

# Introduction à R

Laurent Rouvière

2020-06-22

## Table des matières

<b>Présentation</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 R Script . . . . .	2
1.2 Packages . . . . .	2
1.3 Markdown . . . . .	3
<b>2 Les objets R</b>	<b>8</b>
2.1 Création d'objets . . . . .	8
2.2 Vecteur . . . . .	9
2.3 Matrices . . . . .	11
2.4 Listes . . . . .	12
2.5 Dataframe . . . . .	13
2.6 Quelques fonctions importantes . . . . .	14
2.7 Exercices complémentaires . . . . .	14
<b>3 Manipuler les données avec dplyr</b>	<b>18</b>
3.1 Importer des données . . . . .	18
3.2 Le package dplyr . . . . .	21
<b>4 Visualisation avec ggplot2</b>	<b>40</b>
4.1 Fonctions graphiques conventionnelles . . . . .	40
4.2 La grammaire ggplot2 . . . . .	51
4.3 Compléments . . . . .	80
4.4 Quelques exercices supplémentaires . . . . .	88
<b>5 Faire des cartes interactives avec leaflet</b>	<b>108</b>
5.1 Présentation de leaflet . . . . .	108
5.2 Challenge : Visualisation des stations velib à Paris . . . . .	115
<b>6 Faire de la régression sur R</b>	<b>122</b>
6.1 Modèle linéaire : fonctions lm et predict . . . . .	123
6.2 Sélection de variables . . . . .	125
6.3 Régression logistique et arbre . . . . .	126

## Présentation

Ce tutoriel présente une introduction au logiciel R. Les thèmes suivants sont abordés :

- **Présentation du logiciel**, environnement Rstudio, reporting avec Rmarkdown

- **Objets R**
- **Manipulation des données** (essentiellement avec `dplyr`)
- **Visualisation de données** (représentations standards et avec `ggplot2`)
- **Cartes dynamiques** avec `leaflet`
- **Régression** : ajustement de modèles, formules, prévisions...

On pourra trouver des supports de cours ainsi que les données utilisées à l'adresse suivante [https://lrouviere.github.io/intro\\_R/](https://lrouviere.github.io/intro_R/). Des compléments sur les outils du tidyverse pourront être consultés dans le très complet document de Barnier (2020) ainsi que les ouvrages de Wickham and Grolemund (2017) et de Cornillon et al. (2018).

## 1 Introduction

**R** est un logiciel libre et gratuit principalement dédié aux analyses statistiques et aux représentations graphiques. Il est gratuit et librement distribué par le **CRAN** (Comprehensive R Archive Network) à l'adresse suivante : <https://www.r-project.org>.

L'installation varie d'un système d'exploitation à l'autre (Windows, Mac OS, Linux) mais elle est relativement simple, il suffit de suivre les instructions.

**RStudio** est une interface facilitant l'utilisation de **R**. Elle est également gratuite et librement distribuée à l'adresse <https://www.rstudio.com>.

L'interface **RStudio** est divisée en 4 fenêtres :

- *Console* où on peut entrer et exécuter des commandes (taper 1+2)
- *Environnement, History* où on peut visualiser les objets construits (taper a <- 1+2 dans la console)
- *Files, Plots...* où on peut visualiser les répertoires et fichiers de l'espace de travail, les graphiques, installer des packages...
- *R script* où on conserve les lignes de commandes ainsi que les commentaires sur le travail effectué. Il faut penser à sauvegarder régulièrement ce fichier.

### 1.1 R Script

Il existe différentes façons de travailler sur RStudio. De façon classique, on peut

- ouvrir un **script**.
- entrer les commandes dans le script.
- regarder les sorties dans la console (en cliquant sur le bouton **run**).
- sauver le script.

### 1.2 Packages

Un package est une ensemble de programmes et fonctions **R** qui complètent les fonctions existantes par défaut dans le logiciel. Un package est généralement dédié à une méthode ou un champ d'application spécifique. Il existe plus de 13 000 packages disponibles sur le **CRAN** <https://cran.r-project.org>. On installe un package en

- utilisant la fonction `install.packages` dans la console. ou
- ou cliquant sur le bouton *Packages*.

Une fois le package installé sur la machine, on l'installe avec la fonction `library` :

```
> install.packages(package.name)
> library(packages.name)
```

**Exercice 1.1** (Installation et chargement).

1. Exécuter

```
> iris %>% summarize(mean_Petal=mean(Petal.Length))
```

Que se passe t-il ?

2. Installer et charger le package **tidyverse** et ré-exécuter le code précédent.

```
> install.packages("tidyverse")
> library(tidyverse)
> iris %>% summarize(mean_Petal=mean(Petal.Length))
  mean_Petal
1      3.758
```

## 1.3 Markdown

**markdown** est un package qui permet de créer différents types de documents :

- rapports au format pdf ou rtf
- pages web html
- diaporama pour des présentations (html, beamer, PowerPoint...)
- applications web interactives
- ...

qui comportent du code **R**.

### 1.3.1 Syntaxe

La syntaxe s'apprend assez facilement (il faut pratiquer), on pourra trouver un descriptif synthétique dans la **cheatsheet** dédié à **Rmarkdown**. Par exemple :

- Caractère en italique ou gras : **\*italique\*** et **\*\*gras\*\*** donne *italique* et **gras**
- Listes non ordonnées

```
- item 1
- item 2
```

produit

- item 1
- item 2

- liste ordonnée :

```
1. item 1
2. item 2
```

produit

1. item 1
2. item 2

- tableau :

	Col1	Col2	Col3
Row1	1	2	3
Row2	1	2	3

renvoie

	Col1	Col2	Col3
Row1	1	2	3
Row2	1	2	3

- équation latex :

```
$$\int_a^b f(x)dx=1$$
```

renvoie

$$\int_a^b f(x)dx = 1$$

### 1.3.2 Les chunks

Le **code R** doit être écrit dans des **chunks**. On peut insérer des chunks avec :

- la raccourci clavier **Ctrl + Alt + I** (OS X : Cmd + Option + I)
- la bouton **Insert -> R**
- en tapant :

```
```{r}
commandes...
```
```

Plusieurs options peuvent être spécifiés au chunk en fonction de ce que l'on souhaite voir dans le document, par exemple :

- **echo** : TRUEor FALSE pour spécifier si on souhaite afficher le code ;
- **eval** : TRUEor FALSE pour spécifier si le code doit être évalué ou non ;
- **results** : **hide** si on ne veut pas voir la sortie du code.

On pourra trouver l'ensemble des options disponibles sur cette page : <https://yihui.org/knitr/options/>

**Exercice 1.2** (Premier document).

1. Ouvrir un document mardown (*File -> New File -> R Markdown*).
2. Cliquer sur le bouton **Knit** et visualiser la sortie **html**.
3. Obtenir une sortie **pdf**.
4. Modifier le document en créant une section **Cosinus** dans laquelle on tracera la fonction **cosinus**, on pourra utiliser le code suivant dans un **chunk**.

```
> x <- seq(-2*pi,2*pi,by=0.01)
> y <- cos(x)
> plot(x,y,type="l")
```

5. Ajouter une section **Sinus** dans laquelle on tracera la fonction **sinus**.

### 1.3.3 Notebook

L'environnement **notebook** fonctionne exactement comme un document **markdown** mais permet de visualiser la sortie eu format **html** sans avoir à recompiler le document en entier. Cet environnement est donc souvent privilégié pendant la réalisation d'un projet en science des données. Pour créer un notebook, on peut passer par **RStudio** : *File -> New File -> R Notebook* ou simplement remplacer

```
output: html_document
```

par

```
output: html_notebook
```

dans l'entête d'un document **markdown**.

**Exercice 1.3** (Premier document).

Transformer le document markdown de l'exercice précédent en **notebook**. On pourra visualiser la sortie en cliquant sur **Preview**.

#### 1.3.4 Diaporama R

**Rstudio** propose aussi différents environnements pour construire des **diaporamas**. On pourra utiliser le menu *File -> New File -> R Markdown -> Presentation*, puis sélectionner le format *ioslides* ou *slidy*. On utilisera la même syntaxe que pour les documents markdown. Les slides sont séparés par le symbole `##` et les codes R sont toujours insérés dans des chunks.

**Exercice 1.4** (Premier document).

1. Créer 2 diapositives :
  - Titre : **Cosinus** où on tracera la fonction cosinus.
  - Titre : **Sinus** où on tracera la fonction sinus.
2. En modifiant les options des chunks modifier les diapositives de manière à
  - ne pas voir le code R mais voir les graphiques
  - voir uniquement le code R mais pas les graphiques.

#### 1.3.5 Exemples de styles de documents markdown

Par défaut l'entête d'un document **markdown** est de la forme

```
---
```

```
title: "Mon document"
author: "Laurent Rouviere"
date: "6/18/2020"
output: html_document
---
```

Il existe tout un tas d'options qui vont permettre d'améliorer le document final. On peut par exemple ajouter une table des matières avec

```
output:
  html_document:
    toc: true
```

On peut également utiliser des styles prédéfinis en

```
— changeant le thème, voir https://www.datadreaming.org/post/r-markdown-theme-gallery/
output:
  html_document:
    theme: cerulean
```

# Mon document

Laurent Rouvière

6/18/2020

## Section 1

On effectue un **résumé** du jeu de données.

```
summary(cars)

##      speed         dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

## Section2

— utilisant le package **prettydoc** (il faut l'installer), voir <https://github.com/yixuan/prettydoc>

```
output:
  prettydoc::html_pretty:
    theme: cayman
    highlight: github
```

# Mon document

Laurent Rouvière

6/18/2020

## Section 1

On effectue un **résumé** du jeu de données.

```
summary(cars)
```

```
##      speed          dist
## Min.   : 4.0   Min.   :  2.00
## 1st Qu.:12.0   1st Qu.: 26.00
## Median :15.0   Median : 36.00
## Mean    :15.4   Mean    : 42.98
## 3rd Qu.:19.0   3rd Qu.: 56.00
## Max.   :25.0   Max.   :120.00
```

## Section2

- utilisant le package **rmdformats** (à installer aussi), voir <https://github.com/juba/rmdformats>

```
output:
rmdformats::readthedown:
  highlight: kate
```

Mon document

Section 1  
 Section2

# Mon document

## Section 1

On effectue un **résumé** du jeu de données.

```
summary(cars)
```

```
##      speed         dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

Laurent Rouvière  
 6/18/2020

## 2 Les objets R

On commencera par créer un répertoire dans lequel on mettra tous les fichiers du tutoriel. On pourra créer un **projet** avec Rstudio dans lequel on placera ces fichiers (*File -> New Project...*). Par défaut, le répertoire de travail se situera dans le répertoire de ce projet. On peut le vérifier avec la commande

```
> getwd()
```

La commande **setwd** permet de changer le répertoire de travail si besoin. On peut aussi le faire en utilisant le menu **Session -> Set Working directory -> Choose directory....**

### 2.1 Crédit d'objets

#### 2.1.1 Numérique

On crée un **objet R** en assignant une valeur (ou un caractère, vecteur...) avec les opérateurs **<-**, **->**, **=**

```
> b <- 41.3 # assigne la valeur 41.3 à l'objet b
> x <- b    # b est assigné à x
> x = b    # b est assigné à x
> b -> x    # b est assigné à x
> is.numeric(b)
[1] TRUE
> mode(b)
[1] "numeric"
```

#### 2.1.2 Caractère

Les chaînes de caractères sont définies avec des guillemets : "chaine", par exemple

```
> x <- "La mort"
> y <- "aux trousses"
> paste(x,y)
[1] "La mort aux trousses"
```

```
> is.character(x)
[1] TRUE
```

### 2.1.3 Facteur

L'objet facteur est très utile pour travailler avec des variables qualitatives. Cet objet permet d'identifier les modalités prisent par la variable et de travailler dessus, en changeant par exemple le nom d'une modalité :

```
> V1 <- factor(c("less20years", "more50years", "less20years", "more50years", "less20years"))
> V1
[1] less20years more50years less20years more50years less20years
Levels: less20years more50years
> levels(V1)
[1] "less20years" "more50years"
> levels(V1) <- c("Young", "Old")
> V1
[1] Young Old    Young Old    Young
Levels: Young Old
```

### 2.1.4 Logique (Booléen)

```
> x <- TRUE
> is.logical(x)
[1] TRUE
> mode(x)
[1] "logical"
> a <- 1
> a==1
[1] TRUE
> a!=1
[1] FALSE
> a<0
[1] FALSE
> a>0
[1] TRUE
```

## 2.2 Vecteur

On peut définir un vecteur de plusieurs façons :

— fonction collect **c**

```
> x <- c(1.2, 5, 9, 11)
> x
[1] 1.2 5.0 9.0 11.0
```

— opérateur séquence :

```
> 1:5
[1] 1 2 3 4 5
```

— fonction séquence **seq**

```

> seq(1,10,by=2)
[1] 1 3 5 7 9
> seq(0,1,length=10)
[1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
[8] 0.7777778 0.8888889 1.0000000

```

— fonction **rep**

```

> rep(1,4)
[1] 1 1 1 1
> rep(c(1,3),each=3)
[1] 1 1 1 3 3 3

```

On peut aussi créer des vecteurs *caractère* ou *logique*

```

> x <- c("A","B","C")
> x <- rep("A",5)
> paste("X",1:5,sep="-")
[1] "X-1" "X-2" "X-3" "X-4" "X-5"
> substr("statistician",5,9)
[1] "istic"

```

### 2.2.1 Sélectionner une partie d'un vecteur

La sélection s'effectue à l'aide de crochets [ ]

```

> x <- c(-4,-3,1,3,5,8,0)
> x[2]
[1] -3
> x[c(2,5)]
[1] -3 5
> x>0
[1] FALSE FALSE TRUE TRUE TRUE TRUE FALSE
> x[x>0]
[1] 1 3 5 8

```

### 2.2.2 Opérations sur les vecteurs

On peut facilement additionner, multiplier des vecteurs :

```

> x <- seq(-10,10,by=2)
> y <- 1:length(x)
> x+y
[1] -9 -6 -3  0  3  6  9 12 15 18 21
> x*y
[1] -10 -16 -18 -16 -10   0  14  32  54  80 110
> z <- x>0
> x*z
[1]  0  0  0  0  0  0  2  4  6  8 10

```

**Exercice 2.1** (Manipulation de vecteurs).

1. Calculer la moyenne, la somme, la médiane et la variance du vecteur (1,3,8,9,11).

```

> x <- c(1,3,8,9,11)
> mean(x)
[1] 6.4

```

```

> sum(x)
[1] 32
> median(x)
[1] 8
> var(x)
[1] 17.8

```

2. Créer les vecteurs suivants en utilisant la fonction **rep**.

```

vec1 = 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
vec2 = 1 1 1 2 2 2 3 3 3 4 4 4 4 5 5 5
vec3 = 1 1 2 2 2 3 3 3 3 4 4 4 4 4 4

```

```

> rep(1:5,3)
[1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
> rep(1:5,each=3)
[1] 1 1 1 2 2 2 3 3 3 4 4 4 4 5 5 5
> rep(1:4,2:5)
[1] 1 1 2 2 2 3 3 3 3 4 4 4 4 4 4

```

3. Créer le vecteur suivant à l'aide de la fonction **paste**.

```

vec4 = "A0" "A1" "A2" "A3" "A4" "A5" "A6" "A7" "A8" "A9" "A10"

```

```
paste("A",0:10,"",sep="")
```

4. **letters** est un vecteur qui contient les 26 lettres de l'alphabet.

- Trouver le numéro de la lettre *q* (sans compter avec les mains!). On pourra utiliser la fonction **which**.
- Créer le vecteur “a1”,“b2”,… jusqu'à *q* et son index.

```

> index_q <- which(letters=="q")
> paste(letters[1:index_q],1:index_q,sep="")
[1] "a1" "b2" "c3" "d4" "e5" "f6" "g7" "h8" "i9" "j10" "k11" "l12"
[13] "m13" "n14" "o15" "p16" "q17"

```

## 2.3 Matrices

La fonction **matrix** permet de définir des matrices.

```

> m <- matrix(1:4,ncol=2)
> m
[,1] [,2]
[1,]    1    3
[2,]    2    4
> m <- matrix(1:4,nrow=2)
> m
[,1] [,2]
[1,]    1    3
[2,]    2    4
> m <- matrix(1:4,nrow=2,byrow=TRUE)
> dim(m)
[1] 2 2

```

La position d'un élément dans une matrice est indiquée par ses numéros de ligne et de colonne. Ainsi, pour sélectionner le terme de la 2ème ligne et la 1ère colonne, on utilisera

```
> m[2,1]
[1] 3
```

On peut aussi extraire des lignes et des colonnes :

```
> m[1,] #première ligne
[1] 1 2
> m[,2] #deuxième colonne
[1] 2 4
```

Il n'est pas difficile de faire les calculs usuels sur les matrices :

```
> det(m) #déterminant
[1] -2
> solve(m) #inverse
[,1] [,2]
[1,] -2.0 1.0
[2,] 1.5 -0.5
> t(m) #transposé
[,1] [,2]
[1,] 1 3
[2,] 2 4
> n <- matrix(5:8,nrow=2)
> m+n
[,1] [,2]
[1,] 6 9
[2,] 9 12
> m*n #attention : produit de Hadamard
[,1] [,2]
[1,] 5 14
[2,] 18 32
> m%*%n #Produit matriciel
[,1] [,2]
[1,] 17 23
[2,] 39 53
> eigen(m) #Décomposition en valeurs propres
eigen() decomposition
$values
[1] 5.3722813 -0.3722813

$vectors
[,1]      [,2]
[1,] -0.4159736 -0.8245648
[2,] -0.9093767  0.5657675
```

## 2.4 Listes

Une liste est un objet hétérogène. Elle permet de stocker des objets de différents modes dans un même objet. Par exemple, on peut créer une liste qui contient un vecteur et une matrice à l'aide de

```
> mylist <- list(vector=rep(1:5),mat=matrix(1:8,nrow=2))
> mylist
$vector
[1] 1 2 3 4 5
```

```
$mat
 [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> length(mylist)
[1] 2
```

L'extraction s'effectue en indiquant la position de l'objet à extraire dans un **double crochet** [[ ]] :

```
> mylist[[1]]
[1] 1 2 3 4 5
```

On peut aussi utiliser le **nom** de l'élément à extraire :

```
> mylist$mat
 [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> mylist[["mat"]]
 [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

## 2.5 Dataframe

Les dataframes sont des listes particulières dont les composantes ont la même longueur, mais potentiellement des modes différents. C'est l'objet généralement utilisé pour les **tableaux de données** (qui contiennent souvent des variables quantitatives et qualitatives). Par exemple,

```
> name <- c("Paul", "Mary", "Steven", "Charlotte", "Peter")
> sex <- factor(c("M", "F", "M", "F", "M"))
> size <- c(180, 165, 168, 170, 175)
> data <- data.frame(name, sex, size)
> summary(data)
   name          sex         size
Length:5          F:2    Min. :165.0
Class :character M:3    1st Qu.:168.0
Mode  :character                  Median :170.0
                           Mean   :171.6
                           3rd Qu.:175.0
                           Max.  :180.0
```

On observe que **name** est un vecteur de caractères, **sex** un facteur et **size** un vecteur numérique.

L'extraction est similaire aux matrices et aux listes :

```
> data[2,3]
[1] 165
> data[,2]
[1] M F M F M
Levels: F M
> data$sex
[1] M F M F M
Levels: F M
```

## 2.6 Quelques fonctions importantes

- **summary** produit un résumé d'un objet

```
> summary(data)
  name          sex        size
Length:5      F:2   Min.  :165.0
Class :character M:3  1st Qu.:168.0
Mode  :character           Median :170.0
                           Mean   :171.6
                           3rd Qu.:175.0
                           Max.   :180.0
> summary(1:10)
  Min. 1st Qu. Median   Mean 3rd Qu.   Max.
  1.00    3.25   5.50   5.50   7.75   10.00
```

- **mean, sum, median, var, min, max...** (facile à comprendre)

- **sort, order**

```
> x <- c(1,8,5,4)
> sort(x)
[1] 1 4 5 8
> order(x)
[1] 1 4 3 2
```

- **apply** applique une fonction **f** aux lignes ou colonnes d'une matrice ou datafram

```
> V1 <- 1:10
> V2 <- seq(-20,25,length=10)
> df <- data.frame(V1,V2)
> apply(df,1,mean)
[1] -9.5 -6.5 -3.5 -0.5  2.5  5.5  8.5 11.5 14.5 17.5
> apply(df,2,sum)
V1 V2
55 25
```

## 2.7 Exercices complémentaires

**Exercice 2.2** (Manipulation de matrices).

1. Créer la matrice suivante que l'on appellera *mat* (on pourra utiliser les fonctions **rownames** et **colnames**) :

|       | column 1 | column 2 | column 3 | column 4 |
|-------|----------|----------|----------|----------|
| row-1 | 1        | 5        | 5        | 0        |
| row-2 | 0        | 5        | 6        | 1        |
| row-3 | 3        | 0        | 3        | 3        |
| row-4 | 4        | 4        | 4        | 2        |

```
> mat <- matrix(c(1,0,3,4,5,5,0,4,5,6,3,4,0,1,3,2),ncol=4)
> rownames(mat) <- paste("row-",1:4,sep="")
> colnames(mat) <- paste("column ",1:4)
> mat
     column 1 column 2 column 3 column 4
row-1       1       5       5       0
```

```

row-2      0      5      6      1
row-3      3      0      3      3
row-4      4      4      4      2

```

2. Créer un vecteur qui contient la diagonale de **mat**.

```

> diag(mat)
[1] 1 5 3 2

```

3. Créer une matrice qui contient les 2 premières lignes de **mat**.

```

> mat[1:2,]
  column 1 column 2 column 3 column 4
row-1      1      5      5      0
row-2      0      5      6      1

```

4. Créer une matrice qui contient les 2 dernières colonnes de **mat**.

```

> mat[,3:4]
  column 3 column 4
row-1      5      0
row-2      6      1
row-3      3      3
row-4      4      2

```

5. Calculer le déterminant et l'inverse de **mat**.

```

> det(mat)
[1] 60
> solve(mat)
      row-1   row-2       row-3       row-4
column 1  0.5 -0.5  0.1666667 -1.665335e-16
column 2 -0.6  0.4 -0.4666667  5.000000e-01
column 3  0.7 -0.3  0.4333333 -5.000000e-01
column 4 -1.2  0.8 -0.2666667  5.000000e-01

```

**Exercice 2.3** (Manipulation simples sur un jeu de données).

On considère le jeu de données **iris** disponible sous **R** :

```

> data(iris)
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa

```

1. Calculer les moyennes et variances des variables **Sepal.Width** et **Petal.Length**.

```

> mean(iris$Sepal.Width)
[1] 3.057333
> mean(iris$Petal.Length)
[1] 3.758
> var(iris$Sepal.Width)
[1] 0.1899794
> var(iris$Petal.Length)
[1] 3.116278

```

2. Créer un sous jeu de données qui contient uniquement les iris de l'espèce `versicolor`. On appellera ce tableau `iris2`.

```
> test <- iris$Species=="versicolor"
> iris2 <- iris[test,]
```

3. Ordonner les individus dans `iris2` par valeurs décroissantes de la variable `Sepal.Length` (on pourra utiliser la fonction `order`).

```
> ord <- order(iris2$Sepal.Length,decreasing=TRUE)
> iris3 <- iris2[ord,]
```

4. Calculer les valeurs moyennes de `Sepal.Length` pour chaque espèce.

```
> mean(iris[iris$Species=="versicolor","Sepal.Length"])
[1] 5.936
> mean(iris[iris$Species=="virginica","Sepal.Length"])
[1] 6.588
> mean(iris[iris$Species=="setosa","Sepal.Length"])
[1] 5.006
```

5. Ajouter une variable (qu'on appellera `sum.Petal`) dans le dataframe `iris` qui contiendra la somme de `Petal.Length` et `Petal.Width`.

```
> iris$sum.petal <- iris$Petal.Length+iris$Petal.Width
> head(iris)
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species sum.petal
1          5.1         3.5          1.4         0.2   setosa     1.6
2          4.9         3.0          1.4         0.2   setosa     1.6
3          4.7         3.2          1.3         0.2   setosa     1.5
4          4.6         3.1          1.5         0.2   setosa     1.7
5          5.0         3.6          1.4         0.2   setosa     1.6
6          5.4         3.9          1.7         0.4   setosa     2.1
```

#### Exercice 2.4 (Fonction `apply`).

1. Calculer les indicateurs numériques standards (moyenne, min, max, etc.) des 3 variables du jeux de données `ethanol` (disponible dans le package `lattice`).

```
> library(lattice)
> data("ethanol")
> summary(ethanol)
      NOx           C           E
Min. :0.370  Min. : 7.500  Min. :0.5350
1st Qu.:0.953  1st Qu.: 8.625  1st Qu.:0.7618
Median :1.754  Median :12.000  Median :0.9320
Mean   :1.957  Mean   :12.034  Mean   :0.9265
3rd Qu.:3.003  3rd Qu.:15.000  3rd Qu.:1.1098
Max.  :4.028  Max.  :18.000  Max.  :1.2320
> apply(ethanol,2,mean)
      NOx           C           E
1.9573750 12.0340909  0.9264773
```

2. Calculer les quartiles de chaque variables. On pourra faire un `apply` avec la fonction `quantile`.

```
> quantile(ethanol$NOx,probs=c(0.25,0.5,0.75))
  25% 50% 75%
0.9530 1.7545 3.0030
> quantile(ethanol$C,probs=c(0.25,0.5,0.75))
  25% 50% 75%
8.625 12.000 15.000
```

```

> quantile(ethanol$E,probs=c(0.25,0.5,0.75))
  25%    50%    75%
0.76175 0.93200 1.10975
> #ou mieux
> apply(ethanol,2,quantile,probs=c(0.25,0.5,0.75))
      NOx        C        E
25% 0.9530  8.625 0.76175
50% 1.7545 12.000 0.93200
75% 3.0030 15.000 1.10975

```

3. Faire de même pour les déciles.

