ce nombre comme point de départ. Spécifier cette valeur peut aider l'algorithme à converger rapidement, et parfois, pas du tout. Faire des expériences avec des sous-ensembles du modèle complet pour essayer de trouver des points de départ peut être rentable, par exemple si l'on a un modèle de corrélation et un modèle de variance.

Nous pouvons également réfléchir à la façon dont les éléments dans les données sont susceptibles d'être en interaction avec le modèle. Est-ce que le jeu de données est déséquilibré, y a t-il des résultats aberrants, ou est-il trop petit ? Une de ces conditions peut entraîner des problèmes d'adaptation des algorithmes. L'examen des données avant d'ajuster le modèle est une pratique courante. Soyez prêt à supprimer *temporairement* les points de données ou augmenter les parties sous-représentées , afin de fournir à la fonction `update` un ensemble raisonnable de valeurs de départ.

### 11.6.4 Compromis

Parfois, un modèle contient une hiérarchie complexe d'effets aléatoires. Il vaut mieux se demander si oui ou non une telle profondeur est nécessaire et si un modèle superficiellement plus complexe, mais en fait plus simple, pourrait suffire. L'étude de cas de ce chapitre est un bon exemple : bien que l'ajustement du modèle soit bon en permettant à chaque arbre d'avoir une pente aléatoire, il n'était pas nécessaire de permettre à chaque forêt nationale d'avoir une pente aléatoire. Inclure une pente pour chaque forêt rend le modèle inutilement compliqué, et rend également l'ajustement du modèle beaucoup plus difficile. En revanche, spécifier un petit modèle était moins élégant.

Enfin, parfois, peu importe ce que nous avons comme exigences, un modèle convergera pas. Il est un point dans chaque analyse où nous devons décider de réduire nos pertes et de partir directement avec le modèle que nous avons. Si nous savons que le modèle a des lacunes, il est de notre responsabilité d'attirer l'attention sur ces lacunes. Par exemple, si nous sommes convaincus qu'il y a une auto-corrélation en série dans nos résidus, mais pas parvenir à un ajustement raisonnable à l'aide des ressources disponibles, alors fournir un tracé diagnostic de cette auto-corrélation est indispensable. En outre, il est important de faire des observations sur l'effet probable d'une lacune de modèle sur l'inférence et la prédiction. Si nous avons la chance d'être en mesure d'ajuster un plus simple modèle qui ne comprend pas d'auto-corrélation, par exemple, on peut démontrer quel effet l'inclusion de cette partie du modèle a sur nos conclusions. Nous le ferions en ajustant trois modèles : le modèle complexe, le modèle simple avec l'auto-corrélation,

et le modèle simple sans l'auto-corrélation. Si la différence entre ces deux derniers modèles est modeste, alors nous avons un modeste degré de preuve indirecte que peut-être nos conclusions seront robustes à mauvaise spécification du modèle complexe. Ce n'est pas idéal,mais il faut être pragmatique.

## 11.7 Annexe : les diagnostics Leave-One-Out

Une autre question importante est de savoir si il y a des valeurs aberrantes ou des points de grande influence. Dans un cas comme celui-ci, il est relativement facile de voir par les diagnostics qu'aucun point ne va probablement dominer l'ajustement de cette façon. Toutefois, un examen plus formel de la question est précieux. À ce jour, dans les revues, il y a peu de développement du problème des valeurs aberrantes et de détection d'influence. Schabenberger (2005)[19] donne un aperçu des nombreuses offres disponible dans SAS, dont aucune n'est actuellement disponibles dans **R**. Demidenko et Stukel (2005)[7] fournissent aussi des solutions de rechange.

Le plus simple, dans le cas où un ajustement de modèle est relativement rapide, est de le réajuster en retirant chaque observation une à une, et de collecter les résultats dans un vecteur pour une analyse plus approfondie. C'est au mieux géré avec la fonction `update()`.

```
> all.betas <- data.frame(labels = names(unlist(hd.lme.1$coefficients)))
> cook.0 <- cook.1 <- rep(NA, dim(stage.old)[1])
> p.sigma.0 <- length(hd.lme.1$coefficients$fixed) * var(residuals(hd.lme.1,
+     level = 0))
> p.sigma.1 <- length(hd.lme.1$coefficients$fixed) * var(residuals(hd.lme.1,
+     level = 1))
> for (i in 1:dim(stage.old)[1]) {
+     try({
+         hd.lme.n <- update(hd.lme.1, data = stage.old[-i, ])
+         new.betas <- data.frame(labels = names(unlist(hd.lme.n$coefficients)),
+             coef = unlist(hd.lme.n$coefficients))
+         names(new.betas)[2] <- paste("obs", i, sep = ".")
+         all.betas <- merge(all.betas, new.betas, all.x = TRUE)
+         cook.0[i] <- sum((predict(hd.lme.1, level = 0, newdata = stage.old) -
+             predict(hd.lme.n, level = 0, newdata = stage.old))^2)/p.sigma.0
+         cook.1[i] <- sum((predict(hd.lme.1, level = 1, newdata = stage.old) -
+             predict(hd.lme.n, level = 1, newdata = stage.old))^2)/p.sigma.1
+     })
+ }
```

On peut à présent examiner ces résultats sur des diagnostics graphiques (figures 11.22 et 11.23). Les distances de Cook présentées ici ne sont que des approximations.

```
> all.betas <- t(all.betas[, -1])
> len.all <- length(unlist(hd.lme.1$coefficients))
> len.fixed <- length(hd.lme.1$coefficients$fixed)
> len.ran <- length(hd.lme.1$coefficients$random$Forest.ID)
> opar <- par(mfrow = c(len.all, 1), oma = c(2, 0, 1, 0), mar = c(0,
+     4, 0, 0), las = 1)
> for (i in 1:len.fixed) {
+     plot(all.betas[, i], type = "l", axes = F, xlab = "", ylab = "")
+     text(length(all.betas[, i]) - 1, max(all.betas[, i], na.rm = T),
+         names(unlist(hd.lme.1$coefficients))[i], adj = c(1, 1),
+         col = "red")
+     axis(2)
+     box()
+ }
> for (i in (len.fixed + 1):(len.all)) {
+     plot(all.betas[, i], type = "l", axes = F, xlab = "", ylab = "")
+     text(length(all.betas[, i]) - 1, max(all.betas[, i], na.rm = T),
+         names(unlist(hd.lme.1$coefficients))[i], adj = c(1, 1),
+         col = "red")
+     axis(2)
+     box()
+ }
```

```
> axis(1)
> par(opar)
```

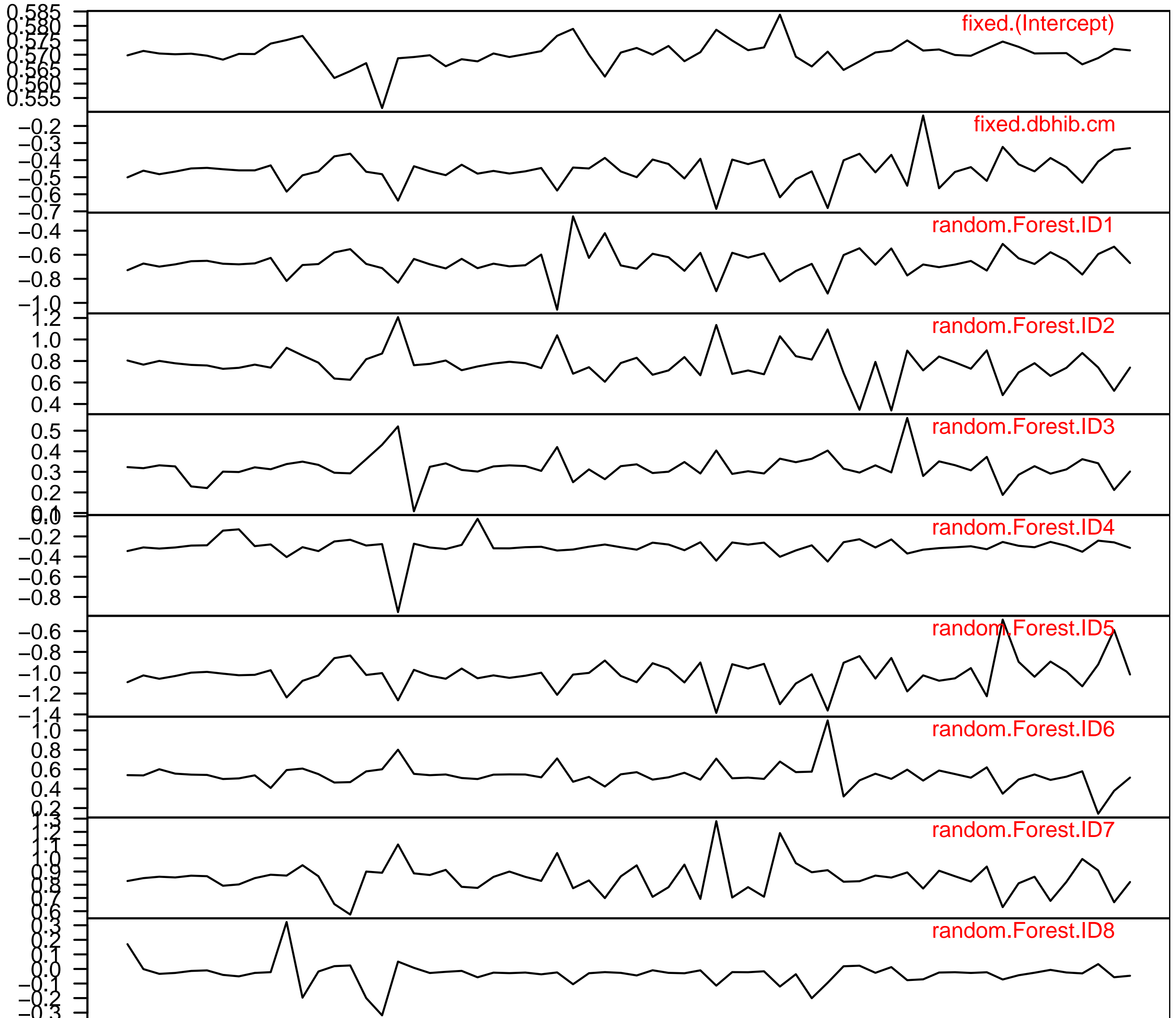


Figure 11.22 – Les estimations de paramètre pour les effets fixes et prédictions pour les effets aléatoires résultant de l'omission d'une observation.

```
> cook <- data.frame(id = stage.old$Tree.ID, fixed = cook.0, forest = cook.1)
> influential <- apply(cook[, 2:3], 1, max) > 1
> plot(cook$fixed, cook$forest, type = "n", xlab = "Seuls effets fixes (les plus externes)",
+     ylab = "Effets fixes et aléatoires (les plus internes)")
> points(cook$fixed[!influential], cook$forest[!influential])
> if (sum(influential) > 0) text(cook$fixed[influential], cook$forest[influential],
+     cook$id[influential], col = "red", cex = 0.85)
```

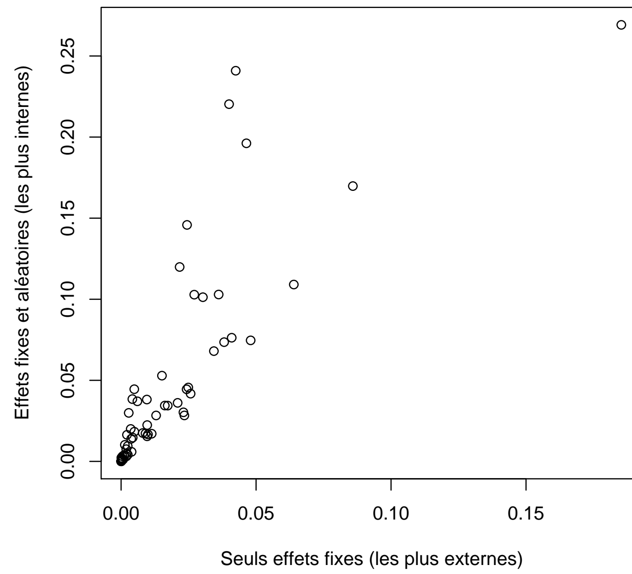


Figure 11.23 – Distances de Cook pour les résidus les plus externes et les plus internes. Les valeurs plus grandes que 1 apparaissent en rouge et sont identifiées par le numéro d'arbre. L'observation correspondante mérite une examination plus aiguë.

Et qu'en est-il de retirer les forêts entières ? On peut calculer les effets de façon similaire.

# Chapitre 12

# Modèle non linéaire

Nous allons maintenant passer un peu de temps à explorer les outils qui seront utiles lorsque, pour quelque raison que ce soit, le monde n' a plus l'air linéaire.

## 12.1 Les modèles non linéaires

Cette section se concentre sur les outils **R** qui peuvent être utilisés pour l'ajustement de régression non-linéaire, sous diverses formes. Comme par le passé, je ne présentent pas de théorie car elle a été traitée plus efficacement ailleurs. Les lectures recommandées sont : Ratkowsky (1983)[16], Gallant (1987)[9], Bates et Watts (1988)[1] et Seber et Wild (2003)[21]. Nos données sont des mesures de l'épinette de Norvège de von Guttenberg (1915), aimablement fournies par le professeur Boris Zeide.

Tout d'abord, lisons et examinons les données.

```
> gutten <- read.csv("../data/gutten.csv")
> str(gutten)
```

```
'data.frame':        1287 obs. of  8 variables:
 $ Site    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Location: int  1 1 1 1 1 1 1 1 1 1 ...
 $ Tree    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Age.base: int  10 20 30 40 50 60 70 80 90 100 ...
 $ Height  : num  1.2 4.2 9.3 14.9 19.7 23 25.8 27.4 28.8 30 ...
 $ Diameter: num  NA 4.6 10.2 14.9 18.3 20.7 22.6 24.1 25.5 26.5 ...
 $ Volume  : num  0.3 5 38 123 263 400 555 688 820 928 ...
 $ Age.bh  : num     NA  9.67 19.67 29.67 39.67 ...
```

Contrôlons les données manquantes :

```
> colSums(is.na(gutten))
```

```
    Site Location     Tree Age.base   Height Diameter   Volume   Age.bh
       0        0        0        0        0       87        6       87
```

Nous mettrons l'accent sur l'ajustement des modèles de croissance de diamètre, et nous devrions éliminer les colonnes et les lignes indésirables. Elles causeront des problèmes plus tard, autrement.

```
> gutten <- gutten[!is.na(gutten$Diameter), c("Site", "Location",
+     "Tree", "Diameter", "Age.bh")]
> names(gutten) <- c("site", "location", "tree", "dbh.cm", "age.bh")
> gutten$site <- factor(gutten$site)
> gutten$location <- factor(gutten$location)
```

Nous devons aussi construire un unique identificateur pour `site/location/tree`.

```
> gutten$tree.ID <- interaction(gutten$site, gutten$location, gutten$tree)
```

Le paquetage `lattice` nous fournit un moyen aisé de tracer les données (cf. figure 12.1. Noter que ce ne sont pas des arbres particulièrement grands !

```
> require(lattice)
> xyplot(dbh.cm ~ age.bh | tree.ID, type = "l", data = gutten)
```

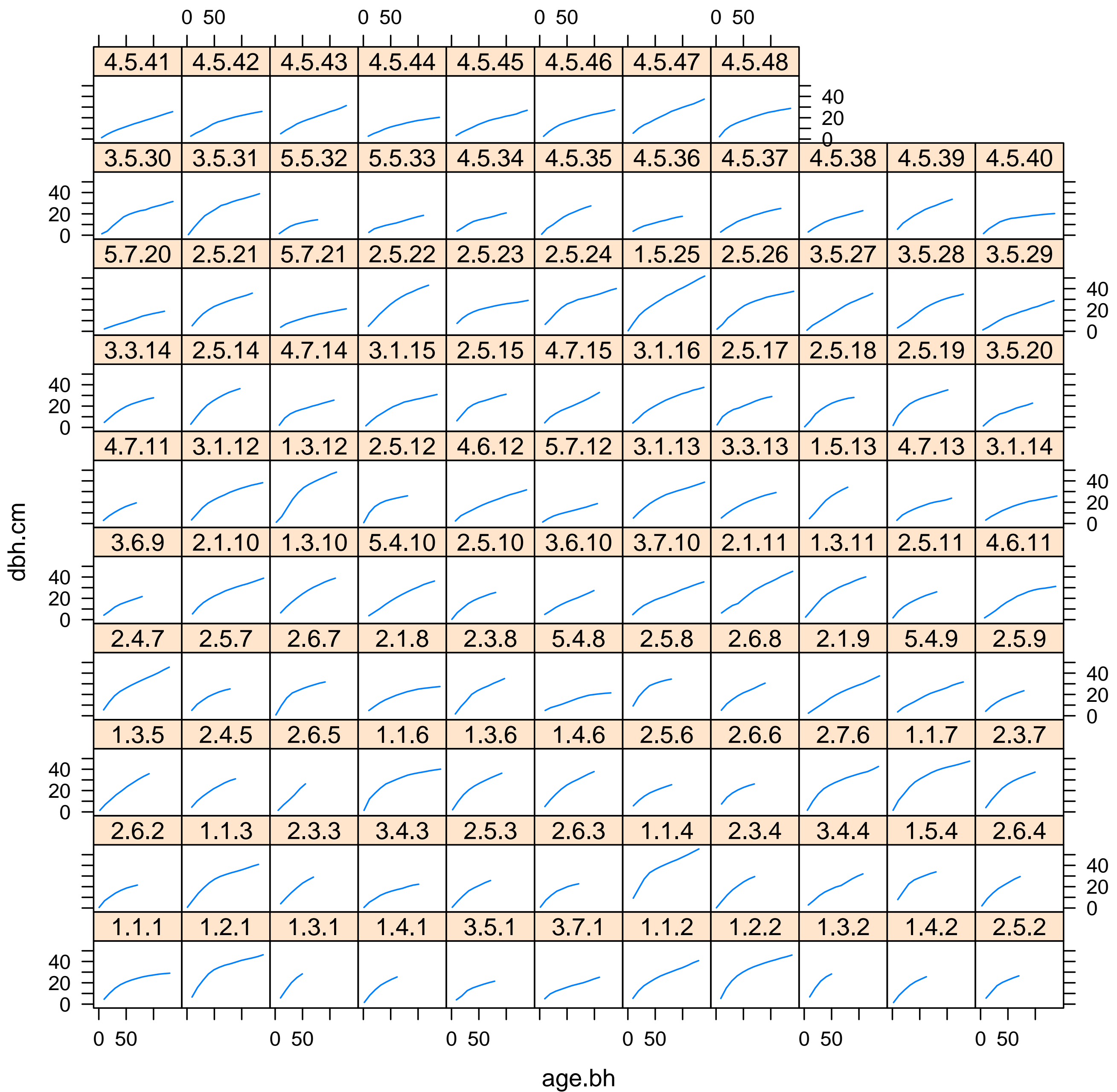


Figure 12.1 – Données diamètres de Guttenberg.

Un regard sur la figure 12.1 nous laisse à penser d'utiliser un modèle non linéaire passant par l'origine avec une asymptote. Passer par l'origine est un point de départ raisonnable car, par définition,`dbh` à hauteur de poitrine est nul jusqu'à ce que l'âge soit supérieur à 0. Cela peut s'avérer une horrible contrainte cependant. Le modèle est alors :

$$y_i = \phi_1 \times (1 - e^{-e^{\phi_2} x}) + \varepsilon_{it} \tag{12.1}$$

où $\phi_1$ est l'asymptote fixée inconnue et $\phi_2$ le facteur fixe inconnu (cf. Pinheiro et Bates, 2000[15]) et $t_{0.5}$ le temps où l'arbre aura atteint la moitié de sa taille maximale, soit $t_{0.5} = \log 2/\exp(\phi_2)$.