```

> apply(ethanol,2,quantile,probs=seq(0.1,0.9,by=0.1))
      NOx        C        E
10% 0.6000  7.5 0.6496
20% 0.8030  7.5 0.7206
30% 1.0138  9.0 0.7977
40% 1.4146  9.0 0.8636
50% 1.7545 12.0 0.9320
60% 2.0994 12.6 1.0104
70% 2.7232 15.0 1.0709
80% 3.3326 15.0 1.1404
90% 3.6329 18.0 1.1920

```

### Exercice 2.5 (Données manquantes).

On considère le jeu de données **presidents**

```

> data("presidents")
> df <- matrix(presidents,ncol=4,byrow=T)

```

1. Est-ce que la ligne 20 contient au moins une données manquante ? On pourra utiliser la fonction **any**.

```

> any(is.na(df[20,]))
[1] FALSE

```

2. Quelles sont les lignes de **df** qui contiennent au moins une donnée manquante ? On pourra utiliser la fonction **which**.

```

> which(apply(is.na(df),1,any))
[1] 1 4 8 28

```

3. Supprimer les lignes de **df** qui contiennent au moins une donnée manquante.

```

> ind_sup <- which(apply(is.na(df),1,any))
> df1 <- df[-ind_sup,]
> summary(df1)
      V1          V2          V3          V4
Min. :28.00  Min. :24.00  Min. :24.00  Min. :23.00
1st Qu.:52.50  1st Qu.:49.00  1st Qu.:46.50  1st Qu.:44.50
Median :64.50  Median :60.50  Median :61.00  Median :55.50
Mean   :60.96  Mean   :56.69  Mean   :56.27  Mean   :53.04
3rd Qu.:71.00  3rd Qu.:64.75  3rd Qu.:66.50  3rd Qu.:61.50
Max.   :80.00  Max.   :83.00  Max.   :79.00  Max.   :78.00

```

On aurait aussi pu utiliser directement la fonction **na.omit** :

```

> df2 <- na.omit(df)
> all(df1==df2)
[1] TRUE

```

### 3 Manipuler les données avec dplyr

Les données sont souvent le point de départ d'une étude statistique. Elles sont généralement d'abord stockées dans des fichiers (txt, xls, csv) et une des premières étapes est d'amener ces données dans **R** et de les affecter à un objet de type **dataframe**. Il est par conséquent très important connaître et maîtriser les opérations qui permettent de réaliser ces importations. Nous présentons tout d'abord quelques fonctions qui permettent de faire ces importations avant de présenter le package **dplyr** qui offre une syntaxe claire pour manipuler des données (correctement importées).

#### 3.1 Importer des données

Les fonction **read.table** et **read.csv** sont les fonctions standards de *R* pour importer des données à partir de fichiers *.txt* ou *.csv*. Il est important de bien gérer le chemin du répertoire où se trouve le fichier. On peut le spécifier explicitement ou utiliser des fonctions comme **file.path** :

```
> path <- file.path("data/", "piscines.csv") #premier : répertoire, deuxième : fichier
> piscines <- read.csv(path)
> class(piscines)
[1] "data.frame"
> summary(piscines)
      Name          Address         Latitude        Longitude 
Length:20      Length:20      Min.   :-27.61    Min.   :152.9 
Class :character  Class :character  1st Qu.:-27.55   1st Qu.:153.0 
Mode  :character  Mode  :character Median :-27.49    Median :153.0 
                           Mean  :-27.49    Mean   :153.0 
                           3rd Qu.:-27.45   3rd Qu.:153.1 
                           Max.  :-27.31    Max.   :153.2
```

Il existe plusieurs **options importantes** dans **read.csv**, notamment

- **sep** : le caractère de séparation (espace, virgule...)
- **dec** : le caractère pour le séparateur décimal (vigule, point...)
- **header** : logique pour indiquer si le nom des variables est spécifié à la première ligne du fichier
- **row.names** : vecteurs des identifiants (si besoin)
- **na.strings** : vecteur de caractères pour repérer les données manquantes.
- ...

Le package **readr** du tidyverse propose d'autres fonctions comme **read\_csv** ou **read\_delim**. Il n'y a pas de différences énormes avec les fonctions standards, les objets créés sont des **tibbles** et plus des **dataframes** (même si les tibbles sont des dataframes...). Par exemple

```
> library(readr)
> piscines <- read_csv("data/piscines.csv")
> summary(piscines)
      Name          Address         Latitude        Longitude 
Length:20      Length:20      Min.   :-27.61    Min.   :152.9 
Class :character  Class :character  1st Qu.:-27.55   1st Qu.:153.0 
Mode  :character  Mode  :character Median :-27.49    Median :153.0 
                           Mean  :-27.49    Mean   :153.0 
                           3rd Qu.:-27.45   3rd Qu.:153.1 
                           Max.  :-27.31    Max.   :153.2

> class(piscines)
[1] "spec_tbl_df" "tbl_df"       "tbl"          "data.frame"
```

Enfin si on n'est pas très à l'aise avec ces fonctions, on pourra utiliser le bouton **Import Dataset** qui se trouve dans l'onglet **Environment** de **RStudio**. Cette manière de procéder fonctionne pour des jeux de données

simples, si les bases de données contient trop de spécificités on devra souvent utiliser les fonctions avec les bonnes options.

**Exercice 3.1** (Importation d'un jeu de données).

1. Importer les données qui se trouvent dans le fichier *mydata.csv*. On utilisera la fonction **read.csv** avec les options par défaut.

```
> data1 <- read.csv("data/mydata.csv")
> summary(data1)
surname.height.weight.feet_size.sex
Length:3
Class :character
Mode  :character
```

2. Importer **correctement** les données qui se trouvent dans le fichier *mydata.csv* (utiliser *sep*, *dec* et *row.names*).

```
> data1 <- read.csv("data/mydata.csv",sep=";",dec=c("."),row.names = 1)
> summary(data1)
    height          weight         feet_size        sex
Min.   :158.0   Min.   :72.00   Min.   :8.000   Length:3
1st Qu.:166.8   1st Qu.:75.00   1st Qu.:8.250   Class  :character
Median :175.5   Median :78.00   Median :8.500   Mode   :character
Mean   :172.5   Mean   :76.67   Mean   :8.667
3rd Qu.:179.8   3rd Qu.:79.00   3rd Qu.:9.000
Max.   :184.0   Max.   :80.00   Max.   :9.500
```

3. Importer les données qui se trouvent dans le fichier *mydata2.csv*.

```
> data2 <- read.csv("data/mydata2.csv")
> summary(data2)
height.weight.size.sex
Length:4
Class :character
Mode  :character
```

4. Ce fichier contient des données manquantes (identifiées par un point). A l'aide de **na.strings**, refaire l'importation en identifiant les données manquantes.

```
> data2 <- read.csv("data/mydata2.csv",sep="",na.strings = ".")
> summary(data2)
    height          weight         size        sex
Min.   :175.5   Min.   :72.00   Min.   :7.00   Length:4
1st Qu.:176.8   1st Qu.:75.00   1st Qu.:7.75   Class  :character
Median :178.0   Median :78.00   Median :8.25   Mode   :character
Mean   :179.2   Mean   :76.67   Mean   :8.25
3rd Qu.:181.0   3rd Qu.:79.00   3rd Qu.:8.75
Max.   :184.0   Max.   :80.00   Max.   :9.50
NA's   :1       NA's   :1
```

5. Changer les **levels** de la variable **sex** en **woman** et **man** (on pourra utiliser la fonction **levels**).

```
> data22 <- data2
```

— 1ère façon :

```
> levels(data2$sex) <- c("woman","man")
> summary(data2)
    height          weight         size        sex
Min.   :175.5   Min.   :72.00   Min.   :7.00   Length:4
1st Qu.:176.8   1st Qu.:75.00   1st Qu.:7.75   Class  :character
```

```

Median :178.0   Median :78.00   Median :8.25   Mode  :character
Mean   :179.2   Mean   :76.67   Mean   :8.25
3rd Qu.:181.0   3rd Qu.:79.00   3rd Qu.:8.75
Max.   :184.0   Max.   :80.00   Max.   :9.50
NA's    :1       NA's    :1

```

— 2ème façon avec `recode_factor`

```

> library(tidyverse)
> data22$sex <- recode_factor(data22$sex, "F"="woman", "M"="man")
> summary(data22)
  height      weight      size      sex
Min.   :175.5   Min.   :72.00   Min.   :7.00   woman:1
1st Qu.:176.8   1st Qu.:75.00   1st Qu.:7.75   man   :3
Median :178.0   Median :78.00   Median :8.25
Mean   :179.2   Mean   :76.67   Mean   :8.25
3rd Qu.:181.0   3rd Qu.:79.00   3rd Qu.:8.75
Max.   :184.0   Max.   :80.00   Max.   :9.50
NA's    :1       NA's    :1

```

**Exercice 3.2** (Jointure de tables).

On considère les 3 jeux de données suivants, au format `tibble` :

```

> df1 <- tibble(name=c("Mary", "Peter", "John", "July"), age=c(18, 25, 21, 43))
> df2 <- tibble(name=c("Zac", "Julian"), age=c(23, 48))
> df3 <- tibble(size=c(154, 178, 182, 134, 142), name1=c("Peter", "Mary", "July", "John", "stef"))
> df1
# A tibble: 4 x 2
  name     age
  <chr> <dbl>
1 Mary     18
2 Peter    25
3 John     21
4 July     43
> df2
# A tibble: 2 x 2
  name     age
  <chr> <dbl>
1 Zac      23
2 Julian   48
> df3
# A tibble: 5 x 2
  size name1
  <dbl> <chr>
1 154 Peter
2 178 Mary
3 182 July
4 134 John
5 142 stef

```

On souhaite assembler ces tables en utilisant les *fonctions de jointure* du `tidyverse` (`left_join`, `full_join` par exemple). On pourra consulter la cheatsheet **Data transformation with dplyr** (`help -> cheatsheets -> ...`).

1. Assembler `df1` avec `df2` en utilisant `bind_rows` et calculer la moyenne de la variable `age`. On appellera `df` cette nouvelle table.

```

> df <- bind_rows(df1,df2)
> mean(df$age)
[1] 29.66667

```

2. Assembler df avec df3 en utilisant **full\_join**.

```

> a1 <- full_join(df,df3,by=c("name"="name1"))
> a1
# A tibble: 7 x 3
  name     age   size
  <chr> <dbl> <dbl>
1 Mary      18    178
2 Peter     25    154
3 John      21    134
4 July      43    182
5 Zac       23     NA
6 Julian    48     NA
7 stef      NA    142

```

3. Faire la même chose avec **inner\_join**.

```

> a2 <- inner_join(df,df3,by=c("name"="name1"))
> a2
# A tibble: 4 x 3
  name     age   size
  <chr> <dbl> <dbl>
1 Mary      18    178
2 Peter     25    154
3 John      21    134
4 July      43    182

```

4. Expliquer les différences entre **full\_join** et **inner\_join**.

**inner\_join** retient uniquement les individus pour lesquels **age** et **size** ont été observés. **full\_join** garde tous les individus, des **NA** sont ajoutées lorsque la variable n'est pas observée.

## 3.2 Le package dplyr

**dplyr** est un package du **tidyverse** qui permet de faciliter la manipulation des données. Il propose une **syntaxe claire** (basée sur une **grammaire**) pour manipuler les données. On pourra trouver des informations [ici](#) ou sur la [cheatsheet](#).

Nous avons vu quelques opérations standards pour manipuler les données. Par exemple, on peut obtenir les **Longitude** et **Latitude** des piscines ayant une *Longitude* supérieure à 153 avec

```

> piscines[piscines$Longitude > 153,c("Longitude","Latitude")]
# A tibble: 16 x 2
  Longitude Latitude
  <dbl>     <dbl>
1     153.    -27.6
2     153.    -27.5
3     153.    -27.4
4     153.    -27.5
5     153.    -27.5
6     153.    -27.5
7     153.    -27.6
8     153.    -27.5
9     153.    -27.5
10    153.    -27.5

```

```

11      153.    -27.5
12      153.    -27.4
13      153.    -27.6
14      153.    -27.3
15      153.    -27.5
16      153.    -27.5

```

**dplyr** propose de faire la même chose avec une syntaxe plus claire

```

> library(tidyverse) #ou library(dplyr)
> piscines %>% select(Longitude, Latitude) %>% filter(Longitude > 153)
# A tibble: 16 x 2
  Longitude Latitude
     <dbl>     <dbl>
1      153.    -27.6
2      153.    -27.5
3      153.    -27.4
4      153.    -27.5
5      153.    -27.5
6      153.    -27.5
7      153.    -27.6
8      153.    -27.5
9      153.    -27.5
10     153.    -27.5
11     153.    -27.5
12     153.    -27.4
13     153.    -27.6
14     153.    -27.3
15     153.    -27.5
16     153.    -27.5

```

Le code est plus efficace et facile à lire.

**dplyr** propose une **grammaire** dont les principaux **verbes** sont :

- **select()** : sélectionner des colonnes (variables)
- **filter()** : filtre des lignes (individus)
- **arrange()** : ordonner des lignes
- **mutate()** : créer des nouvelles colonnes (nouvelles variables)
- **summarise()** : calculer des résumés numériques (ou résumés statistiques)
- **group\_by()** : effectuer des opérations pour des groupes d'individus

Nous les présentons dans la partie suivante.

### 3.2.1 Les principaux verbes dplyr

**3.2.1.1 Le verbe `select()`** Il permet de sélectionner des variables (colonnes) :

```
> select(df, VAR1, VAR2, ...)
```

Par exemple,

```

> coord <- select(piscines, Latitude, Longitude)
> head(piscines, n=2)
# A tibble: 2 x 4
  Name          Address          Latitude  Longitude
  <chr>        <chr>           <dbl>      <dbl>
1 <NA>          <NA>            -27.5      153.
2 <NA>          <NA>            -27.5      153.

```

```

1 Acacia Ridge Leisure Cent~ 1391 Beaudesert Road, Acacia Ri~ -27.6      153.
2 Bellbowrie Pool           Sugarwood Street, Bellbowrie   -27.6      153.
> head(coord, n=2)
# A tibble: 2 x 2
  Latitude Longitude
    <dbl>     <dbl>
1 -27.6      153.
2 -27.6      153.

```

On peut utiliser les **helper functions** (`begins_with`, `end_with`, `contains`, `matches`) pour des sélections plus précises basées sur le nom des variables.

```

> coord <- select(piscines, ends_with("tude"))
> head(coord, n=2)
# A tibble: 2 x 2
  Latitude Longitude
    <dbl>     <dbl>
1 -27.6      153.
2 -27.6      153.

```

### 3.2.1.2 Le verbe `mutate()`

Il permet de créer des nouvelles variables

```
> mutate(df, NEW.VAR = expression(VAR1, VAR2, ...))
```

Par exemple

```

> df <- mutate(piscines, phrase=paste("Swimming pool", Name, "is located at the address", Address))
> select(df,phrase)
# A tibble: 20 x 1
  phrase
  <chr>
1 Swimming pool Acacia Ridge Leisure Centre is located at the address 1391 Bea~
2 Swimming pool Bellbowrie Pool is located at the address Sugarwood Street, Be~
3 Swimming pool Carole Park is located at the address Cnr Boundary Road and Wa~
4 Swimming pool Centenary Pool (inner City) is located at the address 400 Greg~
5 Swimming pool Chermside Pool is located at the address 375 Hamilton Road, Ch~
6 Swimming pool Colmslie Pool (Morningside) is located at the address 400 Lytt~
7 Swimming pool Spring Hill Baths (inner City) is located at the address 14 To~
8 Swimming pool Dunlop Park Pool (Corinda) is located at the address 794 Oxley~
9 Swimming pool Fortitude Valley Pool is located at the address 432 Wickham St~
10 Swimming pool Hibiscus Sports Complex (upper MtGravatt) is located at the ad~
11 Swimming pool Ithaca Pool ( Paddington) is located at the address 131 Caxton~
12 Swimming pool Jindalee Pool is located at the address 11 Yallambee Road, Jin~
13 Swimming pool Manly Pool is located at the address 1 Fairlead Crescent, Manly
14 Swimming pool Mt Gravatt East Aquatic Centre is located at the address Cnr w~
15 Swimming pool Musgrave Park Pool (South Brisbane) is located at the address ~
16 Swimming pool Newmarket Pool is located at the address 71 Alderson Stret, Ne~
17 Swimming pool Runcorn Pool is located at the address 37 Bonemill Road, Runco~
18 Swimming pool Sandgate Pool is located at the address 231 Flinders Parade, S~
19 Swimming pool Langlands Parks Pool (Stones Corner) is located at the address~
20 Swimming pool Yeronga Park Pool is located at the address 81 School Road, Ye~

```

On peut également créer plusieurs variables avec un seul `mutate` :

```
> mutate(piscines,
+       phrase = paste("Swimming pool", Name, "is located at the address", Address),
```

```

+     unused = Longitude + Latitude
+
# A tibble: 20 x 6
  Name      Address    Latitude Longitude phrase      unused
  <chr>    <chr>        <dbl>     <dbl> <chr>      <dbl>
1 Acacia Ridge~ 1391 Beaudeser~ -27.6      153. Swimming pool Acacia~ 125.
2 Bellbowrie P~ Sugarwood Stre~ -27.6      153. Swimming pool Bellbo~ 125.
3 Carole Park   Cnr Boundary R~ -27.6      153. Swimming pool Carole~ 125.
4 Centenary Po~ 400 Gregory Te~ -27.5      153. Swimming pool Centen~ 126.
5 Chermside Po~ 375 Hamilton R~ -27.4      153. Swimming pool Cherms~ 126.
6 Colmslie Poo~ 400 Lytton Roa~ -27.5      153. Swimming pool Colmsl~ 126.
7 Spring Hill ~ 14 Torrington ~ -27.5      153. Swimming pool Spring~ 126.
8 Dunlop Park ~ 794 Oxley Road~ -27.5      153. Swimming pool Dunlop~ 125.
9 Fortitude Va~ 432 Wickham St~ -27.5      153. Swimming pool Fortit~ 126.
10 Hibiscus Spo~ 90 Klumpp Road~ -27.6      153. Swimming pool Hibisc~ 126.
11 Ithaca Pool ~ 131 Caxton Str~ -27.5      153. Swimming pool Ithaca~ 126.
12 Jindalee Pool 11 Yallambee R~ -27.5      153. Swimming pool Jindal~ 125.
13 Manly Pool    1 Fairlead Cre~ -27.5      153. Swimming pool Manly ~ 126.
14 Mt Gravatt E~ Cnr wecker Roa~ -27.5      153. Swimming pool Mt Gra~ 126.
15 Musgrave Par~ 100 Edmonstone~ -27.5      153. Swimming pool Musgra~ 126.
16 Newmarket Po~ 71 Alderson St~ -27.4      153. Swimming pool Newmar~ 126.
17 Runcorn Pool  37 Bonemill Ro~ -27.6      153. Swimming pool Runcor~ 125.
18 Sandgate Pool 231 Flinders P~ -27.3      153. Swimming pool Sandga~ 126.
19 Langlands Pa~ 5 Panitya Stre~ -27.5      153. Swimming pool Langla~ 126.
20 Yeronga Park~ 81 School Road~ -27.5      153. Swimming pool Yerong~ 125.

```

**3.2.1.3 Le verbe filter()** Il permet de sélectionner des individus (lignes) :

```
> filter(df, TEST)
```

Par exemple

```

> p1 <- filter(piscines, Longitude>153.02)
> select(p1,Longitude)
# A tibble: 12 x 1
  Longitude
  <dbl>
1 153.
2 153.
3 153.
4 153.
5 153.
6 153.
7 153.
8 153.
9 153.
10 153.
11 153.
12 153.

```

ou (on sélectionne les piscines dont le nom contient **Pool**)

```

> df <- filter(piscines, !grepl("Pool", Name))
> select(df,Name)
# A tibble: 5 x 1

```

```

Name
<chr>
1 Acacia Ridge Leisure Centre
2 Carole Park
3 Spring Hill Baths (inner City)
4 Hibiscus Sports Complex (upper MtGravatt)
5 Mt Gravatt East Aquatic Centre

```

ou (on sélectionne les piscines avec une longitude plus grande que 153.02 ou une latitude plus petite que -27.488)

```

> p2 <- filter(piscines, Longitude > 153.02 | Latitude < -27.488)
> select(p2, Longitude, Latitude)
# A tibble: 17 x 2
   Longitude Latitude
      <dbl>     <dbl>
1      153.    -27.6
2      153.    -27.6
3      153.    -27.6
4      153.    -27.5
5      153.    -27.4
6      153.    -27.5
7      153.    -27.5
8      153.    -27.5
9      153.    -27.5
10     153.    -27.6
11     153.    -27.5
12     153.    -27.5
13     153.    -27.5
14     153.    -27.6
15     153.    -27.3
16     153.    -27.5
17     153.    -27.5

```

On peut également utiliser la fonction `slice` pour choisir des individus à partir de leurs indices :

```

> slice(piscines, 5:8)
# A tibble: 4 x 4
  Name          Address           Latitude Longitude
  <chr>        <chr>            <dbl>     <dbl>
1 Chermside Pool 375 Hamilton Road, Chermside  -27.4      153.
2 Colmslie Pool (Morningside) 400 Lytton Road, Morningside -27.5      153.
3 Spring Hill Baths (inner Ci~ 14 Torrington Street, Springh~-27.5      153.
4 Dunlop Park Pool (Corinda)  794 Oxley Road, Corinda   -27.5      153.

```

**3.2.1.4 Le verbe `arrange()`** Il permet d'ordonner les individus en fonction d'une variable

```
> arrange(df, VAR) #tri croissant
```

ou

```
> arrange(df, desc(VAR)) #tri décroissant
```

Par exemple

```
> arrange(piscines, Longitude)
# A tibble: 20 x 4
```

|    | Name                         | Address                       | Latitude | Longitude |
|----|------------------------------|-------------------------------|----------|-----------|
|    | <chr>                        | <chr>                         | <dbl>    | <dbl>     |
| 1  | Bellbowrie Pool              | Sugarwood Street, Bellbowrie  | -27.6    | 153.      |
| 2  | Carole Park                  | Cnr Boundary Road and Waterf~ | -27.6    | 153.      |
| 3  | Jindalee Pool                | 11 Yallambee Road, Jindalee   | -27.5    | 153.      |
| 4  | Dunlop Park Pool (Corinda)   | 794 Oxley Road, Corinda       | -27.5    | 153.      |
| 5  | Newmarket Pool               | 71 Alderson Stret, Newmarket  | -27.4    | 153.      |
| 6  | Ithaca Pool ( Paddington)    | 131 Caxton Street, Paddington | -27.5    | 153.      |
| 7  | Musgrave Park Pool (South B~ | 100 Edmonstone Street, South~ | -27.5    | 153.      |
| 8  | Yeronga Park Pool            | 81 School Road, Yeronga       | -27.5    | 153.      |
| 9  | Spring Hill Baths (inner Ci~ | 14 Torrington Street, Spring~ | -27.5    | 153.      |
| 10 | Centenary Pool (inner City)  | 400 Gregory Terrace, Spring ~ | -27.5    | 153.      |
| 11 | Acacia Ridge Leisure Centre  | 1391 Beaudesert Road, Acacia~ | -27.6    | 153.      |
| 12 | Chermside Pool               | 375 Hamilton Road, Chermside  | -27.4    | 153.      |
| 13 | Fortitude Valley Pool        | 432 Wickham Street, Fortitud~ | -27.5    | 153.      |
| 14 | Langlands Parks Pool (Stone~ | 5 Panitya Street, Stones Cor~ | -27.5    | 153.      |
| 15 | Sandgate Pool                | 231 Flinders Parade, Sandgate | -27.3    | 153.      |
| 16 | Hibiscus Sports Complex (up~ | 90 Klumpp Road, Upper Mount ~ | -27.6    | 153.      |
| 17 | Runcorn Pool                 | 37 Bonemill Road, Runcorn     | -27.6    | 153.      |
| 18 | Colmslie Pool (Morningside)  | 400 Lytton Road, Morningside  | -27.5    | 153.      |
| 19 | Mt Gravatt East Aquatic Cen~ | Cnr wecker Road and Newnham ~ | -27.5    | 153.      |
| 20 | Manly Pool                   | 1 Fairlead Crescent, Manly    | -27.5    | 153.      |

ou

|    | Name                         | Address                       | Latitude | Longitude |
|----|------------------------------|-------------------------------|----------|-----------|
|    | <chr>                        | <chr>                         | <dbl>    | <dbl>     |
| 1  | Manly Pool                   | 1 Fairlead Crescent, Manly    | -27.5    | 153.      |
| 2  | Mt Gravatt East Aquatic Cen~ | Cnr wecker Road and Newnham ~ | -27.5    | 153.      |
| 3  | Colmslie Pool (Morningside)  | 400 Lytton Road, Morningside  | -27.5    | 153.      |
| 4  | Runcorn Pool                 | 37 Bonemill Road, Runcorn     | -27.6    | 153.      |
| 5  | Hibiscus Sports Complex (up~ | 90 Klumpp Road, Upper Mount ~ | -27.6    | 153.      |
| 6  | Sandgate Pool                | 231 Flinders Parade, Sandgate | -27.3    | 153.      |
| 7  | Langlands Parks Pool (Stone~ | 5 Panitya Street, Stones Cor~ | -27.5    | 153.      |
| 8  | Fortitude Valley Pool        | 432 Wickham Street, Fortitud~ | -27.5    | 153.      |
| 9  | Chermside Pool               | 375 Hamilton Road, Chermside  | -27.4    | 153.      |
| 10 | Acacia Ridge Leisure Centre  | 1391 Beaudesert Road, Acacia~ | -27.6    | 153.      |
| 11 | Centenary Pool (inner City)  | 400 Gregory Terrace, Spring ~ | -27.5    | 153.      |
| 12 | Spring Hill Baths (inner Ci~ | 14 Torrington Street, Spring~ | -27.5    | 153.      |
| 13 | Yeronga Park Pool            | 81 School Road, Yeronga       | -27.5    | 153.      |
| 14 | Musgrave Park Pool (South B~ | 100 Edmonstone Street, South~ | -27.5    | 153.      |
| 15 | Ithaca Pool ( Paddington)    | 131 Caxton Street, Paddington | -27.5    | 153.      |
| 16 | Newmarket Pool               | 71 Alderson Stret, Newmarket  | -27.4    | 153.      |
| 17 | Dunlop Park Pool (Corinda)   | 794 Oxley Road, Corinda       | -27.5    | 153.      |
| 18 | Jindalee Pool                | 11 Yallambee Road, Jindalee   | -27.5    | 153.      |
| 19 | Carole Park                  | Cnr Boundary Road and Waterf~ | -27.6    | 153.      |
| 20 | Bellbowrie Pool              | Sugarwood Street, Bellbowrie  | -27.6    | 153.      |

### 3.2.2 Les verbes summarize et group\_by

Les verbes précédents permettent de manipuler les données en sélectionnant des individus ou variables essentiellement. Ces deux nouveaux verbes vont permettre de calculer des indicateurs statistiques sur un jeu

de données.

**3.2.2.1 Le verbe `summarise` (ou `summarise`)** Il permet de créer des nouveaux jeux de données qui contiennent des résumés statistiques du jeu de données initial comme la moyenne, variance, médiane de variables. Par exemple