**NB :** Cela peut sembler étrange que nous adoptions ce modèle pour lequel un des paramètres a une simple interprétation, alors que l'autre n'est qu'une fonction d'une quantité qui a une simple interprétation. Nous avons

choisi cette version parce que le paramétrage d'un modèle non-linéaire affecte intimement son ajustement [1]. Certains paramétrages sont très difficile à ajuster au plus près ; Schabenberger et Pierce (2002)[20] ont bien discuté et démontré ce point, et voir également Ratkowsky (1983)[16]. La partie compliquée de ce point, c'est que la qualité du paramétrage dépend également des données, ainsi une version d'un modèle qui fonctionne bien pour un jeu de données peut très mal fonctionner pour un autre.

Pour ajuster un modèle non-linéaire nous avons généralement besoin de construire notre propre fonction non-linéaire de modèle. Nous pouvons le faire simplement par la construction d'une fonction qui produit les prévisions adéquates à entrées données. Toutefois, l'ajustement du modèle se révèle être plus performant si nous pouvons aussi passer la dérivée première, voire même la dérivée seconde de la fonction. Écrire une fonction pour passer les prédictions et ses dérivées est simplifié par l'utilisation de la fonction `deriv()`.

**NB :** R fournit quelques modèles pré-packagés, que nous verrons plus tard.

Notre utilisation de `deriv()` requiert trois choses : une instruction avec la fonction, un vecteur avec les noms des paramètres pour lesquels des dérivées sont requises et le patron de l'appel de fonction.

```
> diameter.growth <- deriv(~asymptote * (1 - exp(-exp(scale) * x)),
+                          c("asymptote", "scale"),
+                          function(x, asymptote, scale) {})
```

### 12.1.1 Un seul arbre

Ayant écrit le modèle non linéaire comme une fonction, nous devrions l'essayer. Sélectionnons-en un :

```
> handy.tree <- gutten[gutten$tree.ID == "1.1.1", ]
```

À l'instar de nos chapitres précédents, nous avons aussi besoin de deviner les premières estimations des paramètres. C'est une douleur très modeste dans ce cas. Ici nous utiliserons la valeur la plus élevée comme une estimation de l'asymptote, et supposer que l'arbre atteint environ la moitié de son diamètre maximal à environ 30 ans.

```
> max(handy.tree$dbh.cm)

[1] 29

> log(log(2)/30)

[1] -3.76771

> handy.nls <- nls(dbh.cm ~ diameter.growth(age.bh, asymptote, scale),
+                  start = list(asymptote = 29, scale = -3.76771),
+                  data = handy.tree)
```

Nous regardons l'objet modèle de la façon habituelle.

```
> summary(handy.nls)

Formula: dbh.cm ~ diameter.growth(age.bh, asymptote, scale)

Parameters:
          Estimate Std. Error t value Pr(>|t|)
asymptote  31.0175     0.4182   74.16 3.33e-16 ***
scale      -3.8366     0.0344 -111.52  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.4507 on 11 degrees of freedom

Number of iterations to convergence: 4
Achieved convergence tolerance: 4.943e-07
```

1. Pardonnez-moi !

Outre le fait que nous ne pouvons pas garantir d'avoir trouvé un minimum global, ce sont les estimations aux moindres carrés. Nous devrions essayer une série d'autres points de départ pour être sûr que tout va bien.

Toutefois, il pourrait également être de intéressant[2] de faire une inférence sur ces estimations. Qu'est-ce qui est sous-jacent à la distribution de la population ? Nous pouvons en apprendre plus avec la fonction `profile()` qui donne un aperçu de l'espace paramètre sur lequel nos données ont été projetées. Idéalement, nous voulons que la distribution des estimations du paramètre soit approximativement normale ; pour que cela soit vrai la fonction objectif devrait être à peu près quadratique au voisinage du minimum désigné.

```
> opar <- par(mfrow = c(2, 2), mar = c(4, 4, 2, 1), las = 1)
> plot(fitted(handy.nls), handy.tree$dbh.cm, xlab = "Valeurs ajustées",
+     ylab = "Valeurs observées")
> abline(0, 1, col = "red")
> plot(fitted(handy.nls), residuals(handy.nls, type = "pearson"),
+     )
> abline(h = 0, col = "red")
> plot(profile(handy.nls), conf = 0.95)
> par(opar)
```

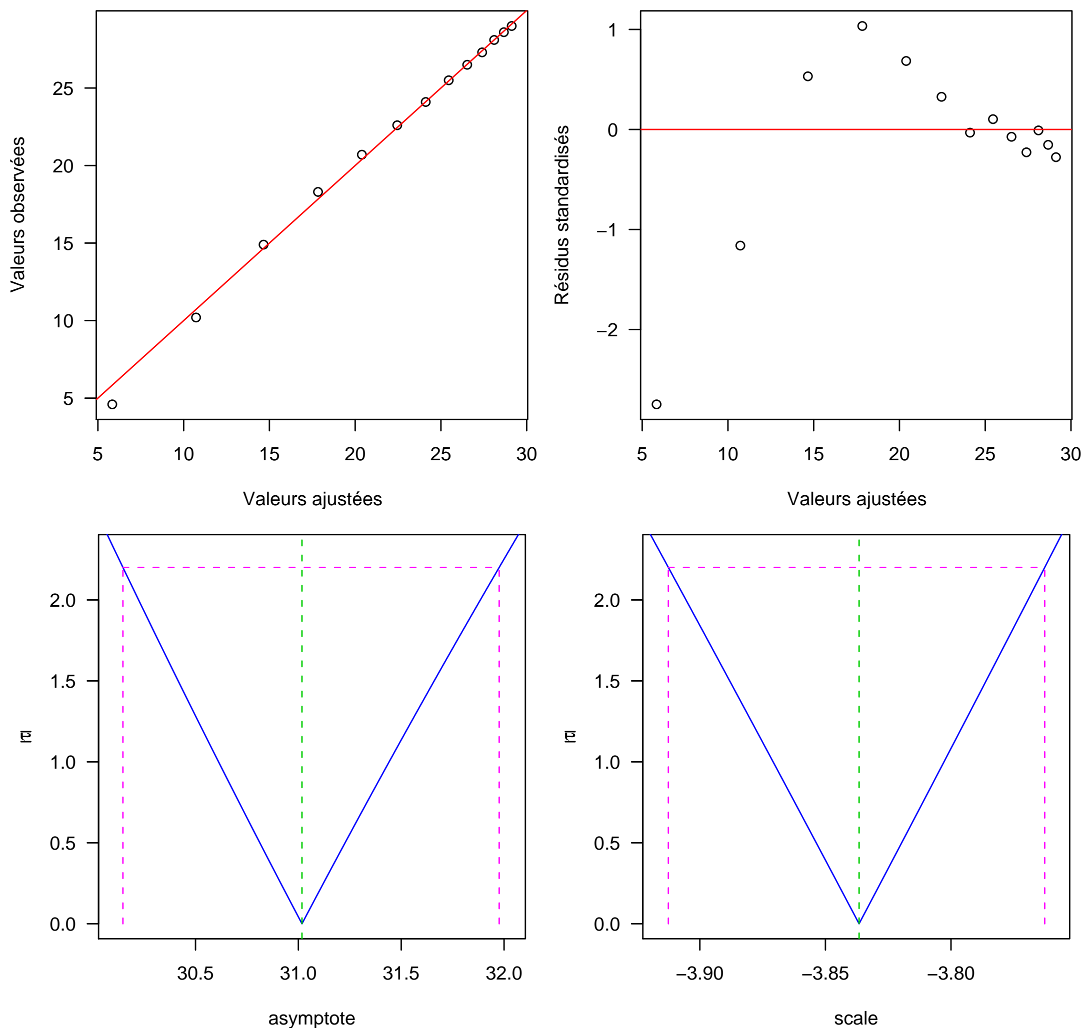


Figure 12.2 – Tracés diagnostics et des profils d'un simple modèle asymptotique avec un arbre seul.

La figure 12.2 nous montre un récapitulatif graphique utile de la performance du modèle, combinant des informations sur les résidus avec des tracés de profils. En haut à gauche est un tracé des valeurs observées fonction des valeurs ajustées, avec une droite $x = y$ imposée. Nous aimerions que les points soient à proximité de cette droite, et ne montrent pas de dispersion ou de courbure. En haut à droite est un tracé des résidus standardisés en fonction des valeurs ajustées, avec une droite $y = 0$ imposée. Comme pour la régression aux MCO, nous ne voulons voir aucun

2. ou non.

profil particulier et aucune valeurs aberrantes influentes. Les panneaux inférieurs affichent les tracés de profil, un pour chaque estimation de paramètre. Ces graphiques fournissent des informations sur l'acceptabilité de l'hypothèse de normalité de la distribution sous-jacente des estimations de paramètres. Notre scénario idéal est que les lignes bleues pleines soient des droites, et les droites verticales roses pointillées coupent l'axe des $x$ sur un intervalle de confiance à 95% pour les grands échantillons. Nous pouvons calculer rapidement estimer ce grand échantillon en utilisant :

```
> (my.t <- qt(0.975, summary(handy.nls)$df[2]))
```

```
[1] 2.200985
```

```
> coef(summary(handy.nls))[, 1:2] %*% matrix(c(1, -my.t, 1, my.t),
+     nrow = 2)
```

```
                [,1]      [,2]
asymptote 30.096977 31.938015
scale     -3.912319 -3.760880
```

et extraire les versions profilées avec :

```
> confint(handy.nls)
```

```
               2.5%     97.5%
asymptote 30.147048 31.976382
scale     -3.912546 -3.762686
```

Nos diagnostics ne laissent aucun doute sur la validité de notre modèle. Mais, les données semblent avoir un léger coude que notre modèle n'a pas réussi à reproduire. Cela pourrait être vrai pour cet arbre seulement, pour un sous-ensemble des arbres ou pour tous les arbres. Nous sommes curieux par nature, mais, on le comprend, réticents à exécuter par le même processus pour chaque arbre. Nous aurons besoin d'outils plus généraux.

### 12.1.2 Plusieurs arbres

Dans la section précédente, nous avons ajusté un bon modèle non linéaire à un seul arbre. Il serait bien d'être en mesure d'ajuster le même modèle à une collection d'arbres avec un minimum d'embaras. Le nouveau paquetage `nlme` nous donne une voie à suivre. Tout d'abord, nous pouvons utiliser la structure `groupedData` pour simplifier le problème d'ajustement d'un modèle à de nombreux arbres et, ensuite, nous pouvons utiliser une fonction d'« auto-allumage », qui fournit ses propres valeurs de départ aux données qui lui sont présentées. Cet auto-allumage est une fonction pré-packagée mentionné plus tôt et son adoption simplifie considérablement notre démarche.

```
> require(nlme)
> gutten.d <- groupedData(dbh.cm ~ age.bh | tree.ID, data = gutten)
```

La fonction d'auto-allumage s'appelle `SSasympOrig`.

```
> gutten.nlsList <- nlsList(dbh.cm ~ SSasympOrig(age.bh, asymptote,
+     scale), data = gutten.d)
```

Nous sommes maintenant en mesure de décider si oui ou non, le coude observé dans les résidus de l'arbre 1.1.1 est répété dans les autres arbres (figure 12.3). Ces resultats suggèrent qu'il y a certainement un manque systématique d'ajustement dans l'ensemble. Nous pouvons avoir besoin d'adopter une fonction plus souple.

```
> plot(gutten.nlsList, residuals(., type = "pearson") ~ fitted(.) |
+     tree.ID, xlab = "Valeurs ajustées", ylab = "Résidus standardisés")
```

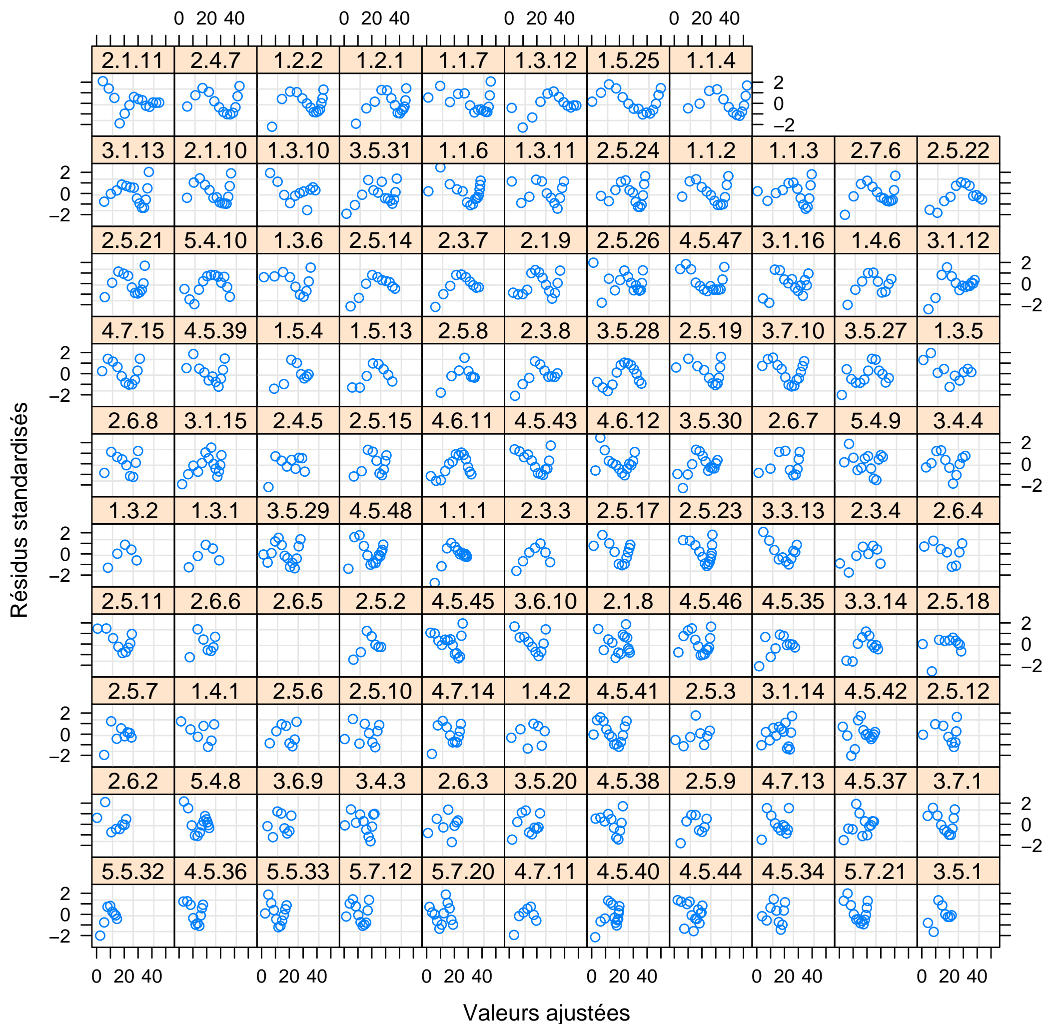


Figure 12.3 – Tracés des résidus en fonction des valeurs ajustées par des modèles non linéaires pour chaque arbre.

Nous pourrions imprimer toutes les estimations des paramètres, mais il est plus utile de construire un récapitulatif graphique (figure 12.4). Noter que les intervalles sont basés sur la théorie des grands échantillons.

```
> plot(intervals(gutten.nlsList), layout = c(2, 1))
```

Nous pouvons extraire et manipuler les estimations de coefficient de façon plus approfondie. L'approfondissement suit, et peut être sauté. En premier, nous essayons de trouver quelles sont les fonctions disponibles pour manipuler des objets de la classe de l'objet que l'on vient de créer.

```
> methods(class = class(gutten.nlsList))

[1] formula.nlsList* nlme.nlsList     summary.nlsList*
[4] update.nlsList*

   Non-visible functions are asterisked
```

Une méthode `summary` est disponible. Trouvons quelles fonctions son disponibles pour manipuler les objets de `summary`.

```
> methods(class = class(summary(gutten.nlsList)))

[1] coef.summary.nlsList*

   Non-visible functions are asterisked
```

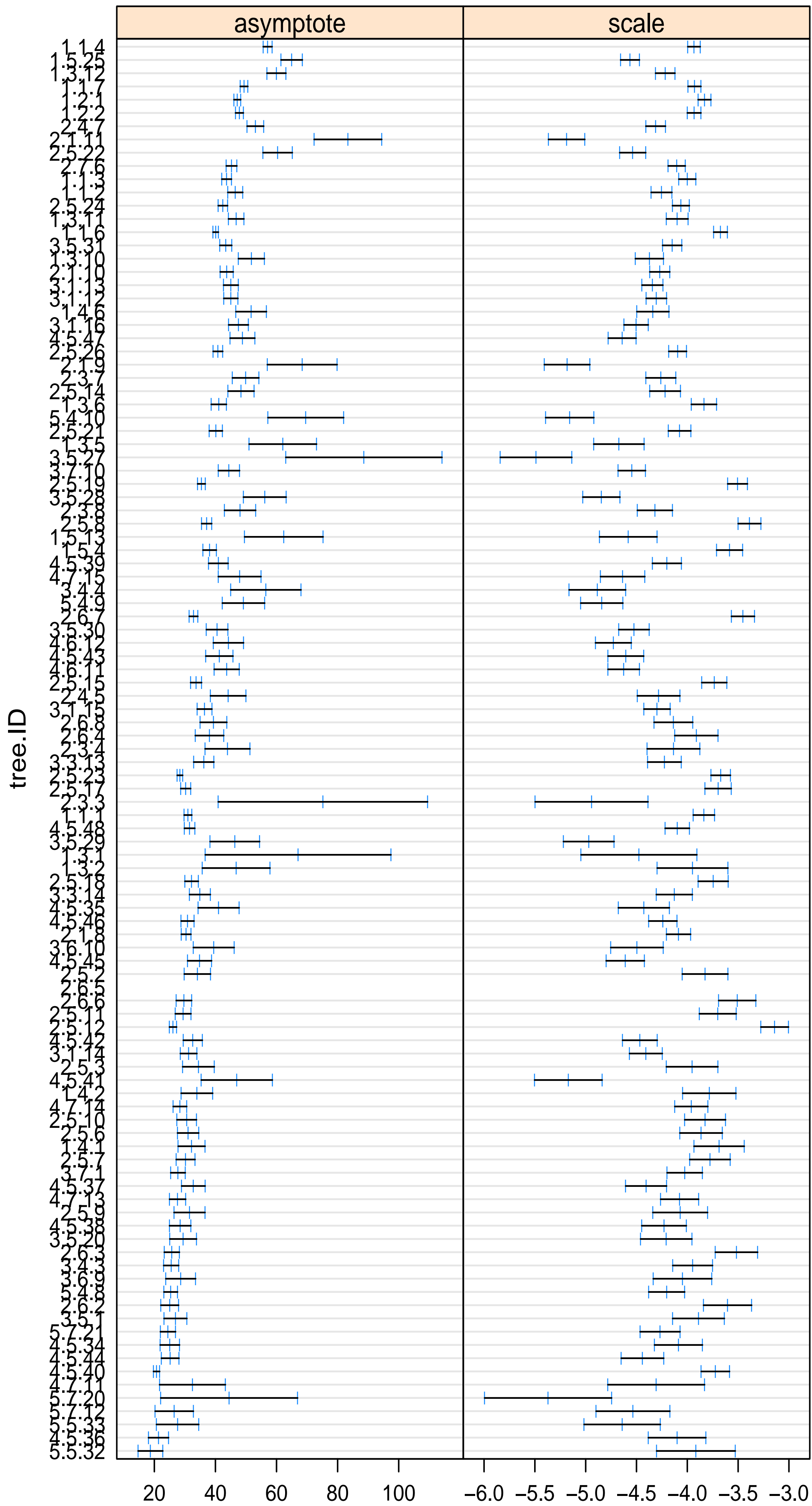


Figure 12.4 – Tracés des intervalles de prédiction du diamètre ajusté sur les données Gutenberg.