```
> summarise(piscines,
+             mean_long = mean(Longitude),
+             med_lat = median(Latitude),
+             min_lat = min(Latitude),
+             sum_long = sum(Longitude)
+ )
# A tibble: 1 x 4
  mean_long med_lat min_lat sum_long
    <dbl>     <dbl>     <dbl>     <dbl>
1      153.    -27.5    -27.6     3061.
```

`dplyr` contient également les fonction suivantes (souvent utilisées en statistique) :

1. `n()` : nombre de lignes (individus d'un jeu de données).
2. `n_distinct()` : nombre d'éléments distincts dans un vecteur.
3. `first()` et `last()` : premier et dernier élément d'un vecteur.

Par exemple, on obtient le nombre de piscines dans le jeu de données, et la longitude de la dernière piscine avec

```
> summarise(piscines,n())
# A tibble: 1 x 1
`n()`
<int>
1     20
> summarise(piscines,last(Longitude))
# A tibble: 1 x 1
`last(Longitude)`
<dbl>
1      153.
```

On peut aussi utiliser `summarise_all`, `summarise_at` qui vont permettre de répéter les mêmes opérations sur plusieurs variables. Par exemple

```
> summarise_at(piscines,3:4,mean)
# A tibble: 1 x 2
 Latitude Longitude
   <dbl>     <dbl>
1    -27.5      153.
```

**3.2.2.2 Regrouper des données avec ‘Group\_by’** `group_by` permet d'appliquer une ou des opérations à des groupes de données (ou d'individus). Par exemple, imaginons que l'on souhaite calculer les longitudes moyennes des piscines scindées en 2 groupes : faible et large latitude. On crée d'abord une variable `lat_dis` qui permet d'identifier les latitudes (faible ou large) :

```
> lat_mean <- piscines %>% summarise(mean(Latitude))
> pisc1 <- piscines %>% mutate(lat_dis=factor(Latitude>as.numeric(lat_mean)))
> levels(pisc1$lat_dis) <- c("Low","High")
```

Il reste maintenant à utiliser `group_by` pour obtenir les longitudes moyennes des 2 groupes :

```
> summarise(group_by(pisc1, lat_dis), mean_long=mean(Longitude))
# A tibble: 2 x 2
  lat_dis mean_long
  <fct>     <dbl>
1 Low        153.
2 High       153.
```

### 3.2.3 Assembler des verbes avec l'opérateur de chainage %>%

Un des principaux intérêts de **dplyr** est bien entendu d'utiliser plusieurs verbes pour arriver au résultat souhaité. C'est ce qui est fait plus haut et nous observons que la syntaxe n'est pas facile à lire. Le package propose un **opérateur de chainage** ou **pipe opérateur** qui permet de rentre cette syntaxe plus lisible. Cet opérateur consiste à décomposer le code étape par étape et à relier ces étapes par le symbole `%>%`. On peut par exemple réécrire l'exemple précédent avec :

1. Le jeu de données

```
> pisc1
```

2. Étape `group_by`

```
> pisc1 %>% group_by(lat_dis)
```

3. Étape `summarise`

```
> pisc1 %>% group_by(lat_dis) %>% summarise(mean_long=mean(Longitude))
# A tibble: 2 x 2
  lat_dis mean_long
  <fct>     <dbl>
1 Low        153.
2 High       153.
```

qui donne le résultat souhaité.

Cet opérateur peut être utilisé pour toutes les fonctions **R**. Il revient à considérer comme premier argument du terme à droite du pipe le terme à gauche de ce dernier. Par exemple

```
> mean(1:10)
[1] 5.5
> 1:10 %>% mean()
[1] 5.5
```

Il est recommandé d'utiliser cet opérateur lorsque on chaîne les verbes **dplyr**, la syntaxe est beaucoup plus claire.

### 3.2.4 Quelques exercices

**Exercice 3.3** (Dplyr sur les iris de Fisher).

On considère le jeu de données `iris`

```
> iris <- iris %>% as_tibble()
```

Répondre aux questions suivantes en utilisant les verbes **dplyr** et l'opérateur `%>%`.

1. Sélectionner les variables `Petal.Width` et `Species`.

```
> iris %>% select(Petal.Width, Species)
# A tibble: 150 x 2
  Petal.Width Species
  <dbl> <fct>
```

```

1      0.2 setosa
2      0.2 setosa
3      0.2 setosa
4      0.2 setosa
5      0.2 setosa
6      0.4 setosa
7      0.3 setosa
8      0.2 setosa
9      0.2 setosa
10     0.1 setosa
# ... with 140 more rows

```

2. Construire une table qui contient uniquement les iris d'espèce `versicolor` ou `virginica` (on pourra utiliser le symbole `|` pour la condition **ou**).

```

> iris %>% filter(Species=="versicolor" | Species=="virginica")
# A tibble: 100 x 6
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species   sum.petal
        <dbl>       <dbl>       <dbl>       <dbl> <fct>       <dbl>
1         7         3.2        4.7        1.4 versicolor    6.1 
2         6.4        3.2        4.5        1.5 versicolor    6    
3         6.9        3.1        4.9        1.5 versicolor    6.4 
4         5.5        2.3        4          1.3 versicolor    5.3 
5         6.5        2.8        4.6        1.5 versicolor    6.1 
6         5.7        2.8        4.5        1.3 versicolor    5.8 
7         6.3        3.3        4.7        1.6 versicolor    6.3 
8         4.9        2.4        3.3        1    versicolor    4.3 
9         6.6        2.9        4.6        1.3 versicolor    5.90 
10        5.2        2.7        3.9        1.4 versicolor    5.3 
# ... with 90 more rows

```

3. Calculer le nombre d'iris de l'espèce `setosa` en utilisant `summarise`.

```

> iris %>% filter(Species=="setosa") %>% summarise(n())
# A tibble: 1 x 1
`n()`
<int>
1     50

```

4. Calculer la moyenne de la variable `Petal.Width` pour les iris de l'espèce `versicolor`.

```

> iris %>% filter(Species=="versicolor") %>%
+   summarise(Mean_PW=mean(Petal.Width))
# A tibble: 1 x 1
  Mean_PW
        <dbl>
1     1.33

```

5. Ajouter dans le jeu de données la variable `Sum_Petal` qui correspond à la somme de `Petal.Width` et `Sepal.Width`.

```

> iris1 <- iris
> iris1 %>% mutate(Sum_Petal=Petal.Width+Sepal.Width)
# A tibble: 150 x 7
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species   sum.petal Sum_Petal
        <dbl>       <dbl>       <dbl>       <dbl> <fct>       <dbl>       <dbl>
1         5.1         3.5        1.4        0.2 setosa      1.60       3.7 
2         4.9         3          1.4        0.2 setosa      1.60       3.2 
3         4.7         3.2        1.3        0.2 setosa      1.5        3.4 

```

```

4      4.6      3.1      1.5      0.2 setosa     1.7      3.3
5      5.0      3.6      1.4      0.2 setosa     1.60     3.8
6      5.4      3.9      1.7      0.4 setosa     2.1      4.3
7      4.6      3.4      1.4      0.3 setosa     1.7      3.7
8      5.0      3.4      1.5      0.2 setosa     1.7      3.6
9      4.4      2.9      1.4      0.2 setosa     1.60     3.1
10     4.9      3.1      1.5      0.1 setosa     1.6      3.2
# ... with 140 more rows

```

6. Calculer la moyenne et la variance de la variable Sepal.Length pour chaque espèce (on pourra utiliser `group_by`).

```

> iris %>% group_by(Species) %>%
+   summarise(mean_PL=mean(Petal.Length),var_PL=var(Petal.Length)) %>%
+   mutate(var_PL=round(var_PL,3))
# A tibble: 3 x 3
  Species    mean_PL  var_PL
  <fct>      <dbl>   <dbl>
1 setosa     1.46    0.03
2 versicolor 4.26    0.221
3 virginica  5.55    0.305

```

### Exercice 3.4 (Trafic aérien aux USA).

On considère la table `hflights` qui contient des informations sur les vols au départ des aéroports Houston airports IAH (George Bush Intercontinental) et HOU (Houston Hobby) :

```

> library(hflights)
> hflights <- as_tibble(hflights)

```

La variable Unique Carrier renseigne sur la compagnie du vol. On recode cette variable afin que la compagnie soit plus explicite :

```

> lut1 <- c("AA" = "American", "AS" = "Alaska", "B6" = "JetBlue", "CO" = "Continental",
+         "DL" = "Delta", "OO" = "SkyWest", "UA" = "United", "US" = "US_Airways",
+         "WN" = "Southwest", "EV" = "Atlantic_Southeast", "F9" = "Frontier",
+         "FL" = "AirTran", "MQ" = "American_Eagle", "XE" = "ExpressJet", "YV" = "Mesa")

```

On fait de même pour la variable CancellationCode :

```

> lut2 <- c("A" = "carrier", "B" = "weather", "C" = "FFA", "D" = "security", "E" = "not cancelled")

```

On effectue maintenant les changements dans la table pour obtenir une nouvelle version de `hflights` :

```

> hflights1 <- hflights
> hflights1$UniqueCarrier <- lut1[hflights1$UniqueCarrier]
> hflights1$CancellationCode[hflights1$CancellationCode==""] <- "Z"
> hflights1$CancellationCode <- lut2[hflights1$CancellationCode]

```

A partir de maintenant, on travaille avec `hflights1`.

1. Sélectionner les variables qui se situent entre Origin et Cancelled de différentes façons.

```

> ind <- match(c("Origin","Cancelled"),names(hflights1))
> hflights1 %>% select(seq(ind[1],ind[2]))
# A tibble: 227,496 x 6
  Origin Dest Distance TaxiIn TaxiOut Cancelled
  <chr>  <chr>    <int>   <int>   <int>    <int>
1 IAH    DFW      224      7       13        0

```

```

2 IAH DFW 224 6 9 0
3 IAH DFW 224 5 17 0
4 IAH DFW 224 9 22 0
5 IAH DFW 224 9 9 0
6 IAH DFW 224 6 13 0
7 IAH DFW 224 12 15 0
8 IAH DFW 224 7 12 0
9 IAH DFW 224 8 22 0
10 IAH DFW 224 6 19 0
# ... with 227,486 more rows
> #ou
> hflights1 %>% select(Origin:Cancelled)
# A tibble: 227,496 x 6
  Origin Dest Distance TaxiIn TaxiOut Cancelled
  <chr>  <chr>    <int>   <int>   <int>   <int>
1 IAH    DFW     224      7      13      0
2 IAH    DFW     224      6      9       0
3 IAH    DFW     224      5      17      0
4 IAH    DFW     224      9      22      0
5 IAH    DFW     224      9       9      0
6 IAH    DFW     224      6      13      0
7 IAH    DFW     224     12      15      0
8 IAH    DFW     224      7      12       0
9 IAH    DFW     224      8      22      0
10 IAH   DFW     224      6      19      0
# ... with 227,486 more rows

```

2. Sélectionner les variables DepTime, ArrTime, ActualElapsedTime, AirTime, ArrDelay and DepDelay. On pourra remarquer que toutes ces variables contiennent les chaînes de caractère Time et Delay et utiliser la *helper function* `contains()`.

```

> hflights1 %>% select(contains("Time"),contains("Delay"))
# A tibble: 227,496 x 6
  DepTime ArrTime ActualElapsedTime AirTime ArrDelay DepDelay
  <int>   <int>        <int>   <int>   <int>   <int>
1 1400    1500         60      40     -10      0
2 1401    1501         60      45     -9       1
3 1352    1502         70      48     -8      -8
4 1403    1513         70      39      3       3
5 1405    1507         62      44     -3       5
6 1359    1503         64      45     -7      -1
7 1359    1509         70      43     -1      -1
8 1355    1454         59      40     -16      -5
9 1443    1554         71      41     44      43
10 1443   1553         70      45     43      43
# ... with 227,486 more rows

```

3. Ajouter une variable ActualGroundTime qui correspond à ActualElapsedTime moins AirTime.

```

> hflights2 <- hflights1 %>% mutate(ActualGroundTime=ActualElapsedTime-AirTime)
> head(hflights2)
# A tibble: 6 x 22
  Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
  <int> <int>    <int>    <int>   <int>   <int> <chr>          <int>
1 2011    1        1        6    1400    1500 American        428
2 2011    1        2        7    1401    1501 American        428

```

```

3 2011    1      3      1   1352   1502 American      428
4 2011    1      4      2   1403   1513 American      428
5 2011    1      5      3   1405   1507 American      428
6 2011    1      6      4   1359   1503 American      428
# ... with 14 more variables: TailNum <chr>, ActualElapsedTime <int>,
#   AirTime <int>, ArrDelay <int>, DepDelay <int>, Origin <chr>, Dest <chr>,
#   Distance <int>, TaxiIn <int>, TaxiOut <int>, Cancelled <int>,
#   CancellationCode <chr>, Diverted <int>, ActualGroundTime <int>

```

4. Ajouter la variable AverageSpeed (=Distance/AirTime) et ordonner la table selon les valeurs décroissantes de cette variable.

```

> hflights3 <- hflights2 %>%
+   mutate(AverageSpeed=Distance/AirTime) %>%
+   arrange(desc(AverageSpeed))

```

5. Sélectionner les vols à destination de JFK.

```

> filter(hflights3, Dest=="JFK")
# A tibble: 695 x 23
   Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
   <int> <int>     <int>     <int>   <int>   <int> <chr>       <int>
1 2011    2        7        1     659    1045 JetBlue      620
2 2011    2        6        7     700    1045 JetBlue      620
3 2011    2        5        6     700    1113 JetBlue      620
4 2011    2        6        7    1529    1917 JetBlue      624
5 2011    1       24        1     707    1059 JetBlue      620
6 2011    1       24        1    1532    1923 JetBlue      624
7 2011    2       12        6     659    1105 JetBlue      620
8 2011   10       19        3     644    1043 JetBlue      620
9 2011   11       10        4    1629    2027 JetBlue      622
10 2011    2        8        2     654    1049 JetBlue      620
# ... with 685 more rows, and 15 more variables: TailNum <chr>,
#   ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>, DepDelay <int>,
#   Origin <chr>, Dest <chr>, Distance <int>, TaxiIn <int>, TaxiOut <int>,
#   Cancelled <int>, CancellationCode <chr>, Diverted <int>,
#   ActualGroundTime <int>, AverageSpeed <dbl>

```

6. Calculer le nombre de vols à destination de JFK.

```

> hflights3 %>% filter(Dest=="JFK") %>% summarise( numb_to_JFK=n())
# A tibble: 1 x 1
  numb_to_JFK
  <int>
1      695

```

7. Créer un résumé de hflights1 qui contient :

- n : le nombre total de vols ;
- n\_dest : le nombre total de destinations ;
- n\_carrier : le nombre total de compagnies.

```

> hflights1 %>% summarise(n_flights=n(),n_dest=n_distinct(Dest),n_carrier=n_distinct(UniqueCarrier))
# A tibble: 1 x 3
  n_flights n_dest n_carrier
  <int>     <int>     <int>
1    227496      116        15

```

8. Créer un résumé de hflights1 qui contient, pour les vols de la compagnie American,

- le nombre total de vols ;
- le nombre total de vols annulés ;
- la valeur moyenne de ArrDelay (attention à la gestion des NA...).

```
> hflights1 %>% filter(UniqueCarrier=="American") %>%
+   summarise(n_fligths_Am=n(),n_can_Am=sum(Cancelled),
+             mean_ArrDelay_am=mean(ArrDelay,na.rm=TRUE))
# A tibble: 1 x 3
  n_fligths_Am n_can_Am mean_ArrDelay_am
      <int>     <int>          <dbl>
1       3244        60         0.892
```

9. Calculer pour chaque compagnie :

- le nombre total de vols ;
- La valeur moyenne de AirTime.

```
> hflights1 %>% group_by(UniqueCarrier) %>%
+   summarise(n_flights=n(),mean_AirTime=mean(AirTime,na.rm=TRUE))
# A tibble: 15 x 3
  UniqueCarrier    n_flights mean_AirTime
      <chr>          <int>        <dbl>
1 AirTran            2139        92.7
2 Alaska              365        254.
3 American            3244        69.7
4 American_Eagle      4648        93.8
5 Atlantic_Southeast  2204        104.
6 Continental        70032       145.
7 Delta                2641        97.8
8 ExpressJet          73053       83.2
9 Frontier              838        125.
10 JetBlue              695        184.
11 Mesa                  79        122.
12 SkyWest              16061       113.
13 Southwest            45343       86.7
14 United                2072        157.
15 US_Airways           4082        134.
```

10. Ordonner les compagnies en fonction des retards moyens au départ.

```
> hflights1 %>%
+   group_by(UniqueCarrier) %>%
+   filter(!is.na(DepDelay) & DepDelay>0) %>%
+   summarise(meanDepDelay = mean(DepDelay)) %>%
+   arrange(meanDepDelay)
# A tibble: 15 x 2
  UniqueCarrier    meanDepDelay
      <chr>          <dbl>
1 Continental        17.9
2 Alaska              20.8
3 Southwest           21.9
4 Frontier             22.7
5 Mesa                 24.5
6 SkyWest              24.6
7 American             24.7
8 US_Airways           26.5
9 ExpressJet            26.9
```

|    |                    |      |
|----|--------------------|------|
| 10 | United             | 28.8 |
| 11 | Delta              | 32.4 |
| 12 | AirTran            | 33.4 |
| 13 | American_Eagle     | 37.9 |
| 14 | JetBlue            | 43.5 |
| 15 | Atlantic_Southeast | 49.3 |

**Exercice 3.5** (Tournois du grand chelem au tennis).

On considère le données sur les résultats de tennis dans les tournois du grand chelem en 2013. Les données, ainsi que le descriptif des variables, se trouvent à l'adresse <https://archive.ics.uci.edu/ml/datasets/Tennis+Major+Tournament+Match+Statistics>.

On s'intéresse d'abord au tournoi masculin de Roland Garros. On répondra aux questions à l'aide des verbes **dplyr**.

1. Importer les données.

```
> FrenchOpen_men_2013 <- read_csv("data/FrenchOpen-men-2013.csv")
> RG2013 <- FrenchOpen_men_2013
> RG2013
# A tibble: 125 x 42
   Player1 Player2 Round Result FNL.1 FNL.2 FSP.1 FSW.1 SSP.1 SSW.1 ACE.1 DBF.1
   <chr>   <chr>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 Pablo ~ Roger ~    1     0     0     3    62    27    38    11     1     1     3
 2 Somdev~ Daniel~    1     1     3     0    62    54    38    22     7     7     3
 3 Tobias~ Paolo ~   1     1     3     2    62    53    38    15     4     4     6
 4 Julien~ Ricard~   1     1     3     1    72    87    28    19    14    14     2
 5 Lukas ~ Sam Qu~   1     0     0     3    52    31    48    22     4     4     4
 6 Jan Ha~ Denis ~  1     1     3     1    70    58    30    18     4     4     4
 7 Adrian~ Pablo ~  1     0     2     3    63    71    37    38     5     5     5
 8 Gilles~ Lleyto~   1     1     3     2    59    42    41    25     7     7     2
 9 Philip~ Marin ~  1     0     0     3    56    27    44    13     0     0     6
10 Radek ~ Nick K~  1     0     0     3    63    62    37    29     5     5     4
# ... with 115 more rows, and 30 more variables: WNR.1 <dbl>, UFE.1 <dbl>,
#   BPC.1 <dbl>, BPW.1 <dbl>, NPA.1 <dbl>, NPW.1 <dbl>, TPW.1 <dbl>,
#   ST1.1 <dbl>, ST2.1 <dbl>, ST3.1 <dbl>, ST4.1 <dbl>, ST5.1 <dbl>,
#   FSP.2 <dbl>, FSW.2 <dbl>, SSP.2 <dbl>, SSW.2 <dbl>, ACE.2 <dbl>,
#   DBF.2 <dbl>, WNR.2 <dbl>, UFE.2 <dbl>, BPC.2 <dbl>, BPW.2 <dbl>,
#   NPA.2 <dbl>, NPW.2 <dbl>, TPW.2 <dbl>, ST1.2 <dbl>, ST2.2 <dbl>,
#   ST3.2 <dbl>, ST4.2 <dbl>, ST5.2 <dbl>
```

2. Afficher le nom des adversaires de Roger Federer.

```
> RG2013 %>% filter(Player1=="Roger Federer" | Player2=="Roger Federer") %>%
+   select(Player1,Player2)
# A tibble: 5 x 2
  Player1          Player2
  <chr>            <chr>
 1 Pablo Carreno-Busta Roger Federer
 2 Somdev Devvarman  Roger Federer
 3 Julien Benneteau Roger Federer
 4 Gilles Simon      Roger Federer
 5 Jo-Wilfried Tsonga Roger Federer
```

3. Afficher le nom des demi-finalistes (ceux qui ont atteint le 6ème tour).

```

> RG2013 %>% filter(Round==6) %>% select(Player1,Player2)
# A tibble: 2 x 2
  Player1      Player2
  <chr>        <chr>
1 David Ferrer Jo-Wilfried Tsonga
2 Novak Djokovic Rafael Nadal

```

4. Combien y a-t-il eu de points disputés en moyenne par match ? Il faudra penser à ajouter dans la table une variable correspondant au nombre de points de chaque match (verbe `mutate`).

```

> RG2013 %>% mutate(nb_points=TPW.1+TPW.2) %>% select(nb_points) %>% summarize_all(mean)
# A tibble: 1 x 1
  nb_points
  <dbl>
1     219.

```

5. Combien y a-t-il eu d'aces par match en moyenne ?

```

> RG2013 %>% mutate(nb_aces=ACE.1+ACE.2) %>% summarize(mean_aces=mean(nb_aces))
# A tibble: 1 x 1
  mean_aces
  <dbl>
1     12.7

```

6. Combien y a-t-il eu d'aces par match en moyenne à chaque tour ?

```

> RG2013 %>% group_by(Round) %>% mutate(nb_aces=ACE.1+ACE.2) %>%
+   summarize(mean_aces=mean(nb_aces))
# A tibble: 7 x 2
  Round mean_aces
  <dbl>    <dbl>
1     1     13.5
2     2     13.2
3     3     12.6
4     4     9.12
5     5      7
6     6     10
7     7      6

```

7. Combien y a-t-il eu de doubles fautes au total dans le tournoi (attention aux données manquantes, taper `help(sum)` pour voir comment les gérer) ?

```

> RG2013 %>% mutate(nb_df=DBF.1+DBF.2) %>%
+   summarize(nb_dbfaults=sum(nb_df,na.rm=TRUE))
# A tibble: 1 x 1
  nb_dbfaults
  <dbl>
1     812

```

8. Importer les données pour le tournoi masculin de Wimbledon 2013.

```

> WIMB2013 <- read_csv("data/Wimbledon-men-2013.csv")
> WIMB2013
# A tibble: 114 x 42
  Player1 Player2 Round Result FNL.1 FNL.2 FSP.1 FSW.1 SSP.1 SSW.1 ACE.1 DBF.1
  <chr>   <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 B. Beck~ A. Murr~     1      0      0      3     59     29     41     14      5      1
2 J. Ward   Y-H. Lu     1      0      1      3     62     77     38     35     18      4
3 N. Mahut  J. Hajek    1      1      3      0     72     44     28     10     17      3
4 T. Robr~  A. Bogo~    1      1      3      0     77     40     23     12      6      0

```

```

5 R.Haase M.Youz~ 1 0 0 3 68 61 32 15 7 2
6 M.Gicq~ V.Posp~ 1 0 0 3 59 41 41 27 7 6
7 A.Kuzn~ A.Mont~ 1 1 3 1 63 56 37 21 21 3
8 J.Tips~ V.Troi~ 1 0 0 3 61 47 39 21 3 1
9 M.Bagh~ M.Cilic 1 0 0 3 61 31 39 16 4 5
10 K.De S~ P.Lore~ 1 1 3 0 67 56 33 21 22 6
# ... with 104 more rows, and 30 more variables: WNR.1 <dbl>, UFE.1 <dbl>,
#   BPC.1 <dbl>, BPW.1 <dbl>, NPA.1 <dbl>, NPW.1 <dbl>, TPW.1 <lgl>,
#   ST1.1 <dbl>, ST2.1 <dbl>, ST3.1 <dbl>, ST4.1 <dbl>, ST5.1 <dbl>,
#   FSP.2 <dbl>, FSW.2 <dbl>, SSP.2 <dbl>, SSW.2 <dbl>, ACE.2 <dbl>,
#   DBF.2 <dbl>, WNR.2 <dbl>, UFE.2 <dbl>, BPC.2 <dbl>, BPW.2 <dbl>,
#   NPA.2 <dbl>, NPW.2 <dbl>, TPW.2 <lgl>, ST1.2 <dbl>, ST2.2 <dbl>,
#   ST3.2 <dbl>, ST4.2 <dbl>, ST5.2 <dbl>

```

9. Concaténer les tables en ajoutant une variable permettant d'identifier le tournoi. On pourra utiliser `bind_rows` avec l'option `.id`.

```
> RG_WIMB2013 <- bind_rows("RG"=RG2013,"WIMB"=WIMB2013,.id="Tournament")
```

10. Afficher les matchs de Federer pour chaque tournoi.

```

> RG_WIMB2013 %>% filter(Player1=="Roger Federer" |
+                           Player2=="Roger Federer" |
+                           Player1=="R.Federer" |
+                           Player2=="R.Federer")
# A tibble: 7 x 43
  Tournament Player1 Player2 Round Result FNL.1 FNL.2 FSP.1 FSW.1 SSP.1 SSW.1
  <chr>     <chr>   <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 RG        Pablo ~ Roger ~    1     0     0     3     62    27    38    11
2 RG        Somdev~ Roger ~    2     0     0     3     61    19    39    16
3 RG        Julien~ Roger ~    3     0     0     3     82    41    18     8
4 RG        Gilles~ Roger ~    4     0     2     3     61    65    39    28
5 RG        Jo-Wil~ Roger ~    5     1     3     0     75    46    25    10
6 WIMB      V.Hane~ R.Fede~   1     0     0     3     85    26    15     3
7 WIMB      S.Stak~ R.Fede~   2     1     3     1     66    83    34    36
# ... with 32 more variables: ACE.1 <dbl>, DBF.1 <dbl>, WNR.1 <dbl>,
#   UFE.1 <dbl>, BPC.1 <dbl>, BPW.1 <dbl>, NPA.1 <dbl>, NPW.1 <dbl>,
#   TPW.1 <dbl>, ST1.1 <dbl>, ST2.1 <dbl>, ST3.1 <dbl>, ST4.1 <dbl>,
#   ST5.1 <dbl>, FSP.2 <dbl>, FSW.2 <dbl>, SSP.2 <dbl>, SSW.2 <dbl>,
#   ACE.2 <dbl>, DBF.2 <dbl>, WNR.2 <dbl>, UFE.2 <dbl>, BPC.2 <dbl>,
#   BPW.2 <dbl>, NPA.2 <dbl>, NPW.2 <dbl>, TPW.2 <dbl>, ST1.2 <dbl>,
#   ST2.2 <dbl>, ST3.2 <dbl>, ST4.2 <dbl>, ST5.2 <dbl>

```

ou

```

> RG_WIMB2013 %>% filter(grepl("Federer",Player2) | grepl("Federer",Player2))
# A tibble: 7 x 43
  Tournament Player1 Player2 Round Result FNL.1 FNL.2 FSP.1 FSW.1 SSP.1 SSW.1
  <chr>     <chr>   <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 RG        Pablo ~ Roger ~    1     0     0     3     62    27    38    11
2 RG        Somdev~ Roger ~    2     0     0     3     61    19    39    16
3 RG        Julien~ Roger ~    3     0     0     3     82    41    18     8
4 RG        Gilles~ Roger ~    4     0     2     3     61    65    39    28
5 RG        Jo-Wil~ Roger ~    5     1     3     0     75    46    25    10
6 WIMB      V.Hane~ R.Fede~   1     0     0     3     85    26    15     3
7 WIMB      S.Stak~ R.Fede~   2     1     3     1     66    83    34    36
# ... with 32 more variables: ACE.1 <dbl>, DBF.1 <dbl>, WNR.1 <dbl>,
#   UFE.1 <dbl>, BPC.1 <dbl>, BPW.1 <dbl>, NPA.1 <dbl>, NPW.1 <dbl>,

```

```
# TPW.1 <dbl>, ST1.1 <dbl>, ST2.1 <dbl>, ST3.1 <dbl>, ST4.1 <dbl>,
# ST5.1 <dbl>, FSP.2 <dbl>, FSW.2 <dbl>, SSP.2 <dbl>, SSW.2 <dbl>,
# ACE.2 <dbl>, DBF.2 <dbl>, WNR.2 <dbl>, UFE.2 <dbl>, BPC.2 <dbl>,
# BPW.2 <dbl>, NPA.2 <dbl>, NPW.2 <dbl>, TPW.2 <dbl>, ST1.2 <dbl>,
# ST2.2 <dbl>, ST3.2 <dbl>, ST4.2 <dbl>, ST5.2 <dbl>
```

11. Comparer les nombres d'aces par matchs à chaque tours pour les tournois de Roland Garros et Wimbledon.

```
> RG_WIMB2013 %>% group_by(Tournament, Round) %>%
+   mutate(nb_aces=ACE.1+ACE.2) %>% summarize(mean_ace=mean(nb_aces))
# A tibble: 14 x 3
# Groups:   Tournament [2]
  Tournament Round mean_ace
  <chr>      <dbl>    <dbl>
1 RG          1     13.5
2 RG          2     13.2
3 RG          3     12.6
4 RG          4     9.12
5 RG          5      7
6 RG          6     10
7 RG          7      6
8 WIMB        1     21.1
9 WIMB        2     23.9
10 WIMB       3     24
11 WIMB       4     24.4
12 WIMB       5     26.5
13 WIMB       6     27.5
14 WIMB       7     13
```

ou pour une présentation plus synthétique

```
> RG_WIMB2013 %>% group_by(Tournament, Round) %>%
+   mutate(nb_aces=ACE.1+ACE.2) %>%
+   summarize(mean_ace=mean(nb_aces)) %>%
+   pivot_wider(names_from = "Round", values_from = "mean_ace")
# A tibble: 2 x 8
# Groups:   Tournament [2]
  Tournament `1` `2` `3` `4` `5` `6` `7`
  <chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 RG         13.5 13.2 12.6 9.12 7     10    6
2 WIMB       21.1 23.9 24    24.4 26.5 27.5 13
```

### 3.2.5 Compléments : Tidy data avec tidyr

L'utilisation de **dplyr** et de **ggplot** (que nous verrons dans la partie suivante) suppose que les données sont présentées sous un format adéquat : une ligne est un individu et une colonne une variable, on parle alors de **tidy data**. Cela n'est pas toujours le cas en pratique, considérons par exemple le tableau suivant qui présente les taux de chômage des départements français en 2002, 2006, 2011

```
> df <- read_delim("data/tauxchomage.csv", delim=";") %>% select(-1)
> df
# A tibble: 96 x 4
  NOM_DPT          TCHOMB1T01 TCHOMB1T06 TCHOMB1T11
  <chr>            <dbl>       <dbl>       <dbl>
1 Ain              3.9         5.9         6.6
2 Aisne            10.6        12          13.2
```

|                           |      |      |      |
|---------------------------|------|------|------|
| 3 Allier                  | 9    | 9.2  | 9.7  |
| 4 Alpes-de-Haute-Provence | 9.5  | 9.7  | 10.3 |
| 5 Hautes-Alpes            | 7.1  | 7.7  | 8.3  |
| 6 Alpes-Maritimes         | 9.1  | 8.9  | 9.2  |
| 7 Ardèche                 | 8.1  | 9.6  | 9.7  |
| 8 Ardennes                | 11.5 | 12.8 | 10.9 |
| 9 Ariège                  | 9.2  | 10.1 | 10.6 |
| 10 Aube                   | 8.2  | 10   | 10   |
| # ... with 86 more rows   |      |      |      |

Ce tableau n'est pas **tidy** dans le sens où les variables mesurées sont

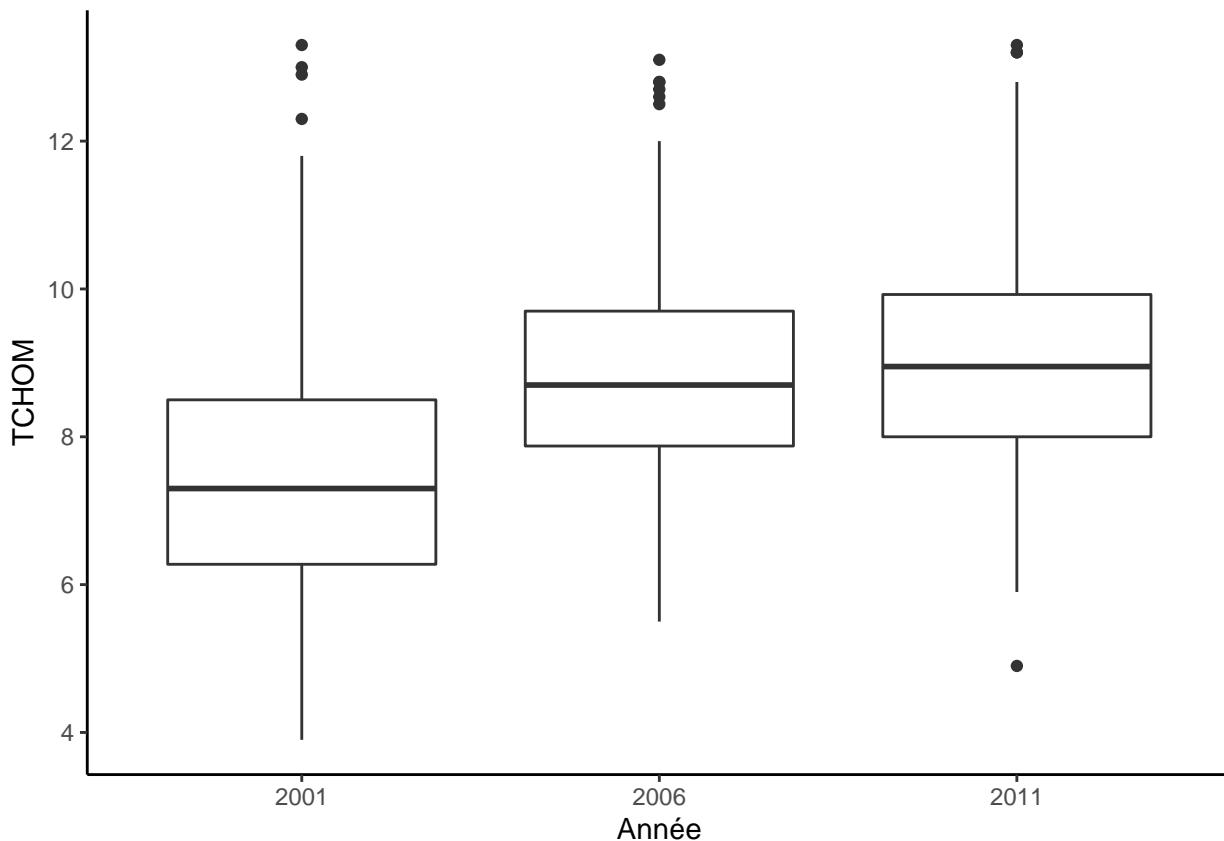
- le département
- l'année
- le taux de chômage

Nous verrons qu'il n'est par exemple pas possible de faire un **boxplot** permettant de visualiser la distribution du taux de chômage en fonction de l'année à l'aide de **ggplot2**. Pour passer au format **tidy** il faut assembler les 3 colonnes correspondant aux taux de chômage en une seule colonne et ajouter une colonne qui permette d'identifier l'année. La fonction **pivot\_longer** du package **tidyverse** permet de faire cela :

```
> df1 <- df %>% pivot_longer(-NOM_DPT, names_to="Année", values_to="TCHOM") %>%
+   mutate(Année=fct_recode(Année, "2001"="TCHOMB1T01", "2006"="TCHOMB1T06", "2011"="TCHOMB1T11"))
> df1
# A tibble: 288 x 3
  NOM_DPT           Année    TCHOM
  <chr>             <fct>   <dbl>
1 Ain               2001     3.9
2 Ain               2006     5.9
3 Ain               2011     6.6
4 Aisne              2001    10.6
5 Aisne              2006     12
6 Aisne              2011    13.2
7 Allier             2001     9
8 Allier             2006    9.2
9 Allier             2011    9.7
10 Alpes-de-Haute-Provence 2001    9.5
# ... with 278 more rows
```

Il sera alors aisé de faire le boxplot souhaité avec

```
> ggplot(df1)+aes(x=Année, y=TCHOM)+geom_boxplot()
```



L'opération inverse peut être effectuée avec **pivot\_wider** :

```
> df1 %>% pivot_wider(names_from = "Année", values_from = "TCHOM")
# A tibble: 96 x 4
   NOM_DPT     `2001` `2006` `2011`
   <chr>      <dbl>   <dbl>   <dbl>
 1 Ain          3.9    5.9    6.6
 2 Aisne        10.6   12     13.2
 3 Allier       9      9.2    9.7
 4 Alpes-de-Haute-Provence  9.5   9.7   10.3
 5 Hautes-Alpes  7.1    7.7    8.3
 6 Alpes-Maritimes  9.1    8.9    9.2
 7 Ardèche      8.1    9.6    9.7
 8 Ardennes     11.5   12.8   10.9
 9 Ariège        9.2    10.1   10.6
10 Aube         8.2    10     10
# ... with 86 more rows
```

Le package **tidyverse** possède plusieurs autres verbes qui pourront aider l'utilisateur à mettre la table sous le meilleur format pour les analyses. Citons par exemple le verbe **separate** qui va séparer une colonne en plusieurs :

```
> df <- tibble(date=as.Date(c("01/03/2015","05/18/2017",
+                           "09/14/2018"), "%m/%d/%Y"), temp=c(18,21,15))
> df
# A tibble: 3 x 2
  date      temp
  <date>    <dbl>
```

```

1 2015-01-03    18
2 2017-05-18    21
3 2018-09-14    15
> df1 <- df %>% separate(date,into = c("year","month","day"))
> df1
# A tibble: 3 x 4
  year   month  day   temp
  <chr>  <chr> <chr> <dbl>
1 2015   01     03     18
2 2017   05     18     21
3 2018   09     14     15

```

ou le verbe **unite** qui fera l'opération inverse

```

> df1 %>% unite(date,year,month,day,sep="/")
# A tibble: 3 x 2
  date      temp
  <chr>    <dbl>
1 2015/01/03    18
2 2017/05/18    21
3 2018/09/14    15

```

Citons enfin les verbes :

- **separate\_rows** qui permettra de séparer des informations en plusieurs lignes ;
- **extract** pour créer de nouvelles colonnes ;
- **complete**

## 4 Visualisation avec ggplot2

Il est souvent nécessaire d'utiliser des techniques de visualisation au cours des différentes étapes d'un projet en science des données. Un des avantages de **R** est qu'il est relativement simple de mettre en œuvre tout les types de graphes généralement utilisés. Dans cette fiche, nous présentons tout d'abord les fonctions classiques qui permettent de tracer des figures. Nous proposons ensuite une introduction aux graphes **ggplot** qui sont de plus en plus utilisés pour faire de la visualisation.

### 4.1 Fonctions graphiques conventionnelles

Pour commencer il est intéressant d'examiner quelques exemples de représentations graphiques construits avec **R**. On peut les obtenir à l'aide de la fonction **demo**.

```
> demo(graphics)
```

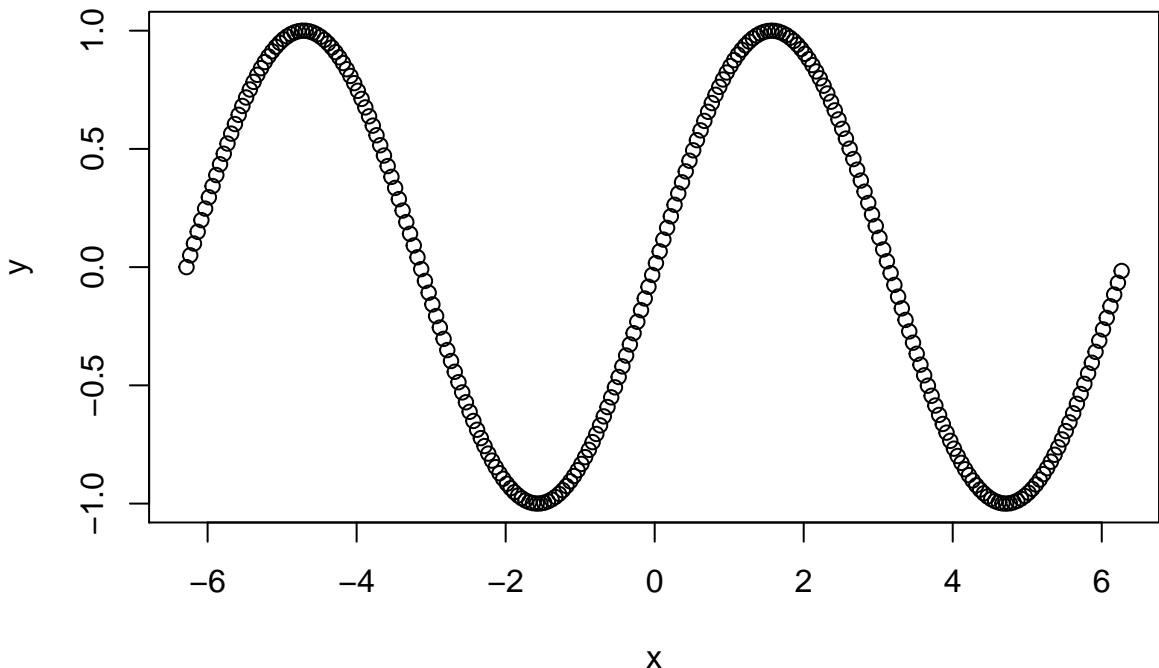
#### 4.1.1 La fonction plot

C'est une **fonction générique** que l'on peut utiliser pour représenter différents types de données. L'utilisation standard consiste à visualiser une variable  $y$  en fonction d'une variable  $x$ . On peut par exemple obtenir le graphe de la fonction  $x \mapsto \sin(2\pi x)$  sur  $[0, 1]$ , à l'aide de

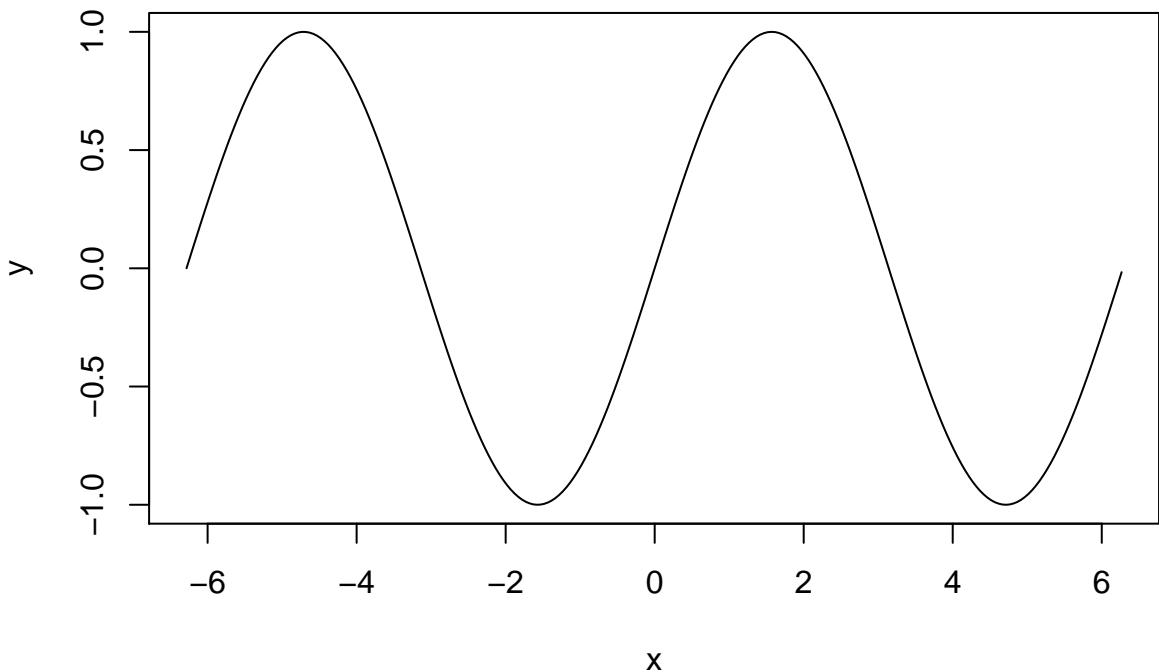
```

> x <- seq(-2*pi,2*pi,by=0.05)
> y <- sin(x)
> plot(x,y) #points (par défaut)

```



```
> plot(x,y,type="l") #représentation sous forme de ligne
```



Nous proposons des exemples de représentations de variables quantitatives et qualitatives à l'aide du jeu de données **ozone.txt** que l'on importe avec

```
> ozone <- read.table("data/ozone.txt")
> summary(ozone)
   max03          T9          T12          T15
Min. : 42.00  Min. :11.30  Min. :14.00  Min. :14.90
1st Qu.: 70.75 1st Qu.:16.20  1st Qu.:18.60  1st Qu.:19.27
Median : 81.50 Median :17.80  Median :20.55  Median :22.05
Mean   : 90.30 Mean  :18.36  Mean   :21.53  Mean   :22.63
```

```

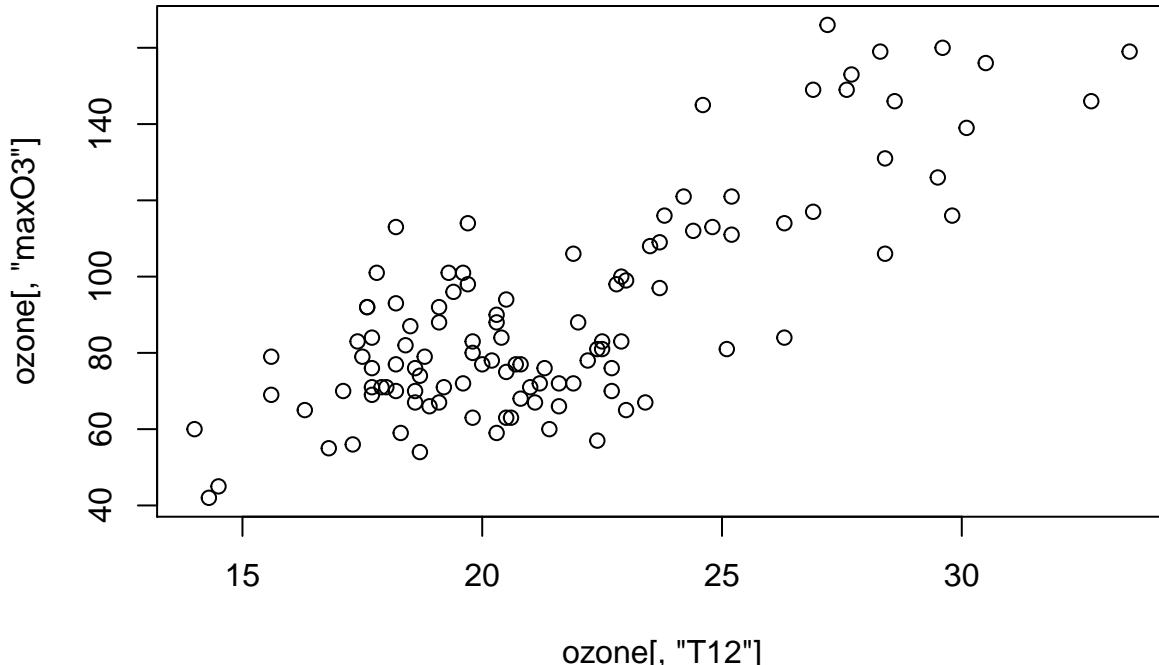
3rd Qu.:106.00   3rd Qu.:19.93    3rd Qu.:23.55    3rd Qu.:25.40
Max. :166.00     Max. :27.00      Max. :33.50      Max. :35.50
  Ne9           Ne12          Ne15          Vx9
Min. :0.000      Min. :0.000      Min. :0.00      Min. :-7.8785
1st Qu.:3.000    1st Qu.:4.000    1st Qu.:3.00    1st Qu.:-3.2765
Median :6.000    Median :5.000    Median :5.00    Median :-0.8660
Mean   :4.929    Mean   :5.018    Mean   :4.83     Mean   :-1.2143
3rd Qu.:7.000    3rd Qu.:7.000    3rd Qu.:7.00    3rd Qu.: 0.6946
Max. :8.000      Max. :8.000      Max. :8.00      Max. : 5.1962
  Vx12          Vx15          max03v        vent
Min. :-7.878    Min. :-9.000    Min. : 42.00    Length:112
1st Qu.:-3.565  1st Qu.:-3.939  1st Qu.: 71.00  Class :character
Median :-1.879  Median :-1.550  Median : 82.50  Mode  :character
Mean   :-1.611  Mean   :-1.691  Mean   : 90.57
3rd Qu.: 0.000  3rd Qu.: 0.000  3rd Qu.:106.00
Max. : 6.578    Max. : 5.000    Max. :166.00

  pluie
Length:112
Class :character
Mode  :character

```

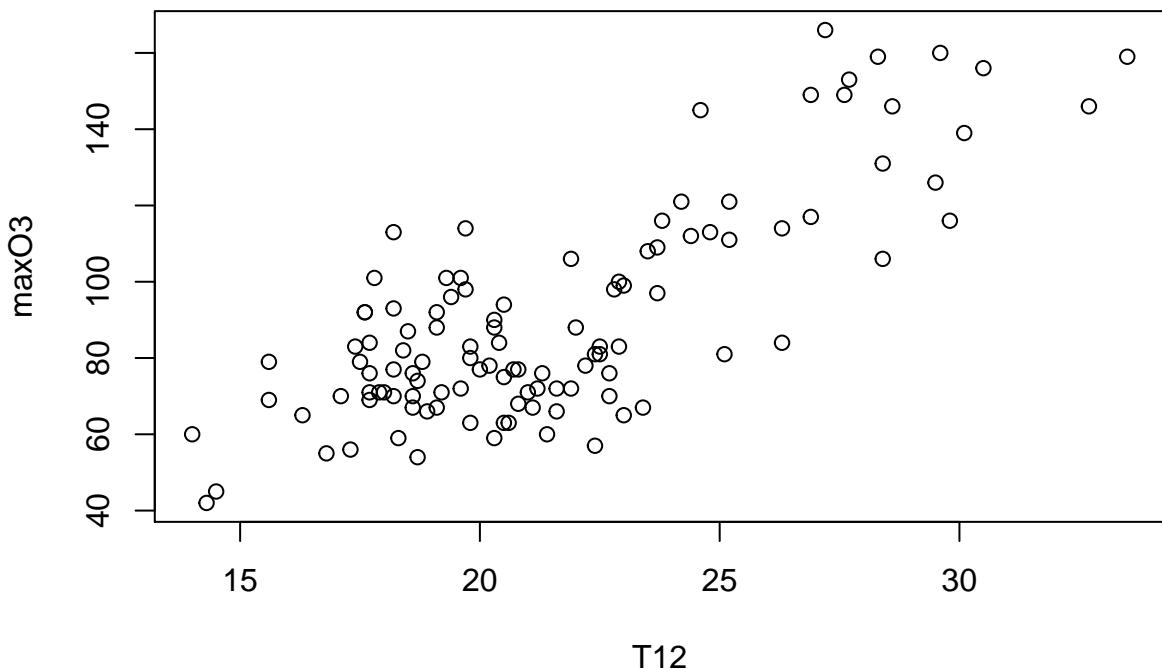
On visualise tout d'abord 2 variables quantitatives à l'aide d'un nuage de points : la concentration en ozone maximale **maxO3** en fonction de la température à 12h **T12**.