Une méthode coef() est disponible. De quoi a l'air sa sortie ?

```
> str(coef(summary(gutten.nlsList)))
```

```
 num [1:107, 1:4, 1:2] 18.7 21.4 27.6 26.5 44.5 ...
 - attr(*, "dimnames")=List of 3
  ..$ : chr [1:107] "5.5.32" "4.5.36" "5.5.33" "5.7.12" ...
  ..$ : chr [1:4] "Estimate" "Std. Error" "t value" "Pr(>|t|)"
  ..$ : chr [1:2] "asymptote" "scale"
```

C'est un tableau. On en extrait les éléments ainsi :

```
> asymptote <- coef(summary(gutten.nlsList))[, "Estimate", "asymptote"]
> half.age <- log(2)/exp(coef(summary(gutten.nlsList))[, "Estimate",
+     "scale"])
```

Donc, retour à l'exercice. La figure 12.5 montre un diagramme de dispersion des paramètres estimés pour chaque arbre : l'estimation de l'asymptote sur l'axe des $y$ et l'estimation de l'âge auquel l'arbre atteint la moitié de son diamètre maximal sur l'axe des x, avec un lissage *loess* ajouté pour décrire la moyenne du motif. Il y a clairement une relation entre l'asymptote et l'estimation de l'âge où la moitié à l'asymptote est atteinte.

```
> opar <- par(las = 1, mar = c(4, 4, 1, 1))
> scatter.smooth(half.age, asymptote, xlab = "Age", ylab = "Asymptote")
```

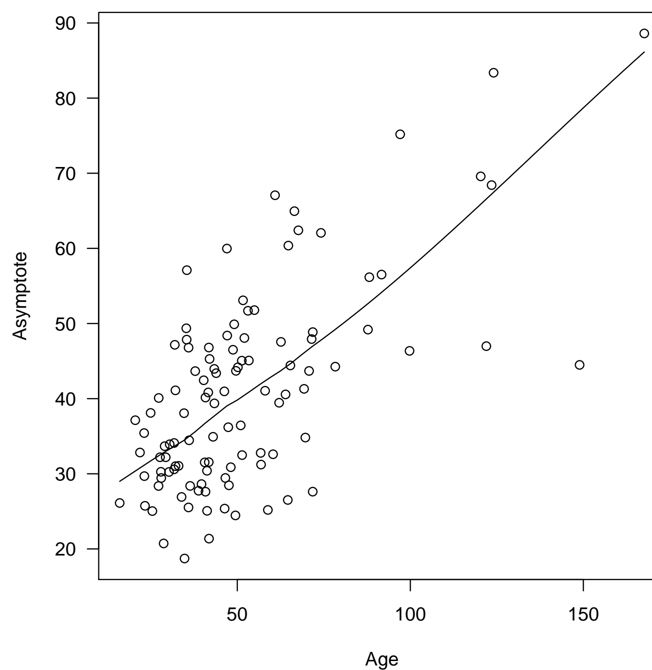


Figure 12.5 – Tracé du diamètre maximal estimé en fonction de l'âge estimé auquel l'arbre atteint la moitié de son diamètre maximal selon l'arbre.

Cette approche nous offre un moyen pratique de penser à l'adaptation des données à de nombreux objets différents. Mais, quelle est l'utilisation pratique du modèle ? Pas beaucoup. Ce modèle est analogue au modèle linéaire qui comprend une ordonnée à l'origine pour chaque parcelle : les hypothèses du modèle sont (probablement) satisfaites, mais le modèle n'est pas vraiment utile.

Nous pourrions aussi essayer d'adapter le modèle à tous les arbres à la fois, c'est-à-dire en ignorant les différences entre les arbres.

```
> (gutten.nls <- nls(dbh.cm ~ diameter.growth(age.bh, asymptote,
+     scale), start = list(asymptote = 50, scale = -5), data = gutten))
```

```
Nonlinear regression model
  model:  dbh.cm ~ diameter.growth(age.bh, asymptote, scale)
   data:  gutten
asymptote     scale
    38.56     -4.20
 residual sum-of-squares: 39130

Number of iterations to convergence: 5
Achieved convergence tolerance: 3.027e-06
```

Ce modèle a des diagnostics montrés en figure 12.6, et construits comme suit :

```
> opar <- par(mfrow = c(2, 2), mar = c(4, 4, 2, 1), las = 1)
> plot(fitted(gutten.nls), gutten$dbh.cm, xlab = "Fitted Values",
+     ylab = "Observed Values")
> abline(0, 1, col = "red")
> plot(fitted(gutten.nls), residuals(gutten.nls, type = "pearson"),
+     xlab = "Fitted Values", ylab = "Standardized Residuals")
> abline(h = 0, col = "red")
> plot(profile(gutten.nls), conf = 0.95)
> par(opar)
```

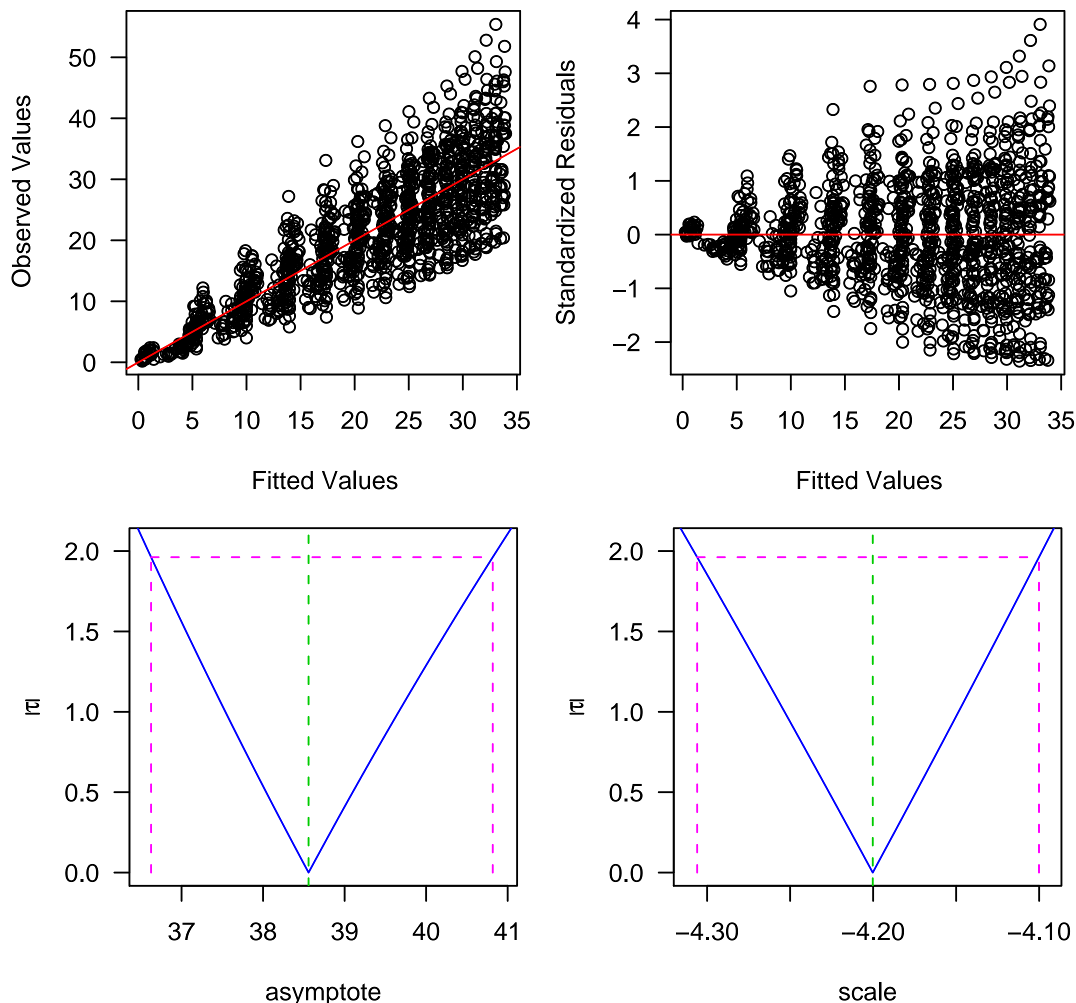


Figure 12.6 – Tracés des profils pour un simple modèle asymptotique avec toutes les données.

À quoi ressemble le modèle ? En voici l'estimation de l'intervalle de confiance à 95% :

```
> (my.t <- qt(0.975, summary(gutten.nls)$df[2]))

[1] 1.961946
```

```
> coef(summary(gutten.nls))[, 1:2] %*% matrix(c(1, -my.t, 1, my.t),
+     nrow = 2)

                [,1]      [,2]
asymptote 36.522095 40.593577
scale     -4.300579 -4.099923
```

Le comparer avec ceci :

```
> confint(gutten.nls)

               2.5%     97.5%
asymptote 36.626641 40.817138
scale     -4.306002 -4.100121
```

Cela semble correct. C'est dommage pour les diagnostics.