```
> plot(ozone[, "T12"], ozone[, "maxO3"])
```



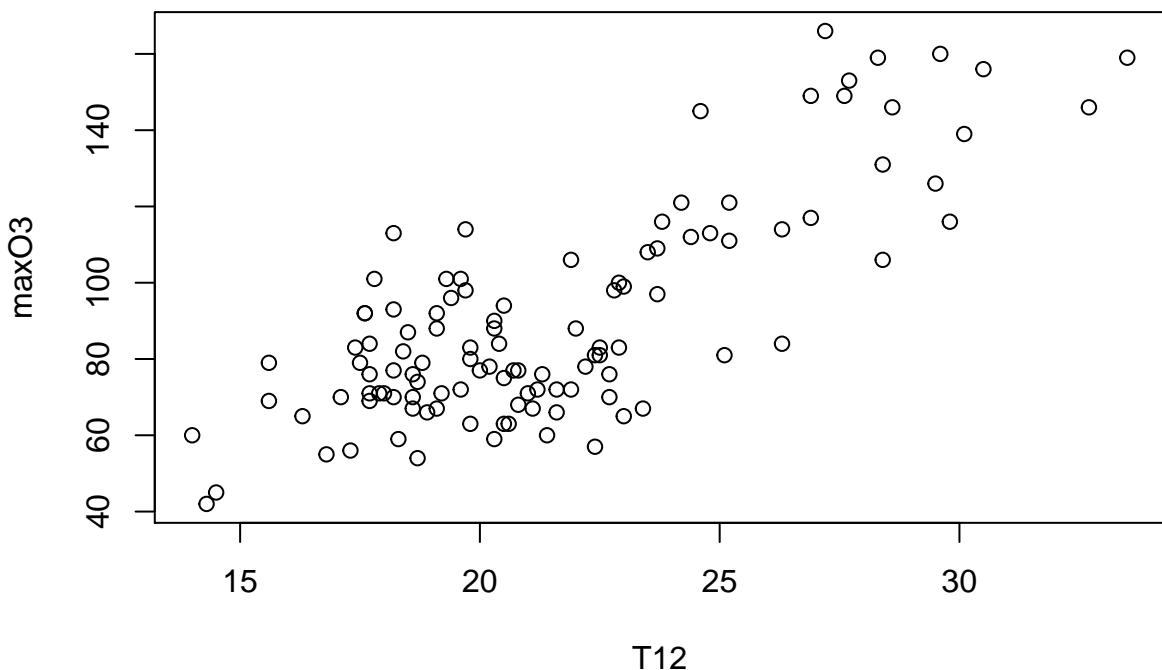
Comme les deux variables appartiennent au même jeu de données, on peut obtenir la même représentation à l'aide d'une syntaxe plus claire qui ajoute automatiquement les noms des variables sur les axes :

```
> plot(max03~T12, data=ozone)
```



Une autre façon de faire (moins naturelle) :

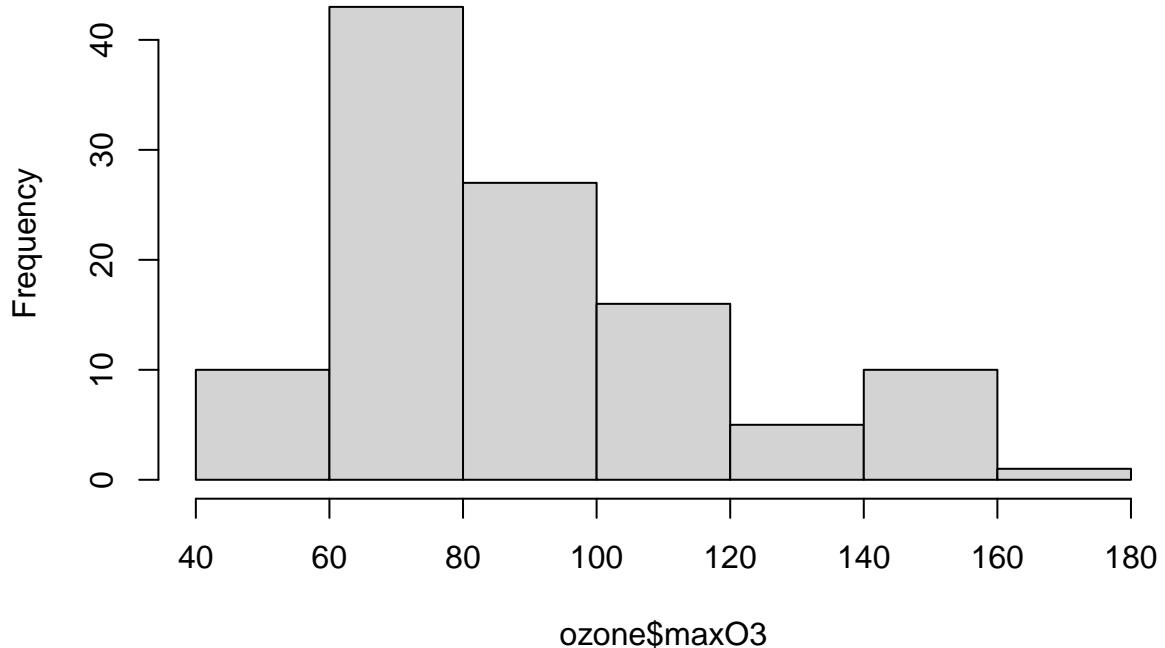
```
> plot(ozone[, "T12"], ozone[, "max03"], xlab="T12", ylab="max03")
```



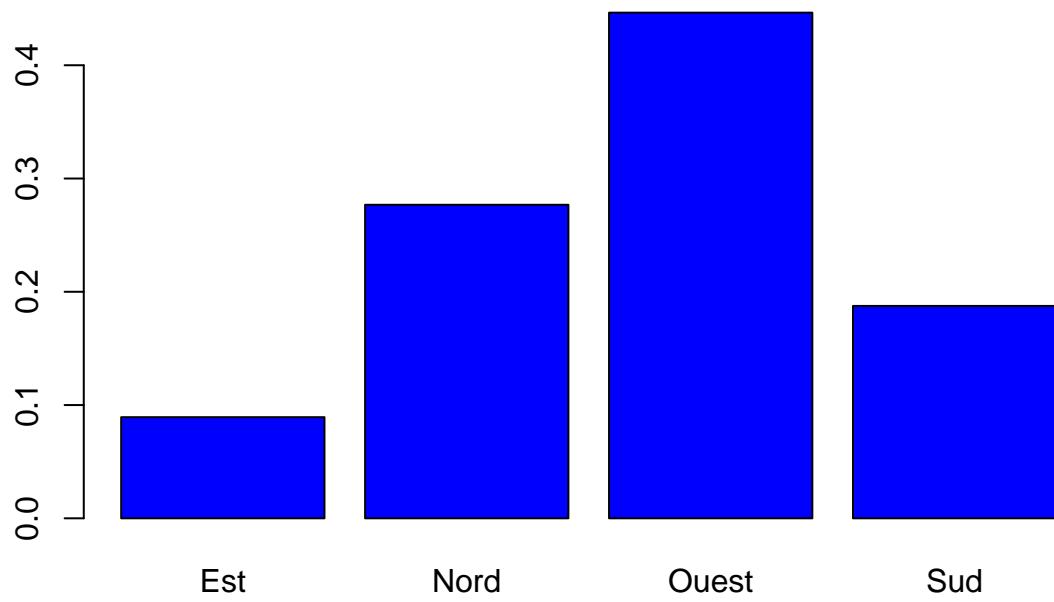
Il existe des fonctions spécifiques pour chaque type de graphs, par exemple **histogram**, **barplot** et **boxplot** :

```
> hist(ozone$maxO3,main="Histogram")
```

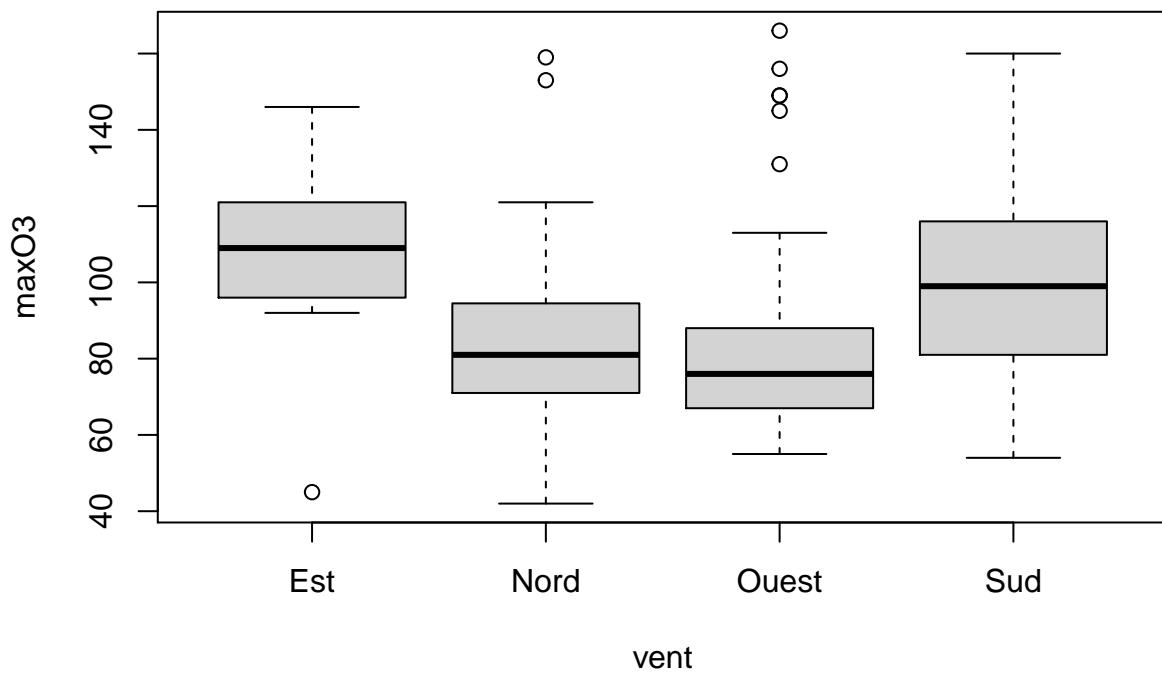
Histogram



```
> barplot(table(ozone$vent)/nrow(ozone),col="blue")
```



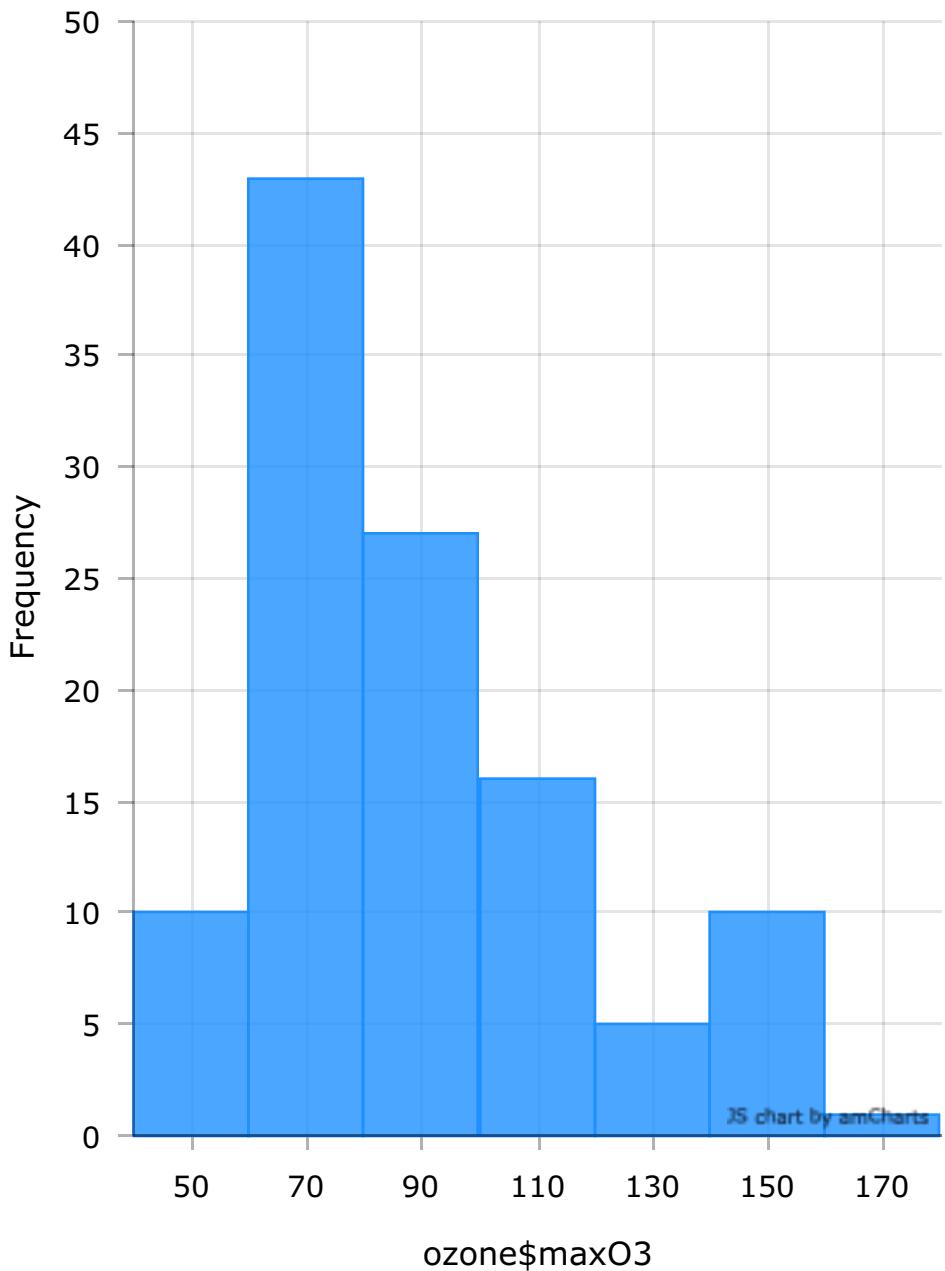
```
> boxplot(maxO3~vent,data=ozone)
```



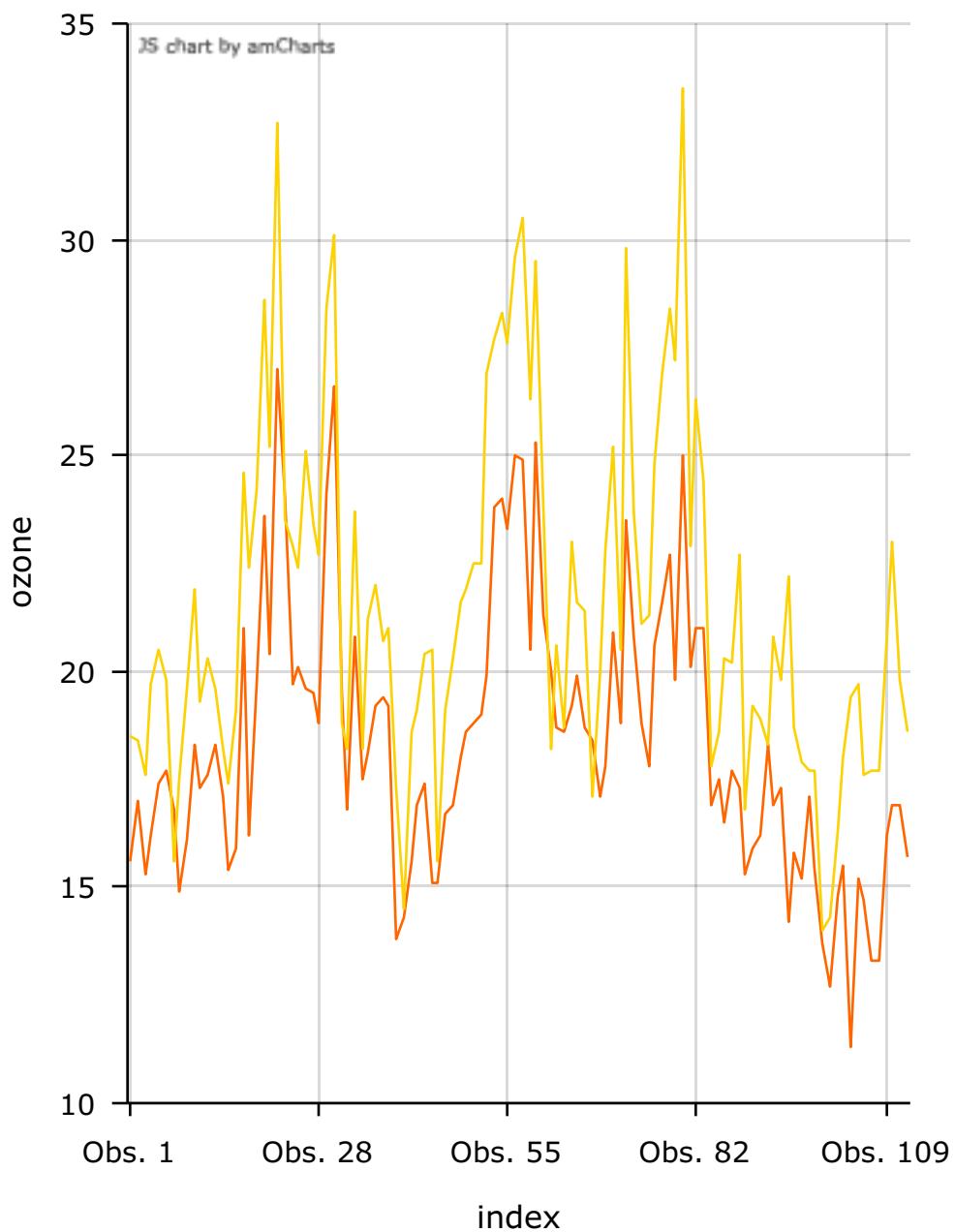
#### 4.1.2 Graphes interactifs avec rAmCharts

On peut utiliser ce package pour obtenir des graphes dynamiques. L'utilisation est relativement simple, il suffit d'ajouter le préfixe **am** devant le nom de la fonction :

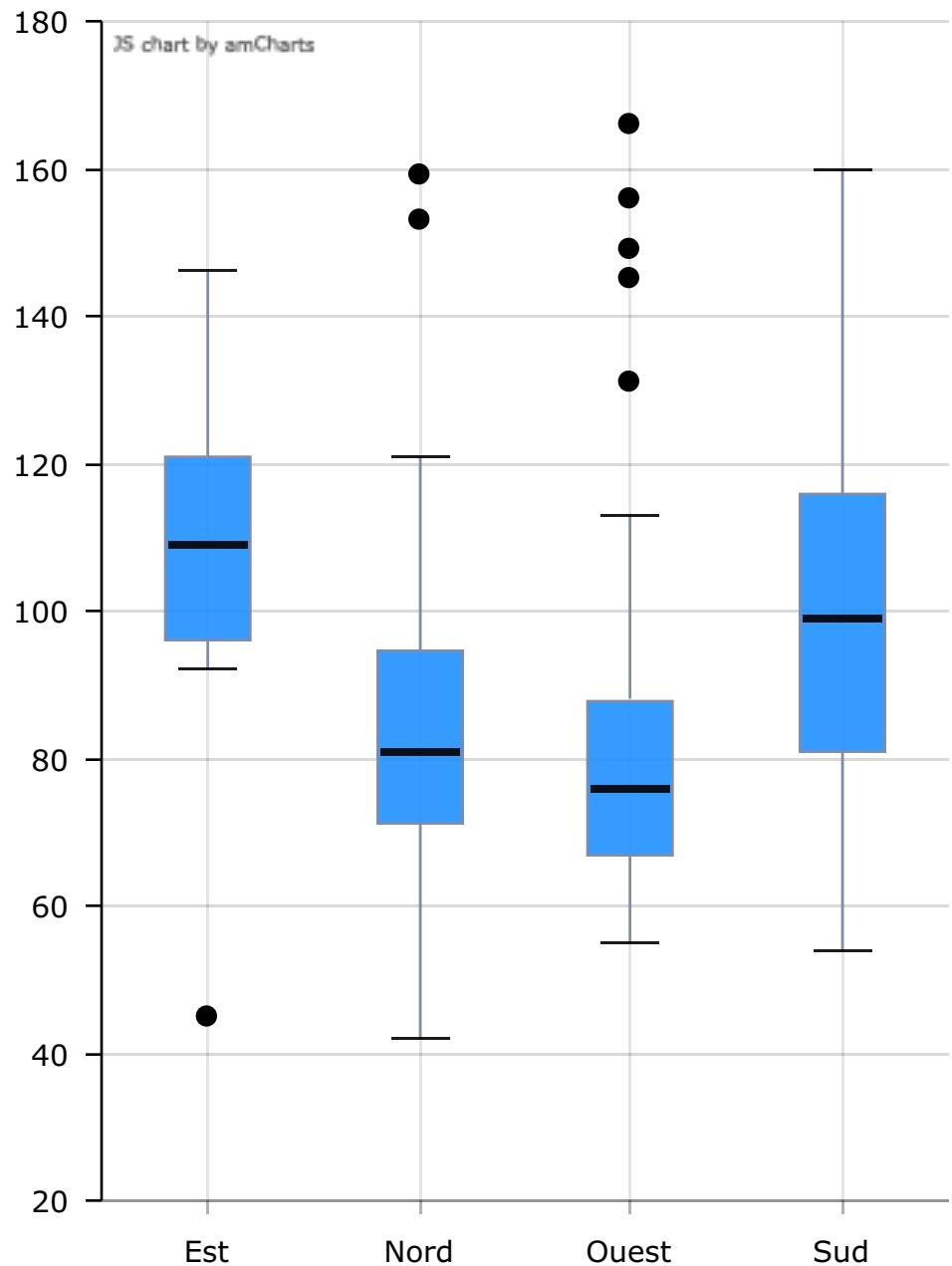
```
> library(rAmCharts)
> amHist(ozone$maxO3)
```



```
> amPlot(ozone,col=c("T9","T12"))
```



```
> amBoxplot(max03~vent, data=ozone)
```

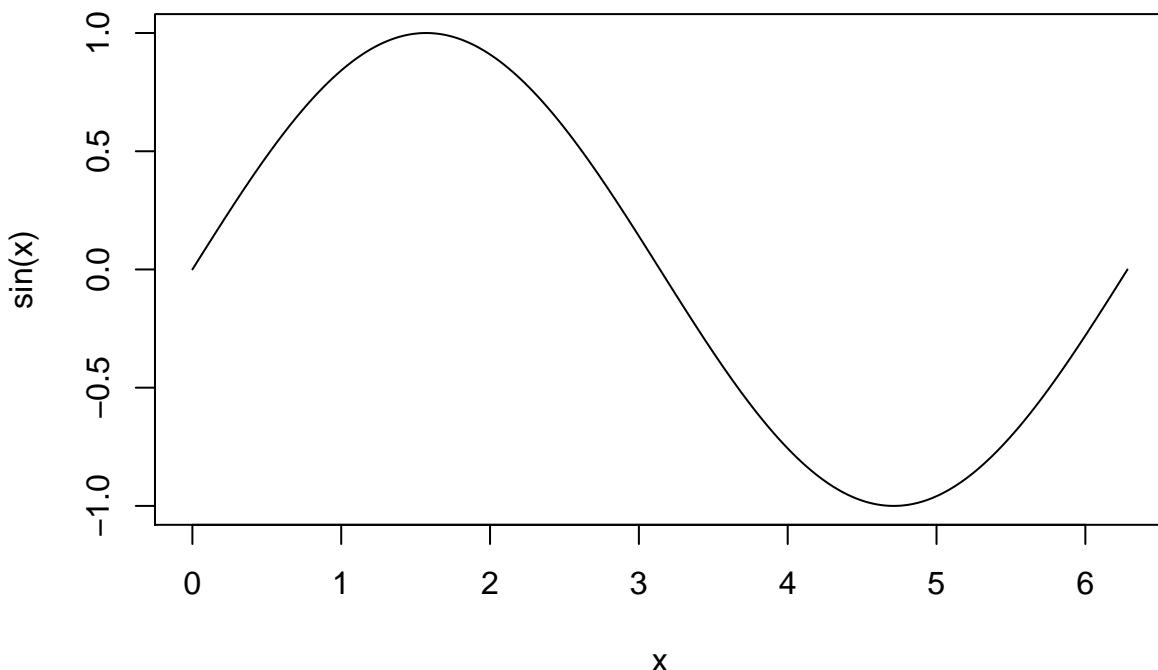


#### Exercice 4.1 (Premier graphe).

1. Tracer la fonction **sinus** entre  $0$  et  $2\pi$ .
2. A l'aide de la fonction **title** ajouter le titre **Représentation de la fonction sinus**.