## 12.2 Splines

Nous avons déjà adopté l'approche traditionnelle de choisir et ajuster des modèles pour lesquels un modèle linéaire paraît inadapté. Ce qui sous-tendait notre approche a été l'hypothèse tacite que nous étions intéressés à connaître les valeurs de certains paramètres, dans notre cas l'asymptote et le facteur. Qu'en est-il si nous ne sommes intéressés que par la prédiction ? C'est-à-dire, que faire si notre objectif est d'utiliser le modèle pour ses sorties seulement et que les estimations de paramètres, quels qu'ils soient, ne sont pas pertinentes ? Dans ces conditions, il pourrait être plus raisonnable d'utiliser une spline, ou d'un modèle additif.

### 12.2.1 Splines cubiques

La fonction `bs()` produit la matrice de modèle pour un spline cubique, en utilisant le B-spline de base. Le B-spline de base est séduisant parce que les fonctions sont locales, tellement, qu'elles sont à zéro dans les zones où ils ne sont voulues. Lire Wood (2006)[28] pour une explication très lisible [3].

Ici, nous utilisons la fonction pour ajuster un modèle linéaire qui représente un spline cubique à un seul nœud (dl - degré = 1). Notez que, par ce moyen, nous avons fixé la complexité du modèle.

```
> require(splines)
> handy.spline <- lm(dbh.cm ~ bs(age.bh, df = 4, degree = 3), data = handy.tree)
> summary(handy.spline)

Call:
lm(formula = dbh.cm ~ bs(age.bh, df = 4, degree = 3), data = handy.tree)

Residuals:
      Min        1Q    Median        3Q       Max
-0.136378 -0.053526  0.009092  0.045355  0.096675

Coefficients:
                                Estimate Std. Error t value Pr(>|t|)
(Intercept)                      4.55464    0.07893   57.71 9.03e-12 ***
bs(age.bh, df = 4, degree = 3)1 13.01407    0.17698   73.54 1.30e-12 ***
bs(age.bh, df = 4, degree = 3)2 20.74721    0.17750  116.88 3.21e-14 ***
bs(age.bh, df = 4, degree = 3)3 23.85037    0.16071  148.40 4.75e-15 ***
bs(age.bh, df = 4, degree = 3)4 24.43626    0.10916  223.85  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.08531 on 8 degrees of freedom
Multiple R-squared: 0.9999,      Adjusted R-squared: 0.9999
F-statistic: 2.406e+04 on 4 and 8 DF,  p-value: 2.386e-16
```

3. C'est réellement un très bon livre pour beaucoup d'autres raisons, pas la moindre quantité qui ne soit clarté d'auteur et volonté de faire des diagrammes à tout instant

Les diagnostics du modèle sont montrés en figure 12.7.

```
> opar <- par(las = 1, mfrow = c(2, 2), mar = c(4, 4, 2, 2))
> plot(dbh.cm ~ age.bh, data = handy.tree)
> curve(predict(handy.spline, data.frame(age.bh = x)), from = 10,
+     to = 140, add = TRUE)
> plot(handy.spline, which = c(1, 2, 5))
> par(opar)
```

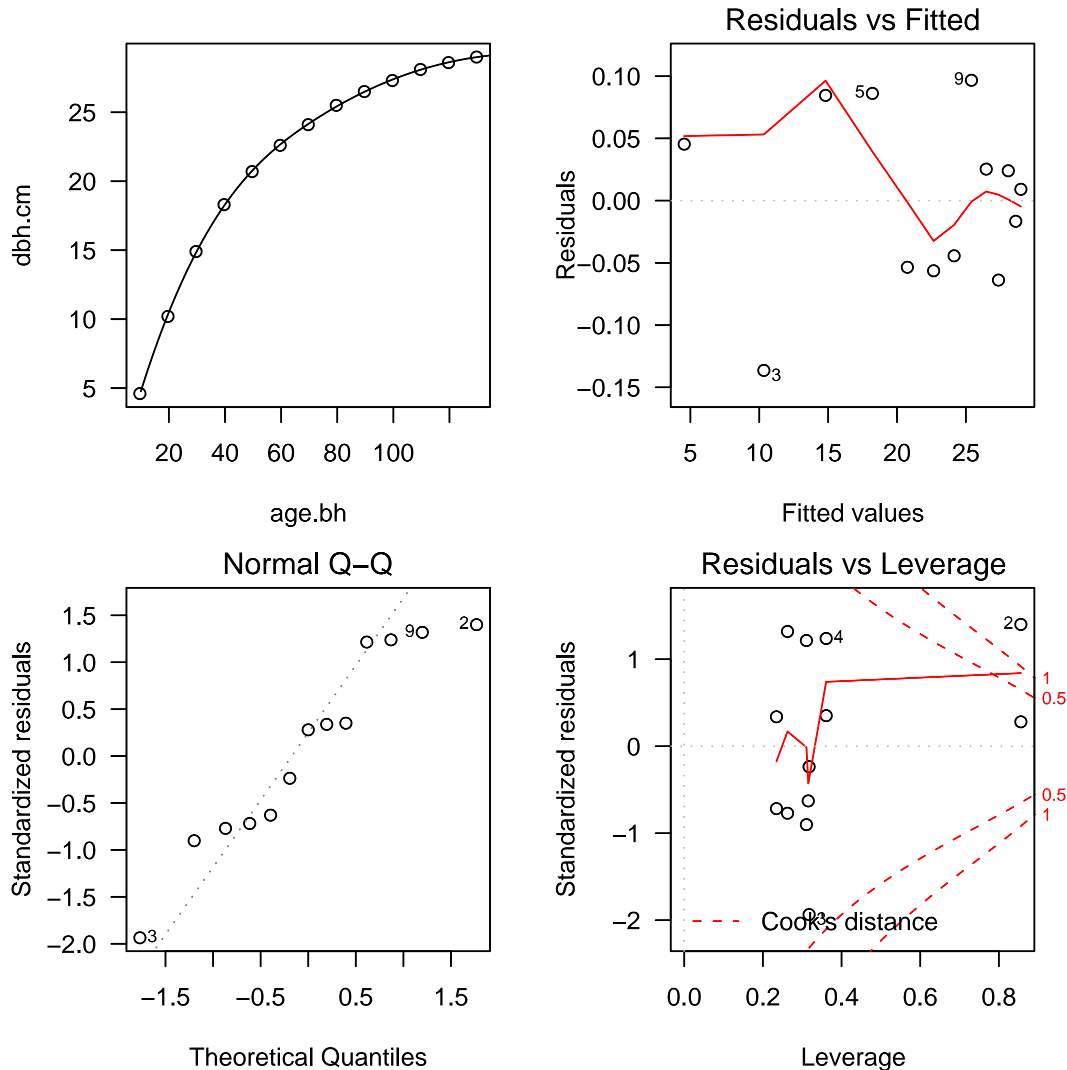


Figure 12.7 – Récapitulé graphique de l'ajustement spline.

Dans l'ensemble, le modèle semble bon, comme on pouvait s'y attendre, avec seulement un point particulier de grande influence : 2.

```
> handy.tree["2", ]
```

```
  site location tree dbh.cm age.bh tree.ID
2    1        1    1    4.6   9.67   1.1.1
```

Le premier point mesuré. Sinon, le modèle semble bien, mais la sélection de nature arbitraire du compter de noeud est un peu sujette à caution.

### 12.2.2 Modèle additif

Les splines cubiques sont suffisamment souples et l'analyste prudent peut s'inquiéter de la possibilité de sur-ajustement. Il semble naturel de chercher à freiner le nombre de noeuds, ou de degrés de liberté, qui sont à la

disposition du modèle. Par exemple, nous pourrions ajuster une suite de modèles, et les comparer par certains moyens, avec une pénalité de courbure. **R** possède une telle solution dans le paquetage `mgcv` (Wood, 2006)[28], entre autres. Ce paquetage utilise la dite Validation Croisée Généralisée (VCG) pour sélectionner la complexité du modèle, et selon la complexité, présenter des statistiques d'ajustement appropriées.

La VCG représente la prédiction estimée de la variance d'erreur, c'est-à-dire, la variance des erreurs de prédiction de points inobservés. En gros, la racine carrée de la statistique VCG est dans les unités de la variable réponse, ici, le centimètre.

```
> require(mgcv)
```

```
This is mgcv 1.4-1
```

```
> handy.gam <- gam(dbh.cm ~ s(age.bh), data = handy.tree)
> summary(handy.gam)
```

```
Family: gaussian
Link function: identity

Formula:
dbh.cm ~ s(age.bh)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 21.56923    0.01325    1628 2.47e-12 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
            edf Ref.df     F  p-value
s(age.bh) 8.168  8.668 35412 4.33e-09 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

R-sq.(adj) =      1   Deviance explained =  100%
GCV score = 0.0077406   Scale est. = 0.0022818  n = 13
```

```
> sqrt(summary(handy.gam)$gcv)
```

```
[1] 0.08798043
```

La figure 12.8 fournit un récapitulatif de la structure du modèle. Le graphique en haut à gauche montre la courbe prédite, *avec un polygone ombré d'erreur standard superposé.* les trois autres sont des tracés diagnostiques standard de régression.

```
> par(las = 1, mfrow = c(2, 2), mar = c(4, 4, 2, 1))
> plot(handy.gam, rug = FALSE, shade = TRUE)
> plot(predict(handy.gam), residuals(handy.gam))
> abline(h = 0, col = "springgreen3")
> qqnorm(residuals(handy.gam))
> qqline(residuals(handy.gam), col = "springgreen3")
> scatter.smooth(predict(handy.gam), sqrt(abs(residuals(handy.gam))))
```

## 12.3 Hiérarchique

Nous allons maintenant nous concentrer sur l'ajustement des mêmes types de modèles à des données hiérarchiques. Cette direction produit des simplifications et des complications, mais malheureusement, plus des dernières que des premières.