```
> x <- seq(0,2*pi,length=1000)
> plot(x,sin(x),type="l")
> title("Représentation de la fonction sinus")
```

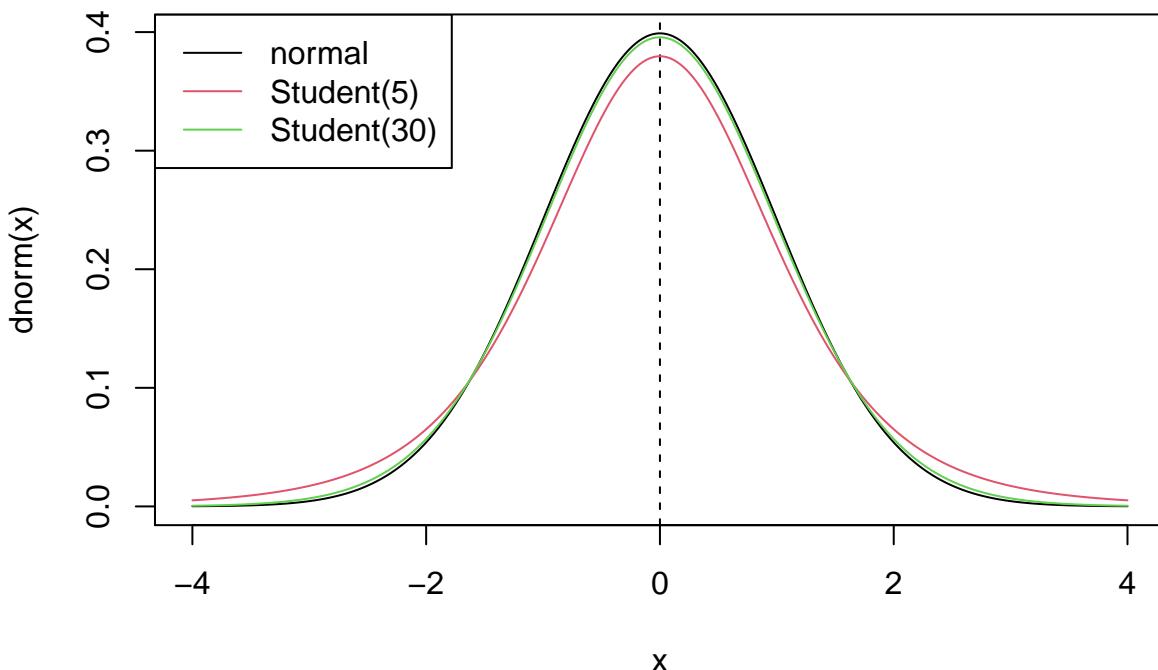
## Représentation de la fonction sinus



**Exercice 4.2** (Tracé de densités).

1. Tracer la densité de la loi normale centrée réduite entre  $-4$  et  $4$  (utiliser **dnorm**).
2. Ajouter une ligne verticale (en tirets) qui passe par  $x = 0$  (utiliser **abline** avec **lty=2**).
3. Sur le même graphe, ajouter les densités de loi la de Student à  $5$  et  $30$  degrés de liberté (utiliser **dt**).  
On utilisera la fonction **lines** et des couleurs différentes pour chaque densité.
4. Ajouter une légende qui permet de repérer chaque densité (fonction **legend**).

```
> x <- seq(-4,4,by=0.01)
> plot(x,dnorm(x),type="l")
> abline(v=0,lty=2)
> lines(x,dt(x,5),col=2)
> lines(x,dt(x,30),col=3)
> legend("topleft",legend=c("normal","Student(5)","Student(30)"),
+       col=1:3,lty=1)
```



**Exercice 4.3** (Tâches solaires).

1. Importer la série `taches_solaires.csv` qui donne, date par date, un nombre de tâches solaires observées.

```
> taches <- read.table("data/taches_solaires.csv",sep=";",header=TRUE,dec=",")
```

2. A l'aide de la fonction `cut_interval` du tidyverse créer un facteur qui sépare l'intervalle d'années d'observation en 8 intervalles de tailles à peu près égales. On appellera **periode** ce facteur.

```
> library(tidyverse)
> periode <- cut_interval(taches$annee,n=8)
```

3. Utiliser les levels suivants pour le facteur **periode**.

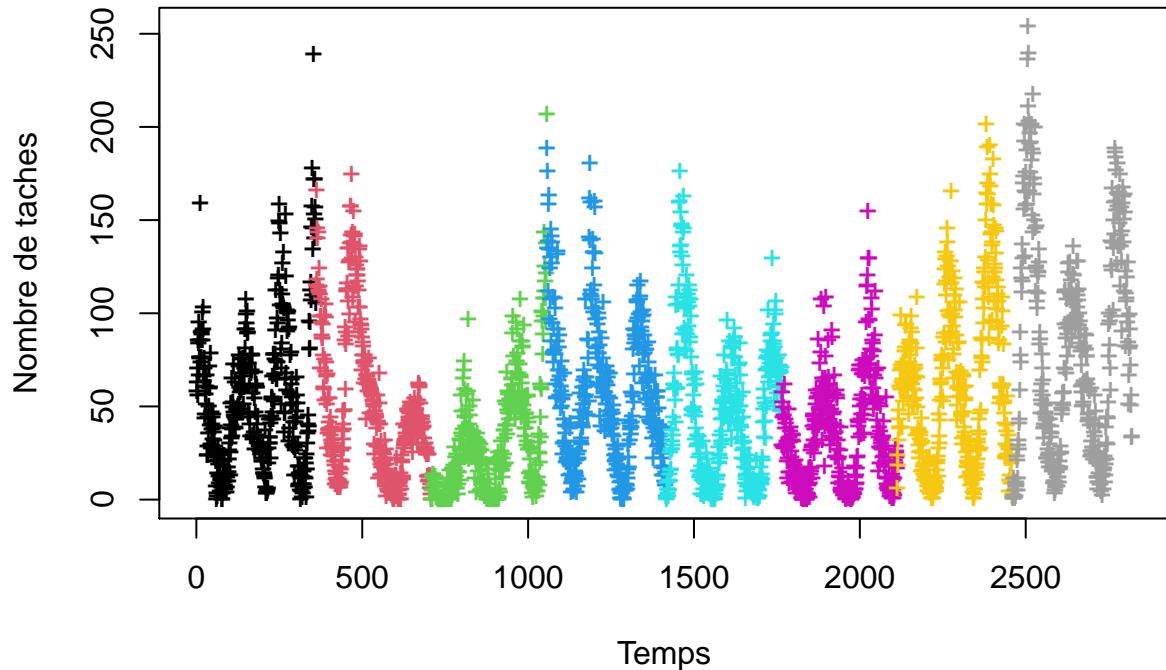
```
> couleurs <- c("yellow", "magenta", "orange", "cyan",
+           "grey", "red", "green", "blue")
> levels(periode) <- couleurs
```

4. Expliquer la sortie de la fonction

```
> coordx <- seq(along=taches[,1])
```

5. On crée une séquence avec un pas de 1 de longueur égale à la dimension de `taches[,1]`. Visualiser la série du nombre de tâches en utilisant une couleur différente pour chaque période.

```
> plot(coordx,taches[,1],xlab="Temps",ylab="Nombre de taches",
+      col=periode,type="p",pch="+")
```



**Exercice 4.4** (Layout). On reprend le jeu de données sur l'ozone. A l'aide de la fonction **layout** séparer la fenêtre graphique en deux lignes avec

1. un graphe sur la première ligne (nuage de points **maxO3 vs T12**) 1. 2 graphes sur la deuxième colonne (histogramme de **T12** et boxplot de **maxO3**).

```
```r
> layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE))
> plot(maxO3~T12,data=ozone)
> hist(ozone$T12)
> boxplot(ozone$maxO3)
```

```

\begin{center}\includegraphics{TUTO\_R\_files/figure-latex/unnamed-chunk-201-1} \end{center}

## 4.2 La grammaire ggplot2

Ce package propose de définir des graphes sur **R** en utilisant une **grammaire des graphiques** (tout comme **dplyr** pour manipuler les données). On peut trouver de la documentation sur ce package aux url <http://ggplot2.org> et <https://ggplot2-book.org>.

### 4.2.1 Premiers graphes ggplot2

Nous considérons un sous échantillon du jeu de données **diamonds** du package **ggplot2** (qui se trouve dans le **tidyverse**).

```
> library(tidyverse)
> set.seed(1234)
> diamonds2 <- diamonds[sample(nrow(diamonds), 5000),]
> summary(diamonds2)
   carat          cut      color      clarity      depth
Min.   :0.2000   Fair    :158   D: 640   SI1     :1189   Min.   :43.00
  
```

```

1st Qu.:0.4000  Good      : 455   E: 916   VS2     :1157   1st Qu.:61.10
Median :0.7000  Very Good:1094   F: 900   SI2     : 876   Median :61.80
Mean   :0.7969  Premium   :1280   G:1018   VS1     : 738   Mean   :61.76
3rd Qu.:1.0400  Ideal      :2013   H: 775   VVS2    : 470   3rd Qu.:62.50
Max.   :4.1300                    I: 481   VVS1    : 326   Max.   :71.60
                                J: 270   (Other): 244

  table          price           x           y
Min.  :49.00  Min.   : 365  Min.   : 0.000  Min.   :3.720
1st Qu.:56.00 1st Qu.: 945  1st Qu.: 4.720  1st Qu.:4.720
Median :57.00  Median : 2376  Median : 5.690  Median :5.700
Mean   :57.43  Mean   : 3917  Mean   : 5.728  Mean   :5.731
3rd Qu.:59.00 3rd Qu.: 5294  3rd Qu.: 6.530  3rd Qu.:6.520
Max.   :95.00  Max.   :18757  Max.   :10.000  Max.   :9.850

  z
Min.  :0.000
1st Qu.:2.920
Median :3.520
Mean   :3.538
3rd Qu.:4.030
Max.   :6.430

> help(diamonds)

```

Pour un jeu de données considéré, un graphe **ggplot** est défini à partir de **couches** que l'on assemblera avec l'opérateur **+**. Il faut à minima spécifier :

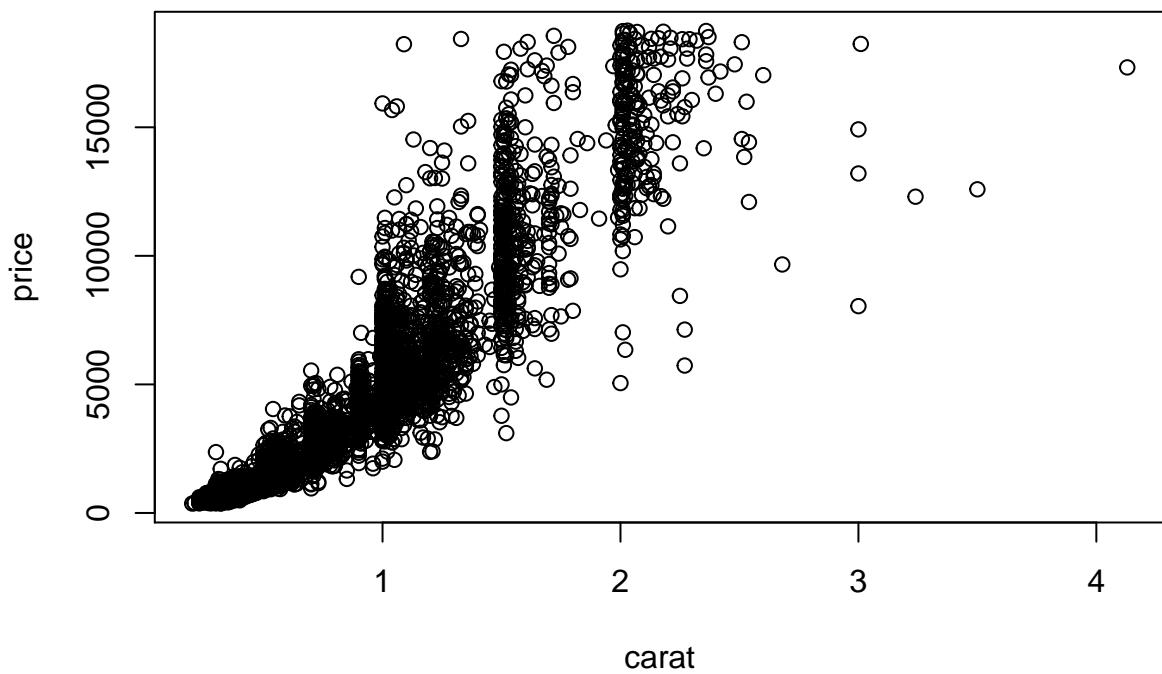
- les données
- les variables que l'on souhaite représenter
- le type de représentation (nuage de points, boxplot...).

Il existe un verbe pour définir chacune de ces couches :

- **ggplot** pour les données
- **aes** (aesthetics) pour les variables
- **geom\_** pour le type de représentation.

On peut obtenir le nuage de points **carat** vs **price** avec la fonction **plot** :

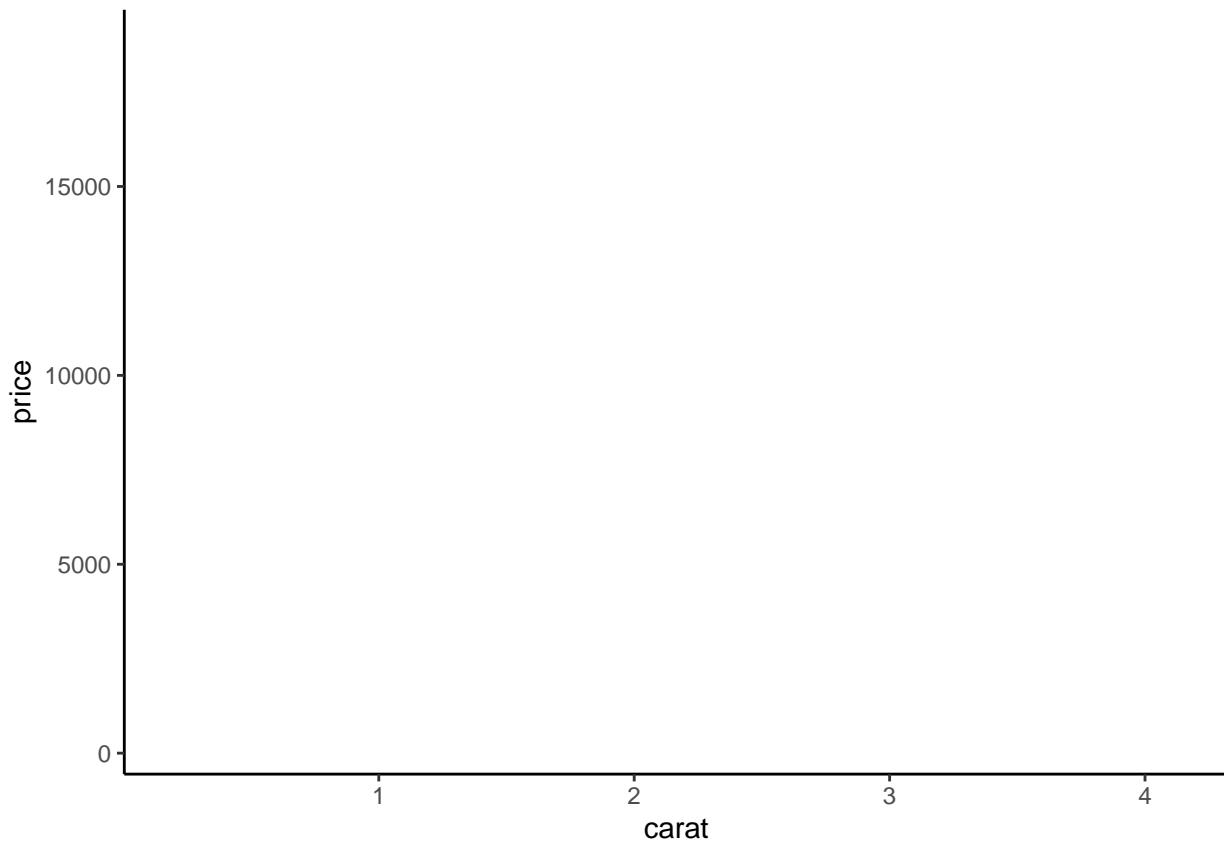
```
> plot(price~carat,data=diamonds2)
```



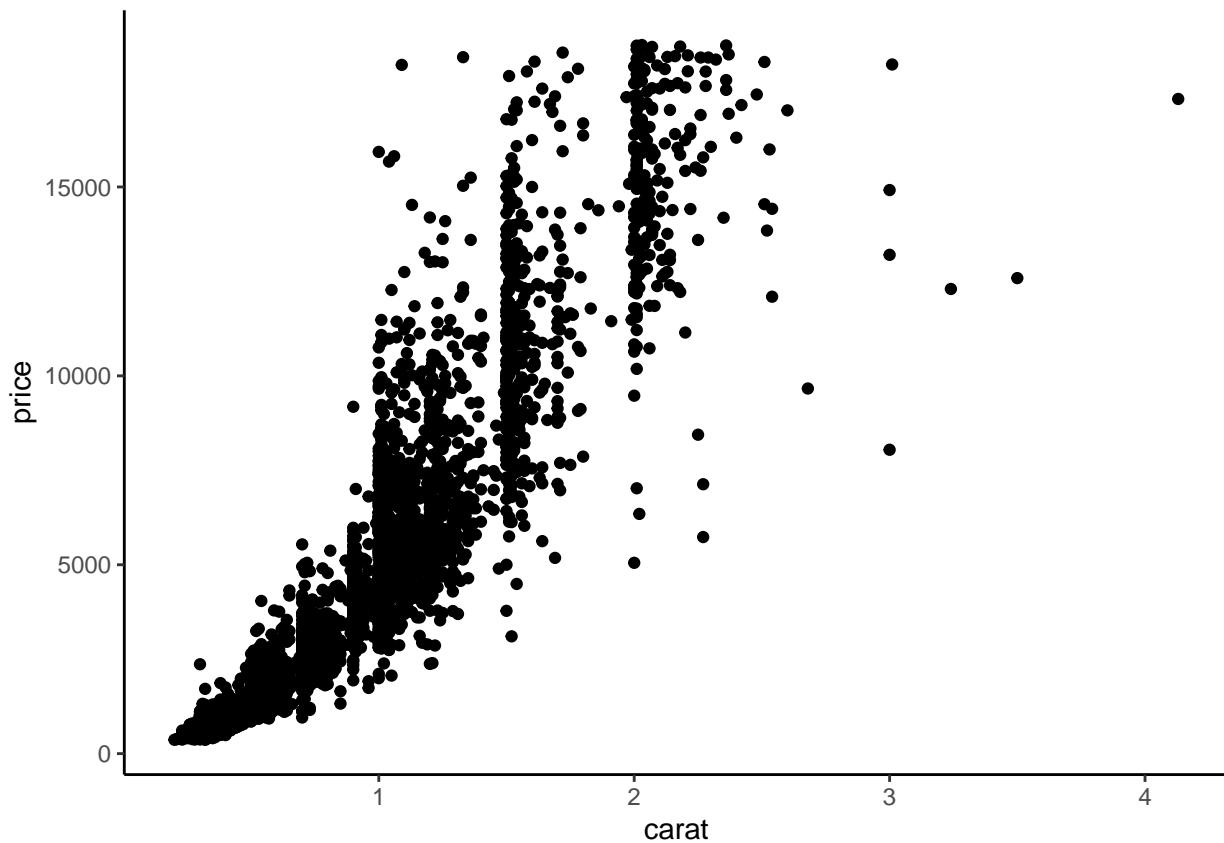
Avec `ggplot`, on va faire

```
> ggplot(diamonds2) #rien
```

```
> ggplot(diamonds2)+aes(x=carat,y=price) #rien
```



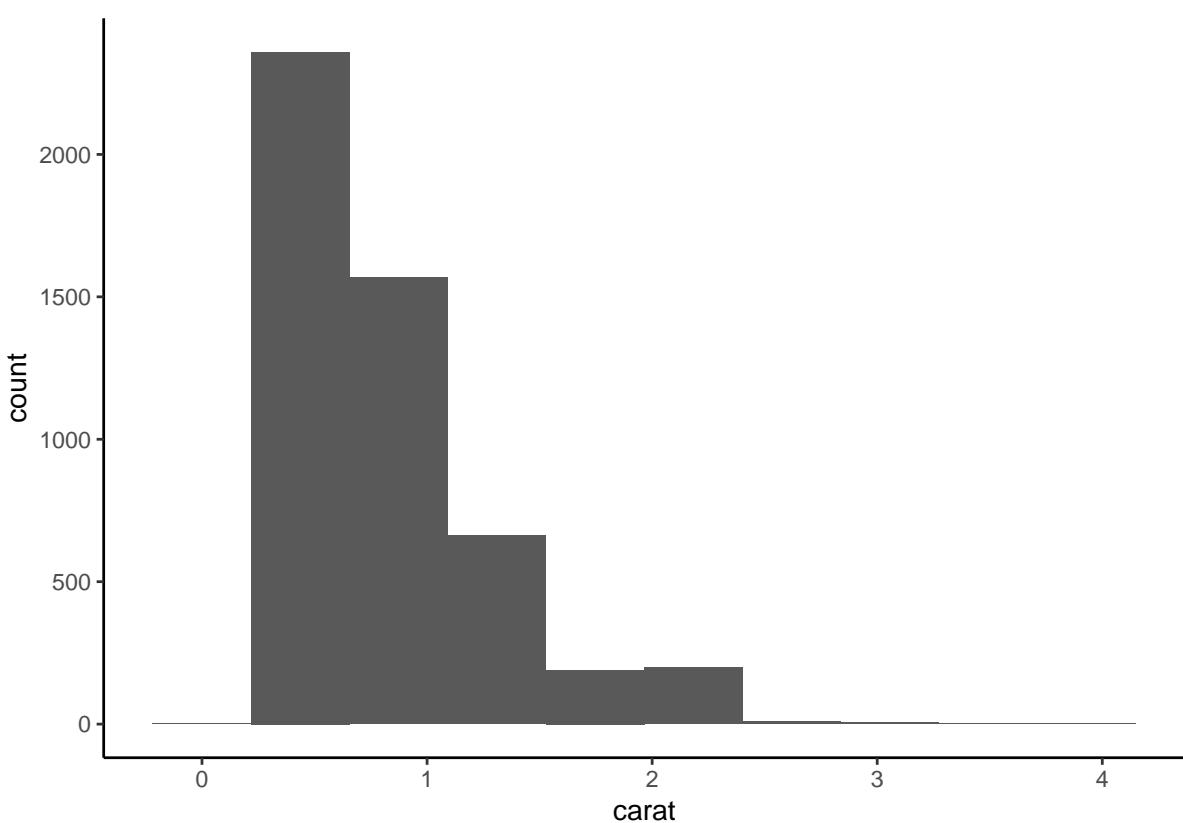
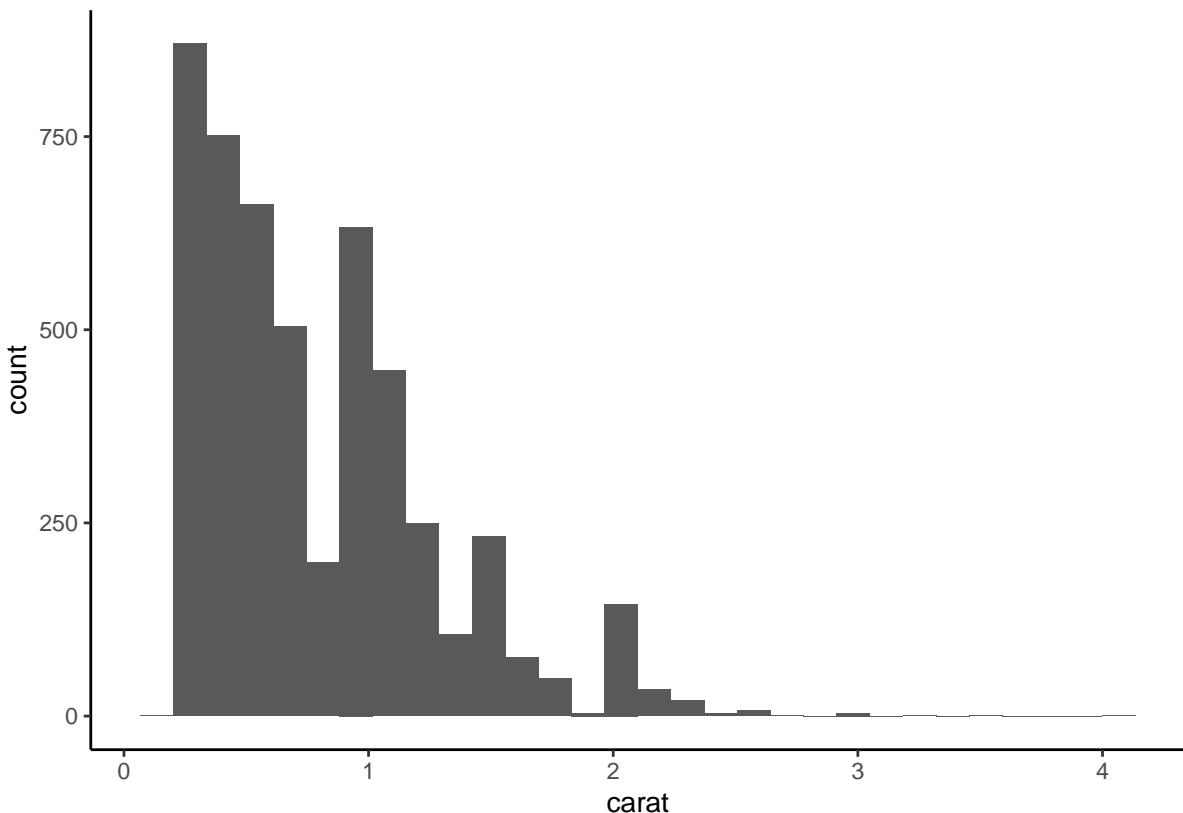
```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point() #bon
```



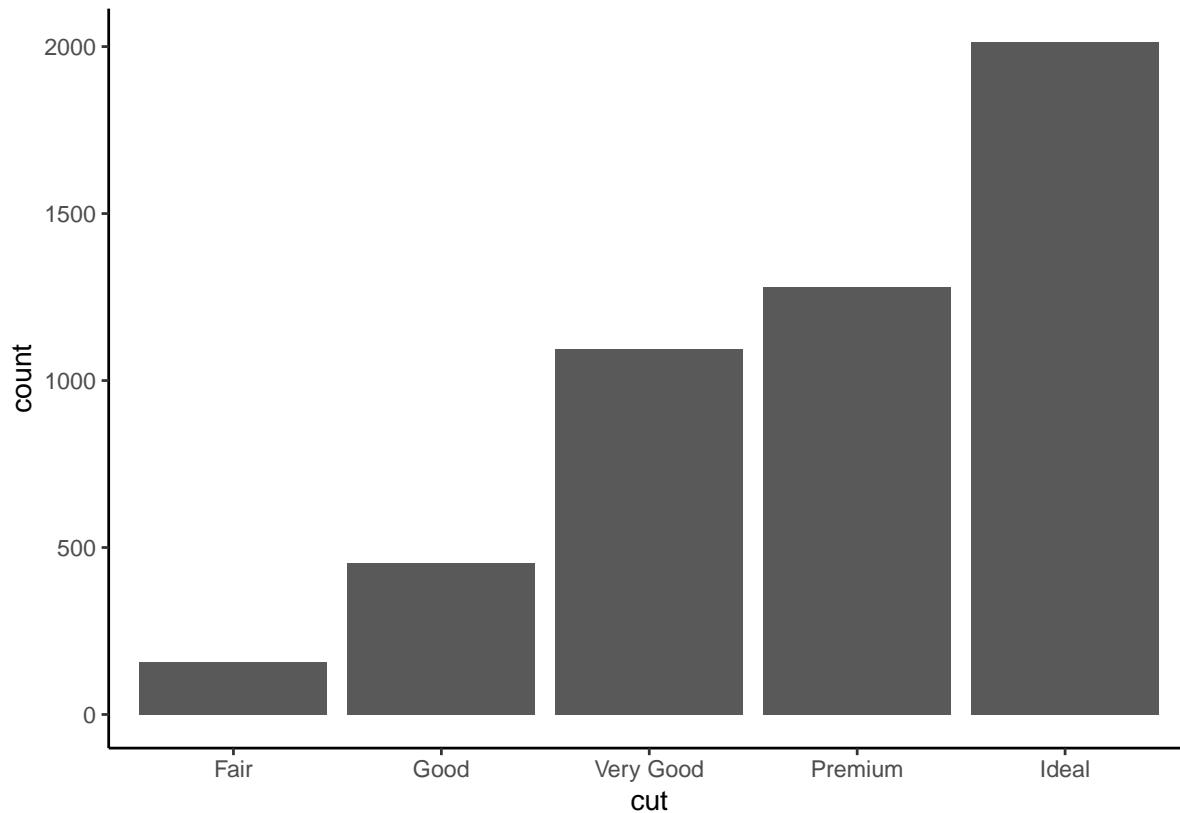
Exercice 4.5 (Premiers graphes ggplot).

1. Tracer l'histogramme de la variable **carat** (utiliser `geom_histogram`).
2. Tracer l'histogramme de la variable **carat** avec 10 classes (`help(geom_histogram)`).
3. Tracer le diagramme à batons de la variable **cut** (utiliser `geom_bar`).

```
> ggplot(diamonds2) + aes(x=carat) + geom_histogram()
```



```
> ggplot(diamonds2)+aes(x=cut)+geom_bar()
```



La syntaxe **ggplot** se construit à partir d'éléments indépendants qui définissent la grammaire de **ggplot**. Les principaux verbes sont :

- **Data** (**ggplot**) : les données au format **dataframe** ou **tibble**.
- **Aesthetics** (**aes**) : pour spécifier les variables à représenter dans le graphe.
- **Geometrics** (**geom\_...**) : le type de graphe (nuage de points, histogramme...).
- **Statistics** (**stat\_...**) : utile pour spécifier des transformations des données nécessaires pour obtenir le graphe.
- **Scales** (**scale\_...**) : pour contrôler les paramètres permettant d'affiner le graphe (changement de couleurs, paramètres des axes...).

Tous ces éléments sont reliés avec le symbole **+**.

#### 4.2.2 Data et aesthetics

Ces deux verbes sont à utiliser pour tous les graphes **ggplot**. Le verbe **ggplot** servira à définir le jeu de données que l'on souhaite utiliser. Si le code est bien fait, nous n'aurons plus à utiliser le nom du jeu de données par la suite pour construire le graphe. Le verbe **aes** est quant à lui utile pour spécifier le nom des variables que l'on souhaite visualiser. Par exemple, pour le nuage de points **price vs carat** la syntaxe devra débuter par

```
> ggplot(diamonds2)+aes(x=carat,y=price)
```

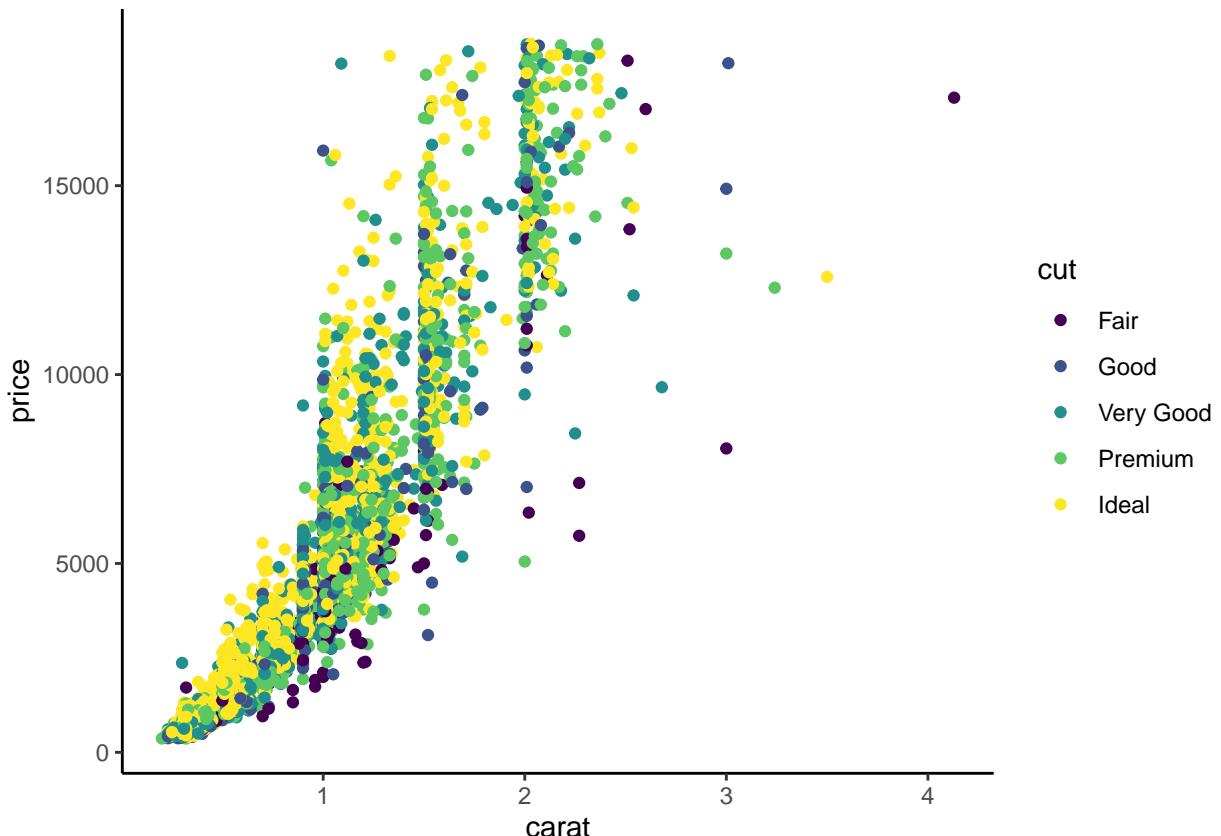
Les variables peuvent également être utilisées pour colorier des points ou des barres, définir des tailles... Dans ce cas on pourra renseigner les arguments **color**, **size**, **fill** dans la fonction **aes**. Par exemple

```
> ggplot(diamonds2)+aes(x=carat,y=price,color=cut)
```

#### 4.2.3 Geometrics

Ce verbe décrira le type de représentation souhaitée. Pour un nuage de points, on utilisera par exemple **geom\_point** :

```
> ggplot(diamonds2)+aes(x=carat,y=price,color=cut)+geom_point()
```



On observe que **ggplot** ajoute la légende automatiquement. Voici les principaux exemples de **geometrics** :

TABLE 3: Principaux geometrics

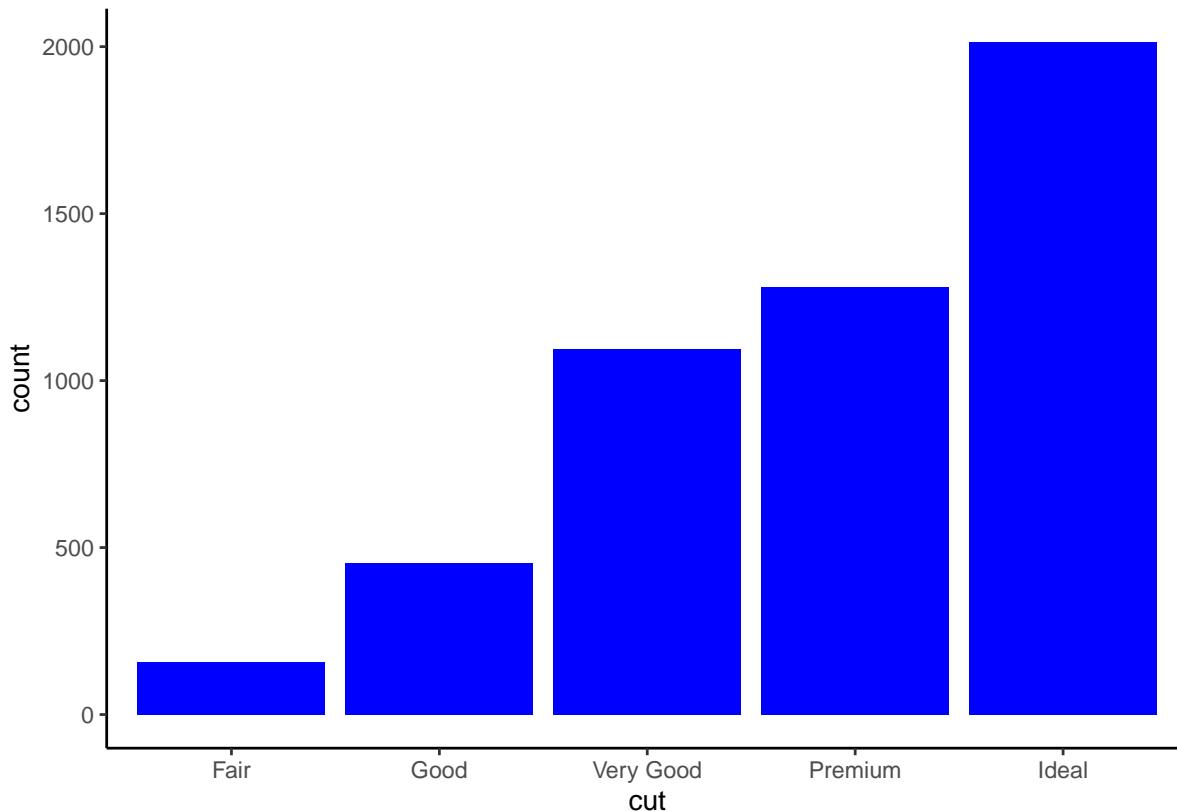
| Geom             | Description                  | Aesthetics                             |
|------------------|------------------------------|--|
| geom_point()     | nuage de points              | x, y, shape, fill                      |
| geom_line()      | Ligne (ordonnée selon x)     | x, y, linetype                         |
| geom_abline()    | Ligne                        | slope, intercept                       |
| geom_path()      | Ligne (ordonnée par l'index) | x, y, linetype                         |
| geom_text()      | Texte                        | x, y, label, hjust, vjust              |
| geom_rect()      | Rectangle                    | xmin, xmax, ymin, ymax, fill, linetype |
| geom_polygon()   | Polygone                     | x, y, fill, linetype                   |
| geom_segment()   | Segment                      | x, y, xend, yend, fill, linetype       |
| geom_bar()       | Diagramme en barres          | x, fill, linetype, weight              |
| geom_histogram() | Histogramme                  | x, fill, linetype, weight              |
| geom_boxplot()   | Boxplot                      | x, fill, weight                        |
| geom_density()   | Densité                      | x, y, fill, linetype                   |
| geom_contour()   | Lignes de contour            | x, y, fill, linetype                   |

| Geom          | Description                        | Aesthetics           |
|---------------|------------------------------------|----------------------|
| geom_smooth() | Lisseur (linéaire ou non linéaire) | x, y, fill, linetype |
| Tous          |                                    | color, size, group   |

**Exercice 4.6** (Diagrammes en barres).

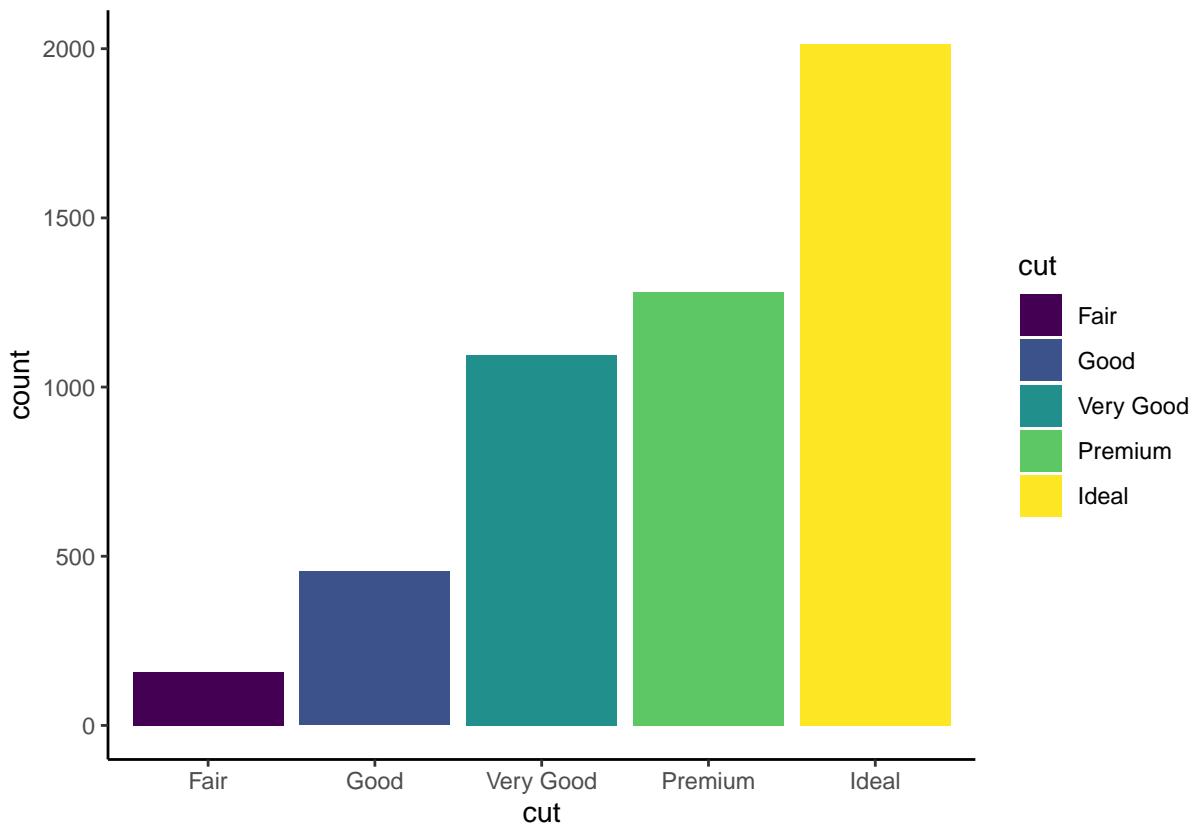
- Tracer le diagramme en barres de la variable **cut** avec des barres bleues.

```
> ggplot(diamonds2)+aes(x=cut)+geom_bar(fill="blue")
```



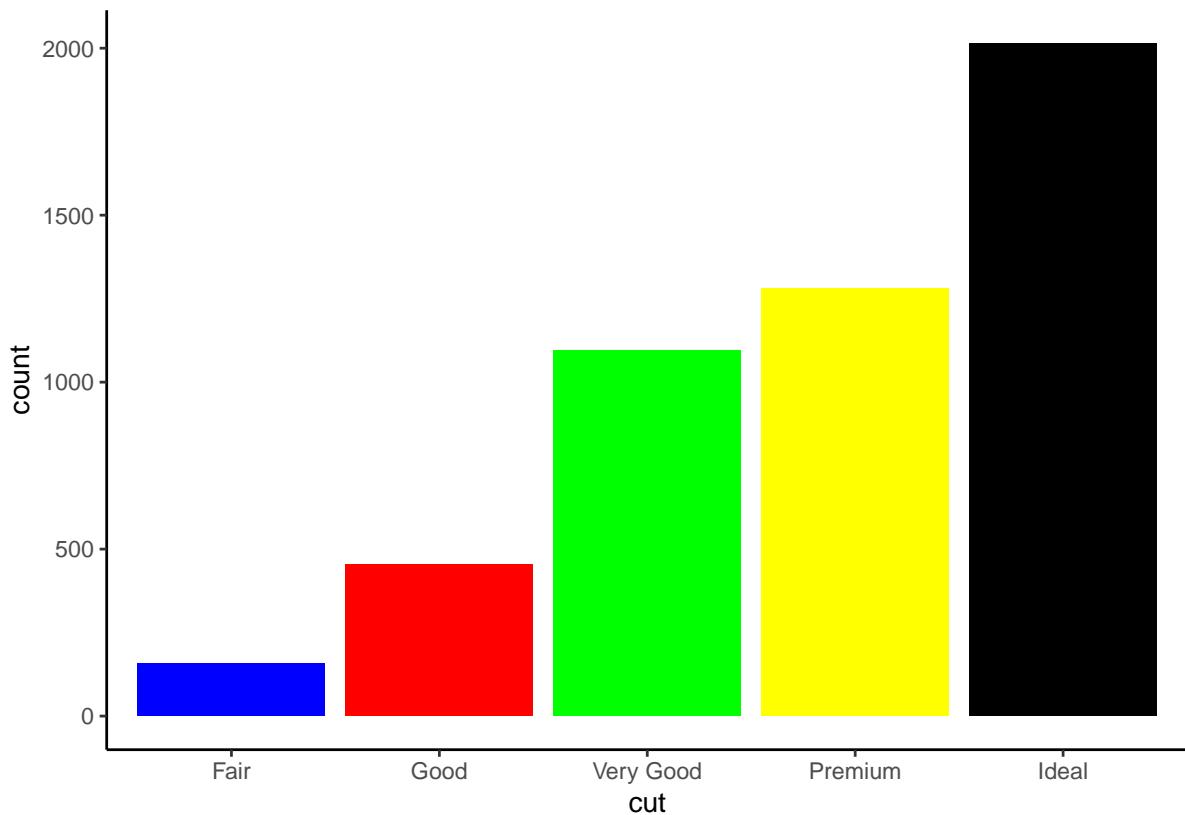
- Tracer le diagramme en barres de la variable **cut** avec une couleur pour chaque modalité de **cut** ainsi qu'une légende qui permet de repérer la couleur.

```
> ggplot(diamonds2)+aes(x=cut,fill=cut)+geom_bar()
```



3. Tracer le diagramme en barres de la variable **cut** avec une couleur pour chaque modalité que vous choisisrez (et sans légende).

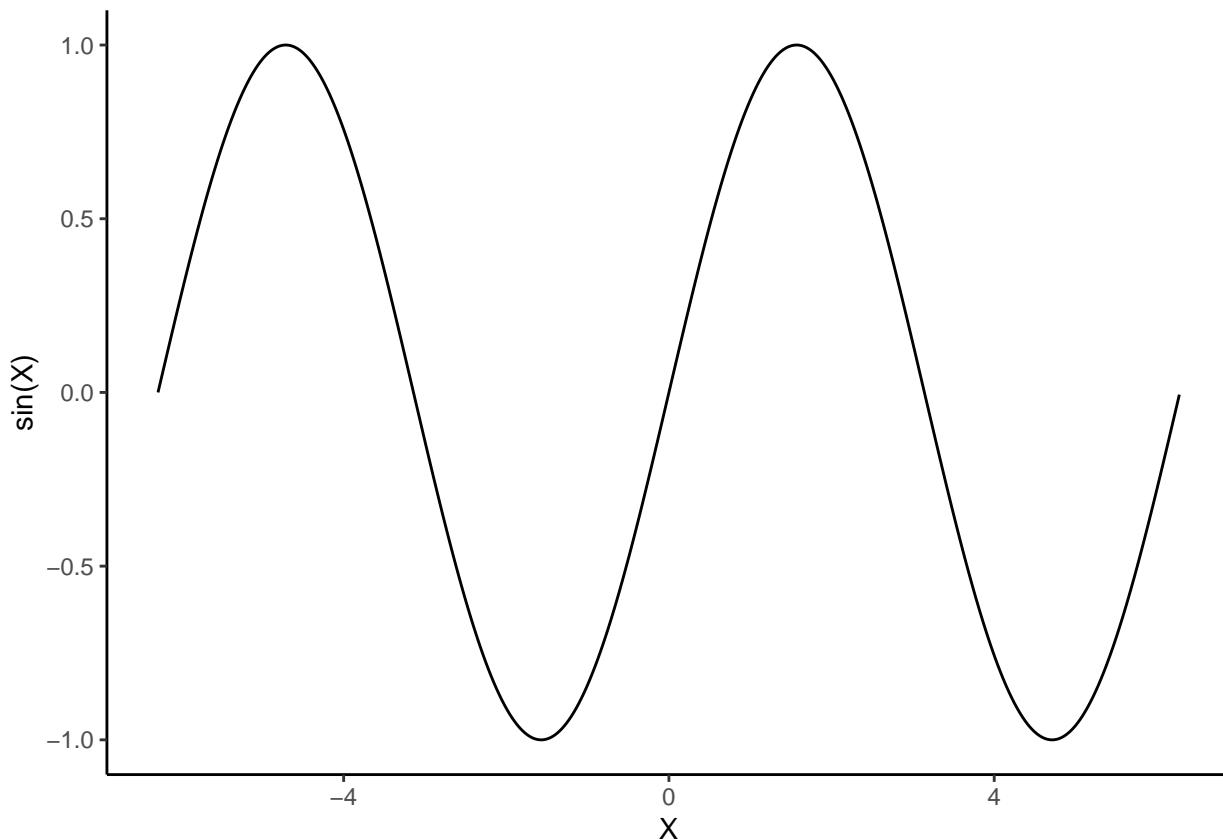
```
> ggplot(diamonds2)+aes(x=cut)+geom_bar(fill=c("blue","red","green","yellow","black"))
```



#### 4.2.4 Statistics

Certains graphes nécessitent des calculs d'indicateurs statistiques pour être tracé. C'est par exemple le cas pour le diagamme en barres et l'histogramme où il faut calculer des hauteurs des barres. Les transformations simples peuvent se faire rapidement, on peut par exemple tracer la fonction **sinus** avec