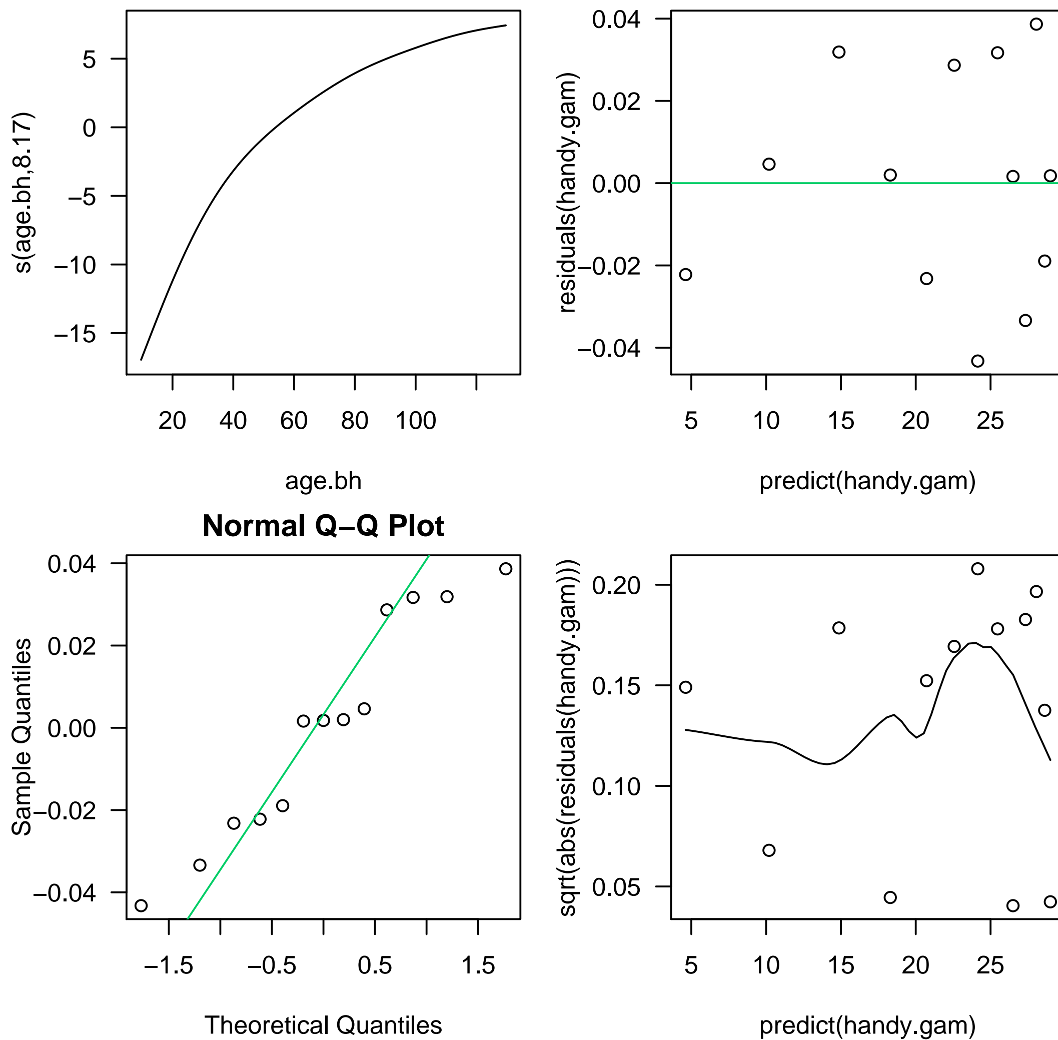


Figure 12.8 – Tracés diagnostiques du modèle additif généralisé pour prédire le diamètre en fonction de l'âge d'un arbre commun.

Nous commençons avec un modèle à effets mixtes non-linéaire qui généralise notre précédent modèle non-linéaire (équation 12.1). Nous allons commencer par permettre à chaque arbre d'avoir une asymptote aléatoire et un facteur. C'est-à-dire, pour la mesure de diamètre t de l'arbre i :

$$y_{it} = (\phi_1 + \phi_{i1}) \times (1 - e^{-e^{(\phi_2+\phi_{i2})x}}) + \varepsilon_{it} \tag{12.2}$$

où $\phi_1$ est l'asymptote inconnue, fixe, $\phi_2$ le facteur inconnu, fixe, et

$$\begin{pmatrix}\phi_{i1}\\ \phi_{i2}\end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}\sigma_1^2 & \sigma_{12}\\ \sigma_{12} & \sigma_2^2\end{pmatrix}\right) \tag{12.3}$$

Le processus d'ajustement et la critique de ces modèles est bien documentée dans Pinheiro et Bates (2000)[15]. Étant donné que nous avons déjà utilisé `nlsList()`, la meilleure approche des deux points de vue, de la frappe, et d'avoir des points de départ raisonnables, est la suivante :

```
> gutten.nlme.0 <- nlme(gutten.nlsList)
> summary(gutten.nlme.0)

Nonlinear mixed-effects model fit by maximum likelihood
  Model: dbh.cm ~ SSasympOrig(age.bh, asymptote, scale)
 Data: gutten.d
       AIC      BIC    logLik
  3627.763 3658.304 -1807.882

Random effects:
```

```
 Formula: list(asymptote ~ 1, scale ~ 1)
 Level: tree.ID
 Structure: General positive-definite, Log-Cholesky parametrization
          StdDev     Corr
asymptote 12.2386790 asympt
scale      0.4191592 -0.596
Residual   0.7092730

Fixed effects: list(asymptote ~ 1, scale ~ 1)
             Value Std.Error   DF    t-value p-value
asymptote 40.19368 1.2164375 1092   33.04212       0
scale     -4.20333 0.0417687 1092 -100.63355       0
 Correlation:
      asympt
scale -0.613

Standardized Within-Group Residuals:
        Min          Q1         Med          Q3         Max
-4.50181460 -0.51490231 -0.03653829  0.44912909  3.96736354

Number of Observations: 1200
Number of Groups: 107
```

Si on veut construire le modèle à partir de rien, il faudrait faire ceci :

```
> gutten.nlme.0 <- nlme(dbh.cm ~ SSasympOrig(age.bh, asymptote, scale),
+                       fixed = asymptote + scale ~ 1,
+                       random = asymptote + scale ~ 1,
+                       start = c(asymptote = 50, scale = -5),
+                       data = gutten.d)
```

Comme avec les modèles à effets mixtes, nous avons un large tableau de différents tracés diagnostiques que nous pouvons déployer, et les fonctions pour obtenir ces diagnostics sont presque identiques. Ici nous allons nous concentrer sur l'examen au sein de l'auto-corrélation intra-arbre (figure 12.9).

```
> plot(ACF(gutten.nlme.0, form = ~1 | tree.ID), alpha = 0.01,
+          xlab = "Décalage", ylab = "Auto-corrélation")
```

La figure montre qu'il y a une importante auto-corrélation intra-arbre, ce qui suggère un manque systématique d'ajustement. À partir de là, deux options sont possibles : essayer une autre moyenne ou essayer de modéliser l'auto-corrélation. En général, on doit les essayer dans cet ordre. Pour le moment, cependant, voyons la modélisation de l'auto-corrélation.

Après quelques expériences, je suis arrivé à :

```
> gutten.nlme.1 <- update(gutten.nlme.0,
+                         correlation = corARMA(p = 1, q = 2))
```

Qui produit des résidus avec motif d'auto-corrélation présenté en figure 12.10.

```
> plot(ACF(gutten.nlme.1, resType = "n", form = ~1 | tree.ID),
+     alpha = 0.01, xlab = "Décalage", ylab = "Auto-corrélation")
```

C'est clairement supérieur au modèle précédent mais demande encore des améliorations. Peut-être y a t-il trop de manque d'ajustement ? Comme nous avons construit un objet `groupedData`, on peut utiliser le tracé de prédiction agrémenté comme un récapitulatif très utile du comportement du modèle. Ce graphique est présenté en figure 12.11.

```
> plot(augPred(gutten.nlme.1))
```

On peut obtenir les intervalles de confiance à 95% des estimations avec l'appel suivant :

```
> intervals(gutten.nlme.1)
```

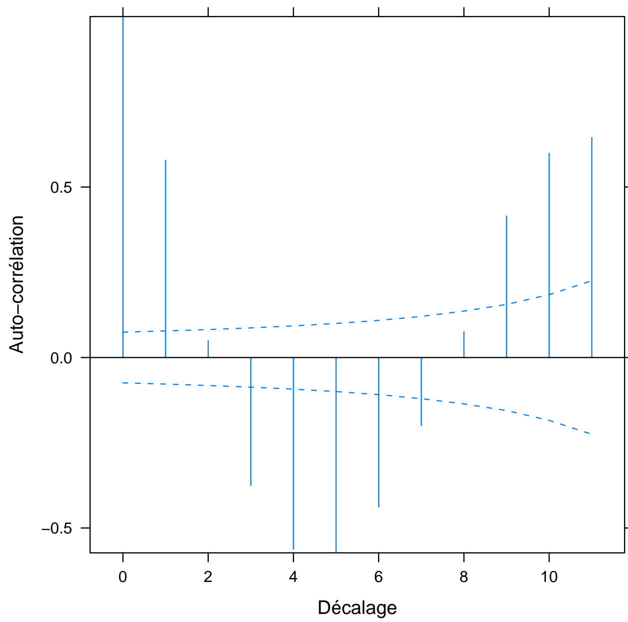


Figure 12.9 – Auto-corrélation des résidus intra-arbre pour le modèle à effets mixtes non linéaire.

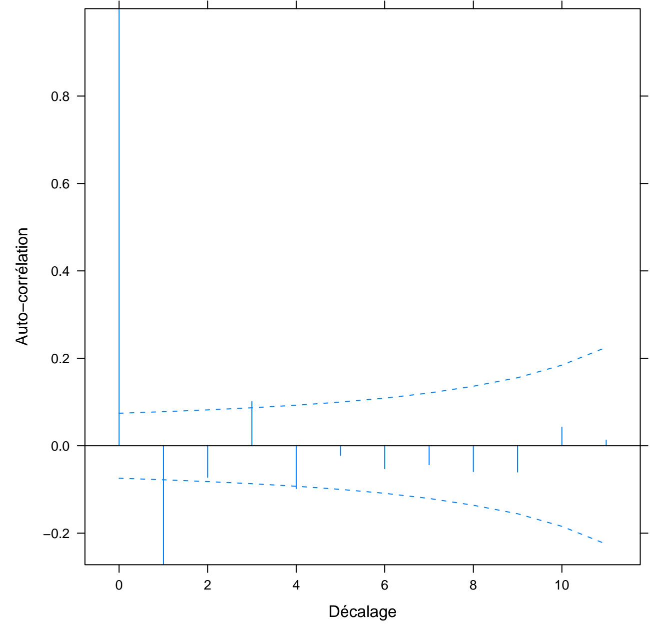


Figure 12.10 – Auto-corrélation des résidus intra-arbre pour le modèle à effets mixtes non linéaire avec modèle explicite d'auto-corrélation.

```
Approximate 95% confidence intervals

 Fixed effects:
              lower      est.     upper
asymptote 36.477681 38.649530 40.821379
scale     -4.203298 -4.119053 -4.034807
attr(,"label")
[1] "Fixed effects:"

 Random Effects:
  Level: tree.ID
                            lower       est.      upper
sd(asymptote)           8.9040681 10.5144072 12.4159831
sd(scale)               0.3232660  0.3855318  0.4597909
cor(asymptote,scale) -0.6726752 -0.5225958 -0.3312174

 Correlation structure:
           lower      est.     upper
Phi1   0.7550318 0.8217120 0.8715701
Theta1 0.2939313 0.4007553 0.5092043
Theta2 0.5606067 0.6827487 0.7758152
attr(,"label")
[1] "Correlation structure:"

 Within-group standard error:
   lower     est.    upper
1.291425 1.524229 1.799001
```

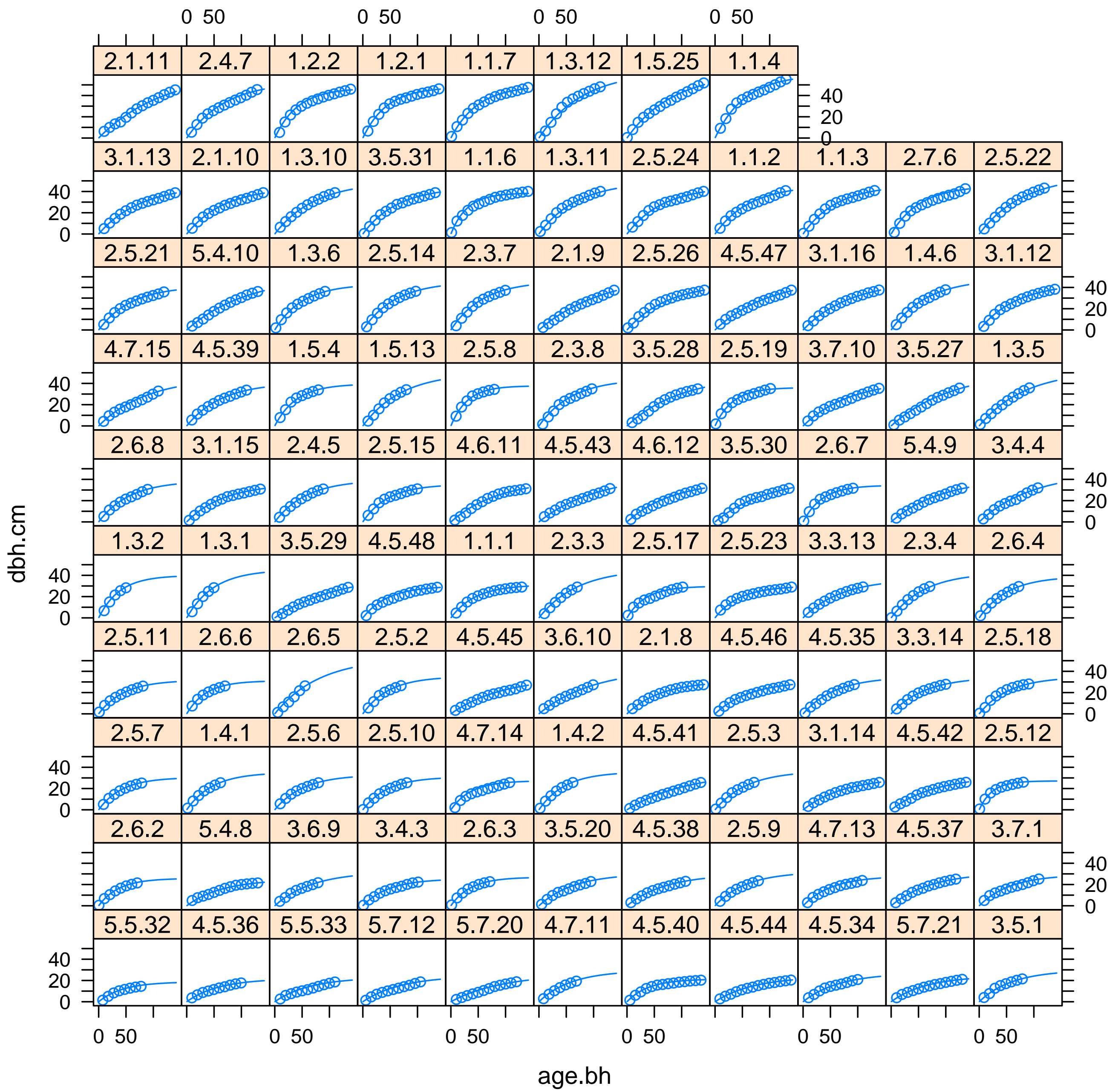


Figure 12.11 – Données diamètre de Guttenberg avec modèle asymptotique d'ajustement passant par l'origine et auto-corrélation agrémentée. Ici les modèles de niveau arbre équation (12.2) sont tracés.

# Bibliographie


[1] Bates, D. M., Watts, D. G., 1988. *Nonlinear regression analysis and its applications.* John Wiley & Sons, Inc., 365 p.

[2] Box, G. E. P., 1953. *Non-normality and tests on variances.* Biometrika 40 (3/4), 318-–335.

[3] Casella, G., Berger, R. L., 1990. *Statistical Inference.* Duxbury Press, 650 p.

[4] Chambers, J. M., 2008. *Software for Data Analysis : Programming With R.* Springer, 512 p.

[5] Daubenmire, R., 1952. *Forest vegetation of Northern Idaho and adjacent Washington, and its bearing on concepts of vegetation classification.* Ecological Monographs 22, 301-–330.

[6] Davison, A. C., Hinkley, D. V., 1997. *Bootstrap methods and their application.* Cambridge University Press, 582 p.

[7] Demidenko, E., Stukel, T. A., 2005. *Influence analysis for linear mixed-effects models.* Statistics in Medicine 24, 893-–909.

[8] Fitzmaurice, G., Laird, N., Ware, J., 2004. *Applied Longitudinal Analysis.* John Wiley & Sons, 506 p.

[9] Gallant, A. R., 1987. *Nonlinear statistical models.* John Wiley & Sons, 624 p.

[10] Gelman, A., Hill, J., 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models.* Cambridge University Press, 625 p.

[11] Hollings, Triggs, 1993. *Influence of the new rules in international rugby football : Implications for conditioning,* Unpublished.

[12] Huber, P. J., 1981. *Robust Statistics.* John Wiley & Sons, Inc., 308 p.

[13] Laird, N. M., Ware, J. H., 1982. *Random-effects models for longitudinal data.* Biometrics 38, 963-–974.

[14] Lee, A., 1994. *Data Analysis : An introduction based on R.* Department of Statistics, University of Auckland.

[15] Pinheiro, J. C., Bates, D. M., 2000. *Mixed-effects models in S and Splus.* Springer-Verlag, 528 p.

[16] Ratkowsky, D. A., 1983. *Nonlinear regression modeling.* Marcel Dekker, New York, 276 p.

[17] Robinson, A. P., Wykoff, W. R., 2004. *Imputing missing height measures using a mixedeffects modeling strategy.* Canadian Journal of Forest Research 34, 2492-–2500.

[18] Robinson, G. K., 1991. *That BLUP is a good thing : The estimation of random effects.* Statistical Science 6 (1), 15-–32.

[19] Schabenberger, O., 2005. *Mixed model influence diagnostics.* In : SUGI 29. SAS Institute, pp. Paper 189-–29.

[20] Schabenberger, O., Pierce, F. J., 2002. *Contemporary statistical models for the plant and soil sciences.* CRC Press, 738 p.

[21] Seber, G. A. F., Wild, C. J., 2003. *Nonlinear regression.* John Wiley & Sons, 792 p.

[22] Stage, A. R., 1963. *A mathematical approach to polymorphic site index curves for grand fir.* Forest Science 9 (2), 167-–180.

[23] Venables, W. N., Ripley, B. D., 2000. *S programming.* Springer-Verlag, 264 p. 99

[24] Venables, W. N., Ripley, B. D., 2002. *Modern applied statistics with S and Splus, 4th Ed.* Springer–Verlag, 495 p.

[25] von Guttenberg, A. R., 1915. *Growth and yield of spruce in Hochgebirge.* Franz Deuticke, Vienna, 153 p. (In German).

[26] Weisberg, S., 2005. *Applied Linear Regression, 3rd Edition.* John Wiley & Sons, Inc., 310 p.

[27] Wilkinson, L., 2005. *The Grammar of Graphics, 2nd Edition.* Springer, 694 p.

[28] Wood, S. N., 2006. *Generalized additive models : an introduction with R.* Chapman & Hall/CRC, 391 p.

[29] Wykoff, W. R., Crookston, N. L., Stage, A. R., 1982. *User's guide to the Stand Prognosis model.* GTR–INT 133, USDA Forest Service.