```
> D <- data.frame(X=seq(-2*pi,2*pi,by=0.01))
> ggplot(D)+aes(x=X,y=sin(X))+geom_line()
```

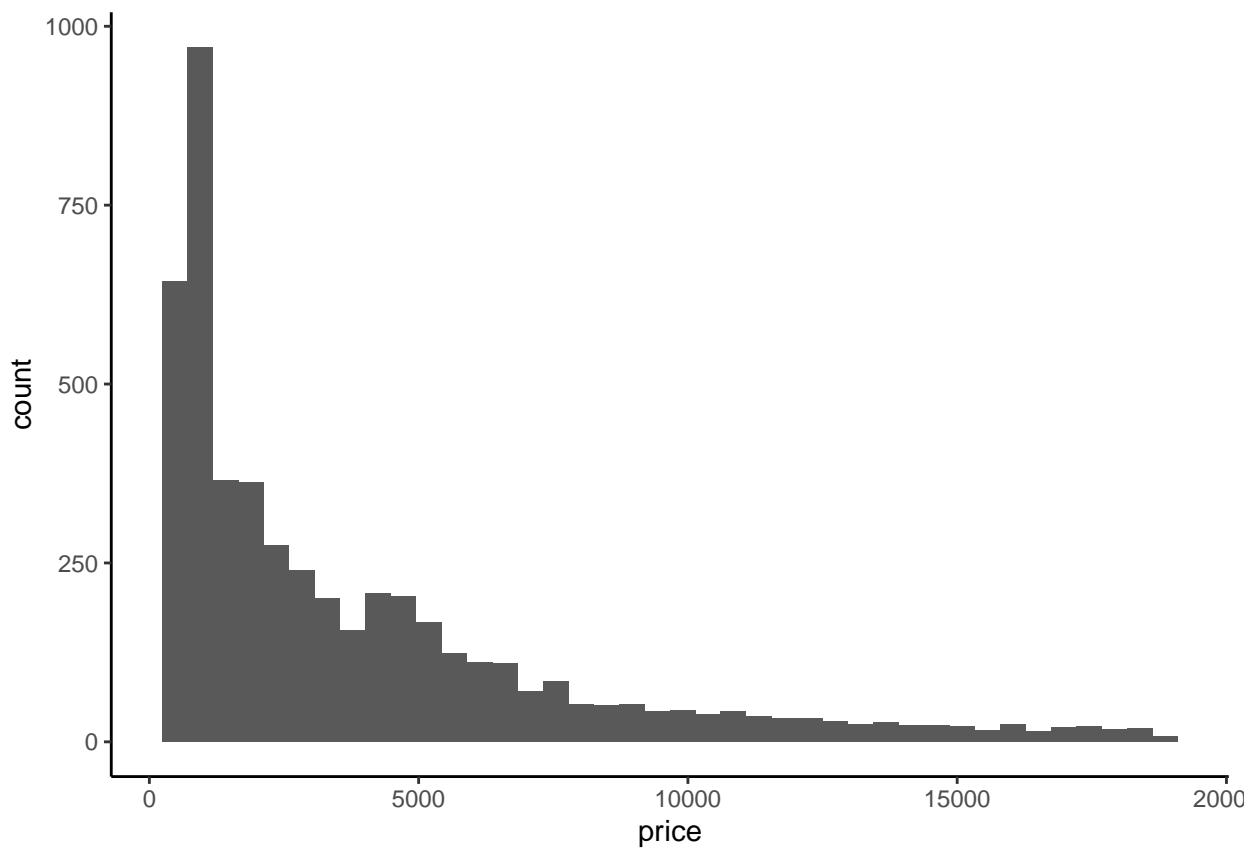


La transformation est spécifiée dans la fonction **aes**. Pour des transformations plus complexes, nous devons utiliser des **statistics**. Une fonction **stat** permet de définir des nouvelles variables à partir du jeu de données initial, il est ensuite possible de représenter ces nouvelles variables. Par exemple, la fonction **stat\_bin**, qui est utilisée par défaut pour construire des histogrammes, produit les variables suivantes :

- **count**, le nombre d'observations dans chaque classes.
- **density**, la valeur de la densité des observations dans chaque classe (fréquence divisée par largeur de la classe).
- **x**, le centre de la classe.

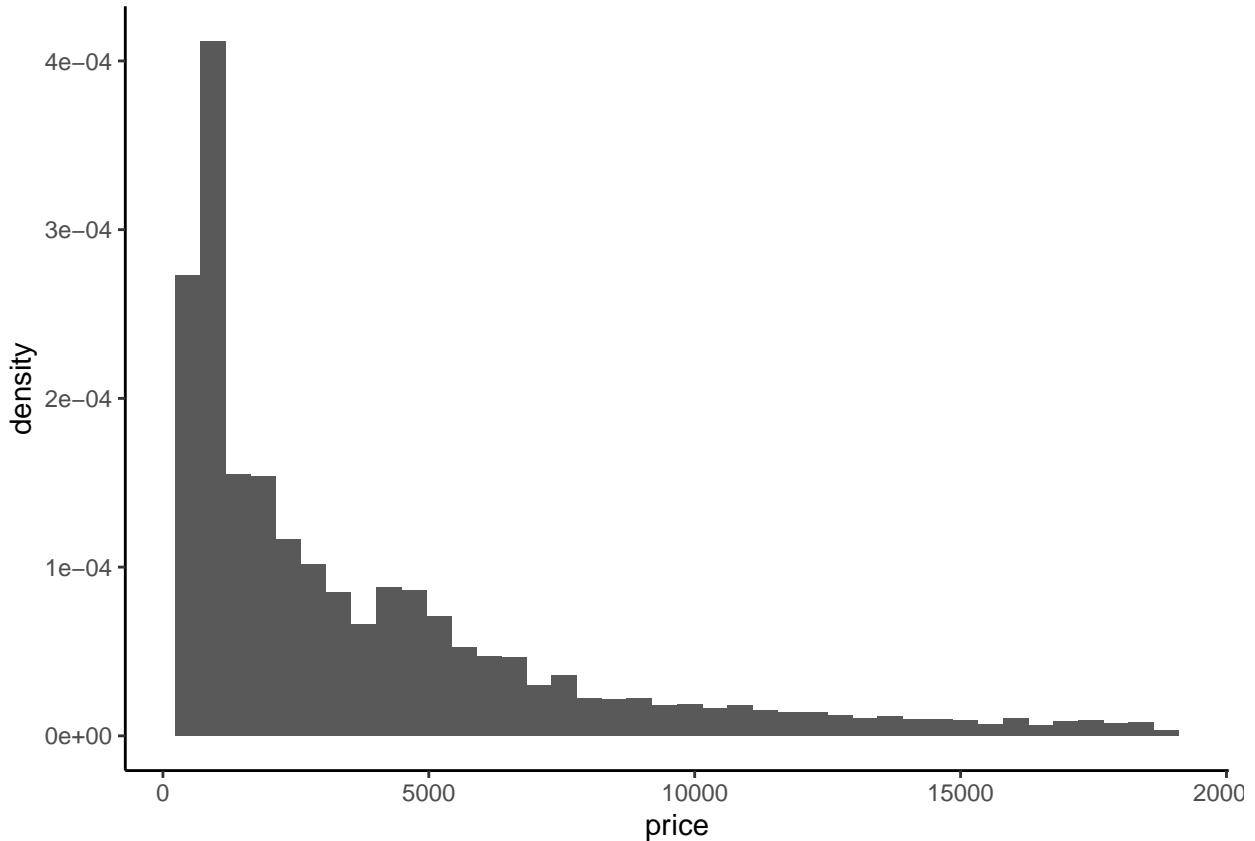
Par défaut *geom\_histogram* fait appel à cette fonction **stat\_binet** représentée sur l'axe *y* le nombre d'observations dans chaque classe (la variable **count**).

```
> ggplot(diamonds2)+aes(x=price)+geom_histogram(bins=40)
```



Si on souhaite une autre variable issue de `stat_bin`, comme par exemple la densité, il faudra utiliser

```
> ggplot(diamonds2)+aes(x=price,y=..density..)+geom_histogram(bins=40)
```



Il est possible d'utiliser les fonctions **stat\_** à la place des **geom\_** pour certaines représentations. Chaque fonction **stat\_** possède par défaut un **geom\_** et réciproquement. On peut par exemple obtenir le même graphe que précédemment avec

```
> ggplot(diamonds2)+aes(x=price,y=..density..)+stat_bin()
```

Voici quelques exemple de fonctions **stat\_**

TABLE 4: Exemples de statistics.

| Stat            | Description           | Paramètres       |
|-----------------|-----------------------|------------------|
| stat_identity() | aucune transformation |                  |
| stat_bin()      | Count                 | binwidth, origin |
| stat_density()  | Density               | adjust, kernel   |
| stat_smooth()   | Smoother              | method, se       |
| stat_boxplot()  | Boxplot               | coef             |

*stat* et *geom* ne sont pas toujours simples à combiner. Nous recommandons d'utiliser **geom** lorsqu'on débute avec **ggplot**, les **statistics** par défaut ne doivent en effet être changés que rarement.

**Exercice 4.7** (Diagramme en barres "très simple" ...).

On considère une variable qualitative  $X$  dont la loi est donnée par

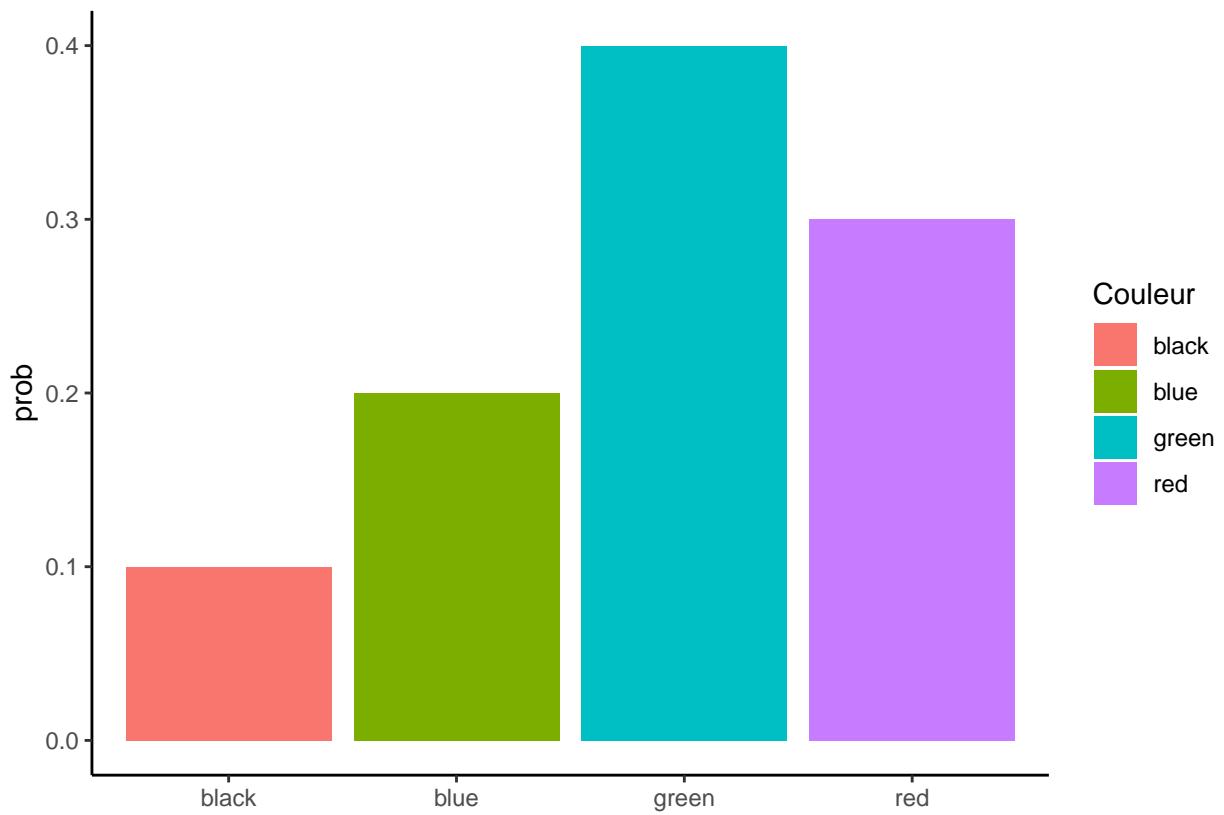
$$P(X = \text{red}) = 0.3, P(X = \text{blue}) = 0.2, P(X = \text{green}) = 0.4, P(X = \text{black}) = 0.1$$

Représenter cette distribution de probabilité avec un diagramme en barres.

```

> X <- data.frame(X1=c("red","blue","green","black"),prob=c(0.3,0.2,0.4,0.1))
> ggplot(X)+aes(x=X1,y=prob,fill=X1)+geom_bar(stat="identity")+
+   labs(fill="Couleur")+xlab("")

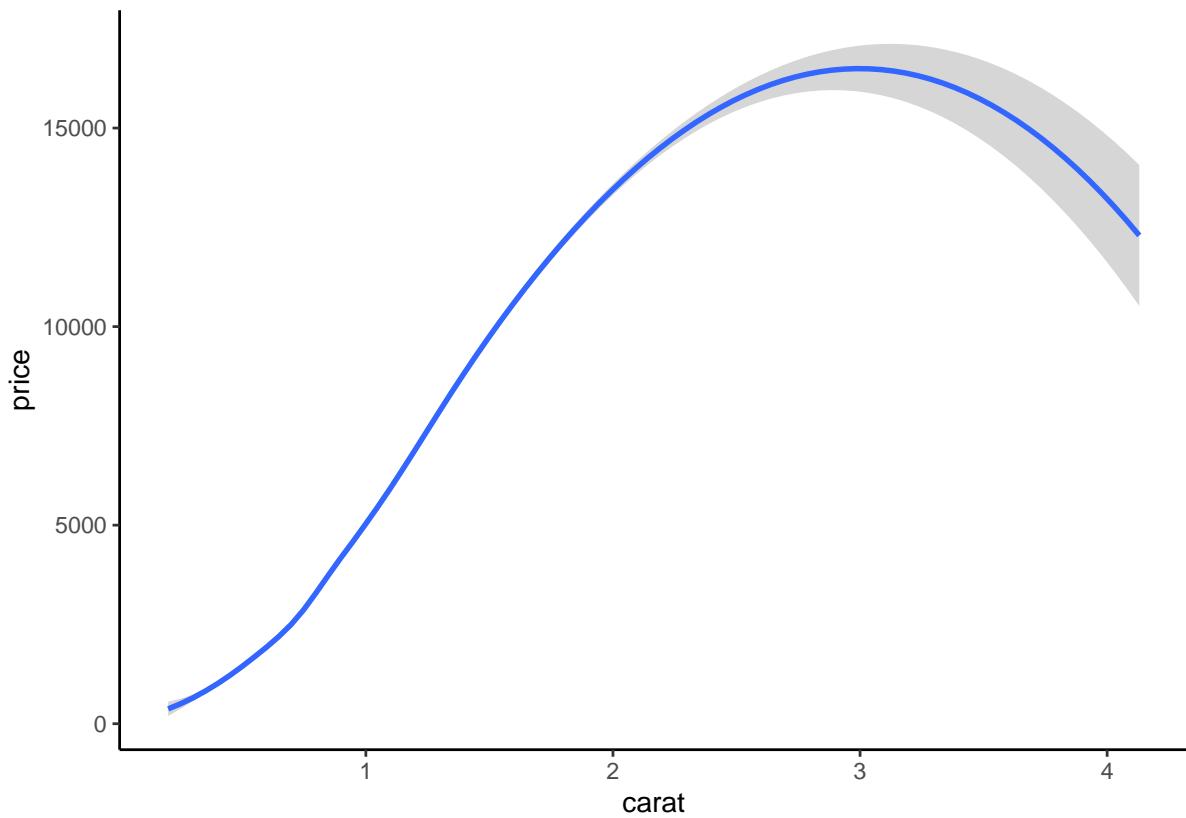
```

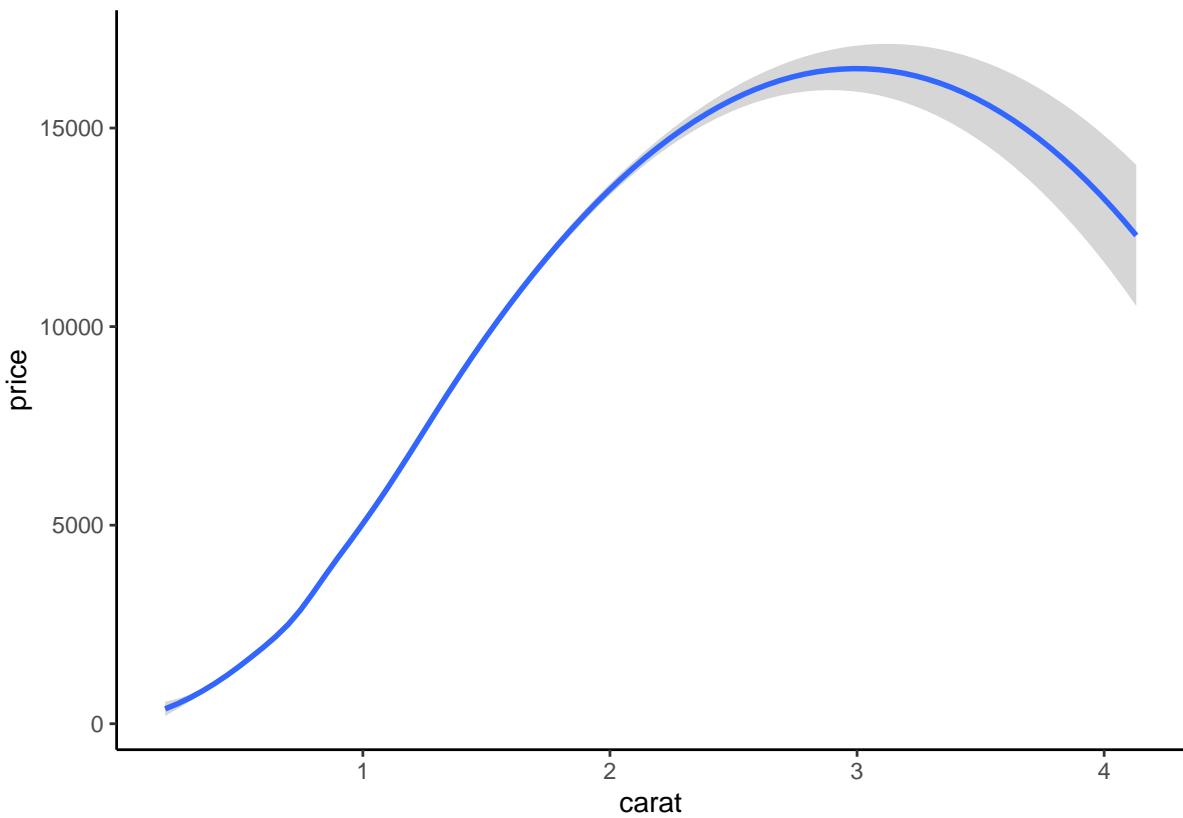


#### Exercice 4.8 (Lissage).

- Représenter le lissage non linéaire de la variable `price` contre la variable `carat` à l'aide de `geom_smooth` puis de `stat_smooth`.

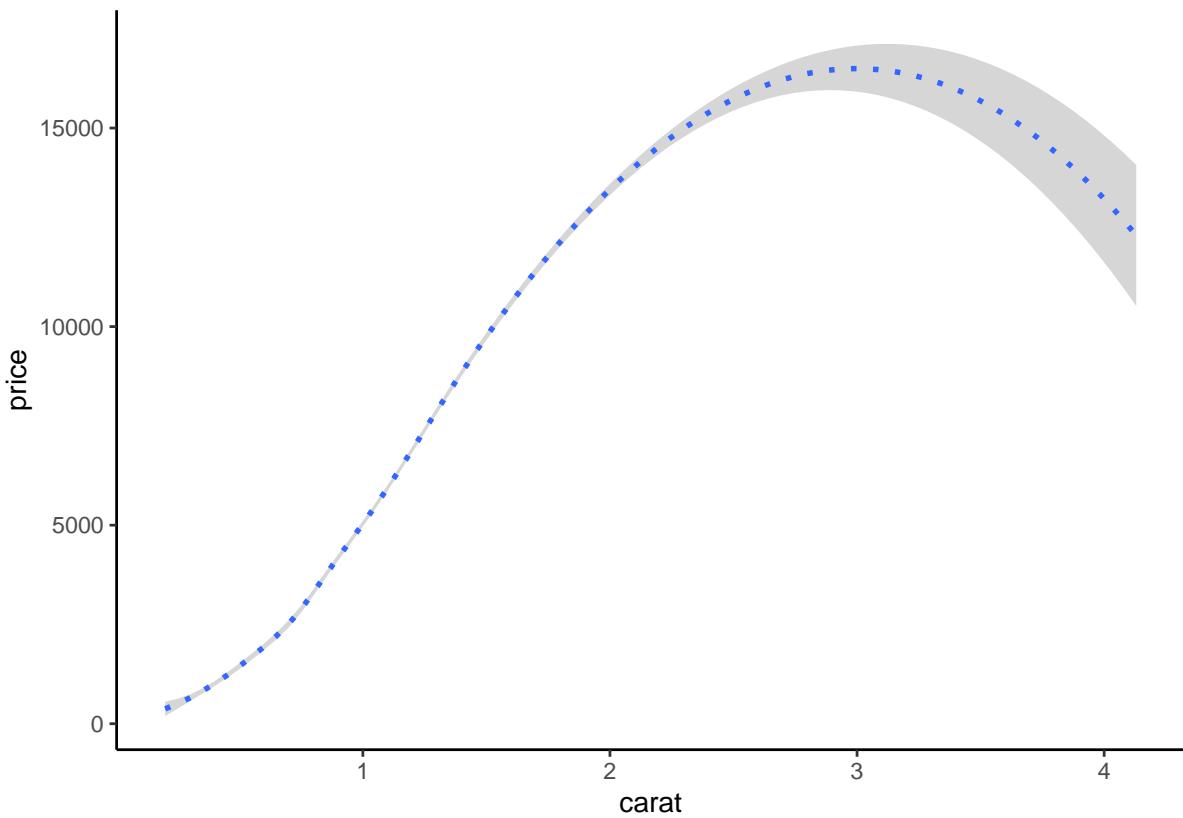
```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_smooth(method="loess")
```



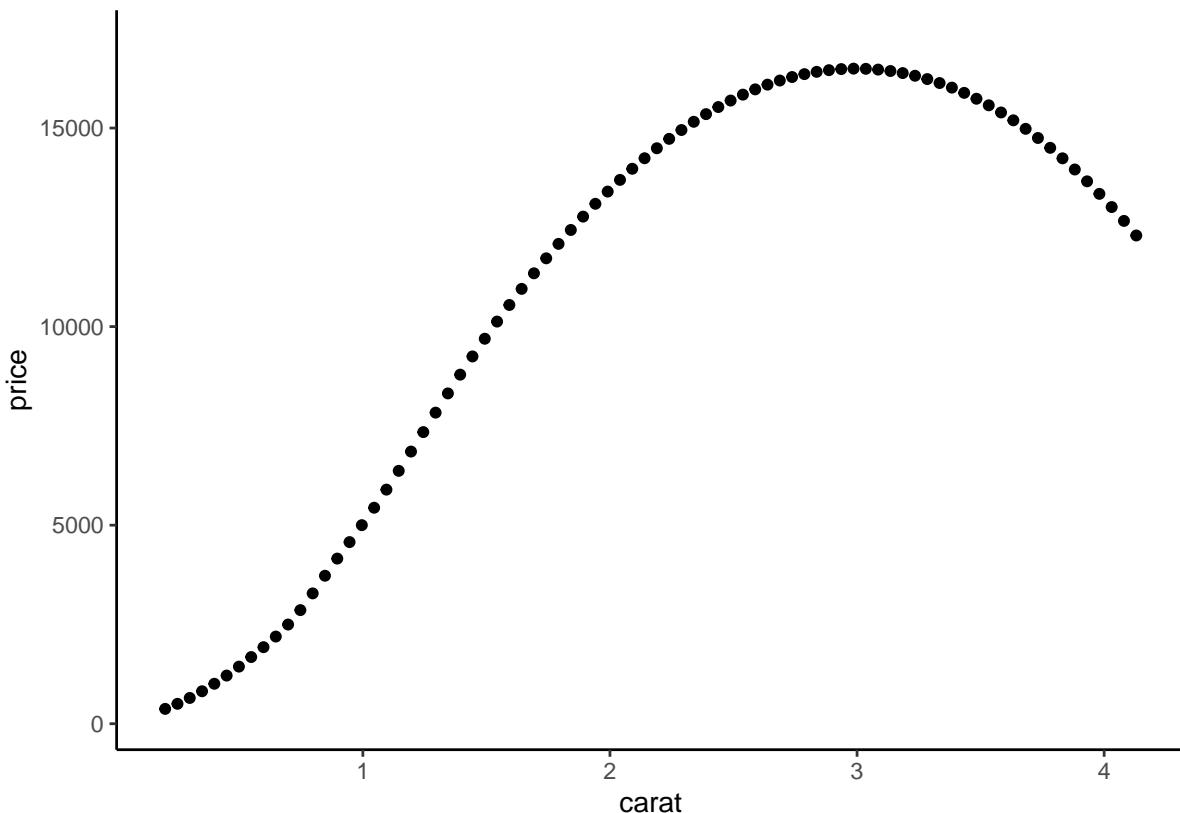


2. Même question mais avec une ligne en pointillés à la place d'un trait plein.

```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_smooth(method="loess",linetype="dotted")
```



```
> ggplot(diamonds2)+aes(x=carat,y=price)+stat_smooth(method="loess",geom="point")
```



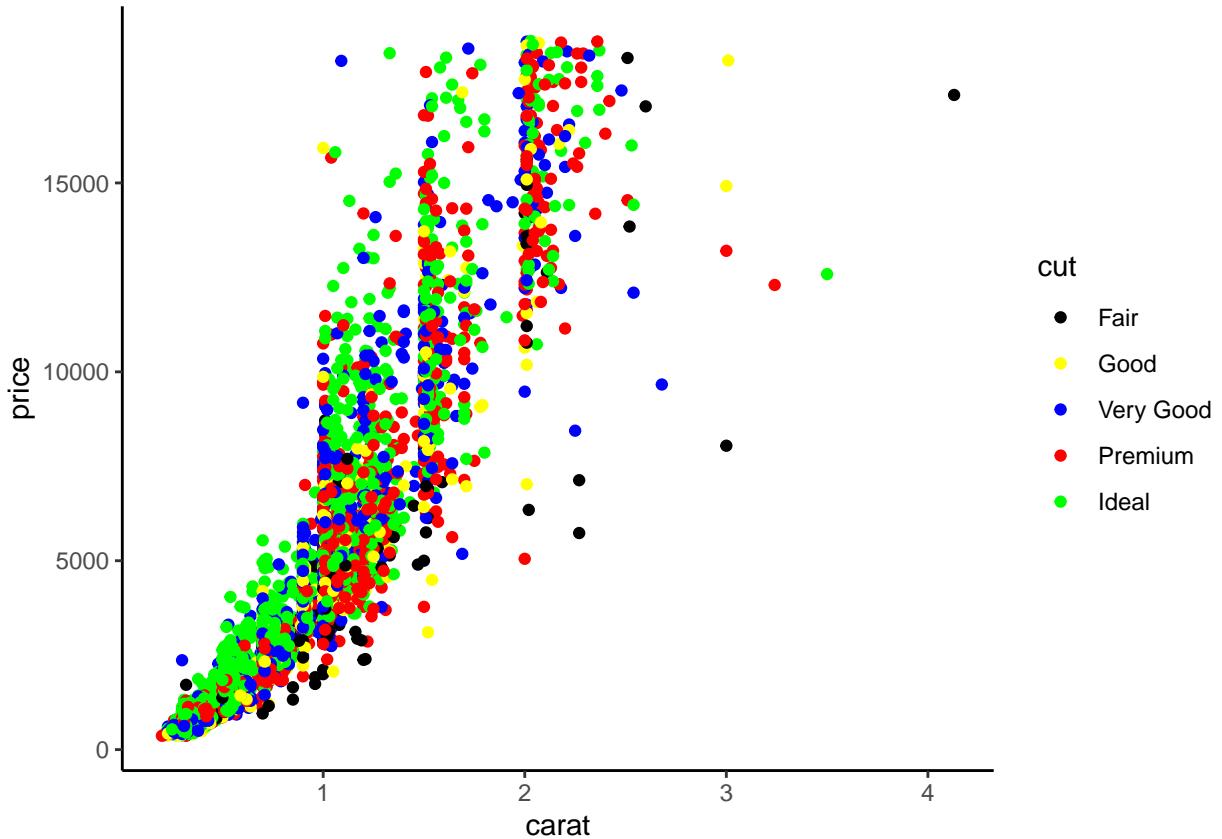
#### 4.2.5 Scales

Les échelles (**scales**) contrôlent tout un tas d'options telles que des changements de couleurs, d'échelles ou de limites d'axes, de symboles, etc... L'utilisation n'est pas simple et nécessite de la pratique. On utilise généralement ce verbe à la dernière étape de construction du graphe. La syntaxe est définie comme suit :

- début : `scale_`.
- ajout de l'aesthetics que l'on souhaite modifier (`color`, `fill`, `x_`).
- fin : nom de l'échelle (`manual`, `identity`...)

Par exemple,

```
> ggplot(diamonds2)+aes(x=carat,y=price,color=cut)+geom_point()+
+   scale_color_manual(values=c("Fair"="black","Good"="yellow",
+                           "Very Good"="blue","Premium"="red","Ideal"="green"))
```



Voici quelques exemples des principales échelles :

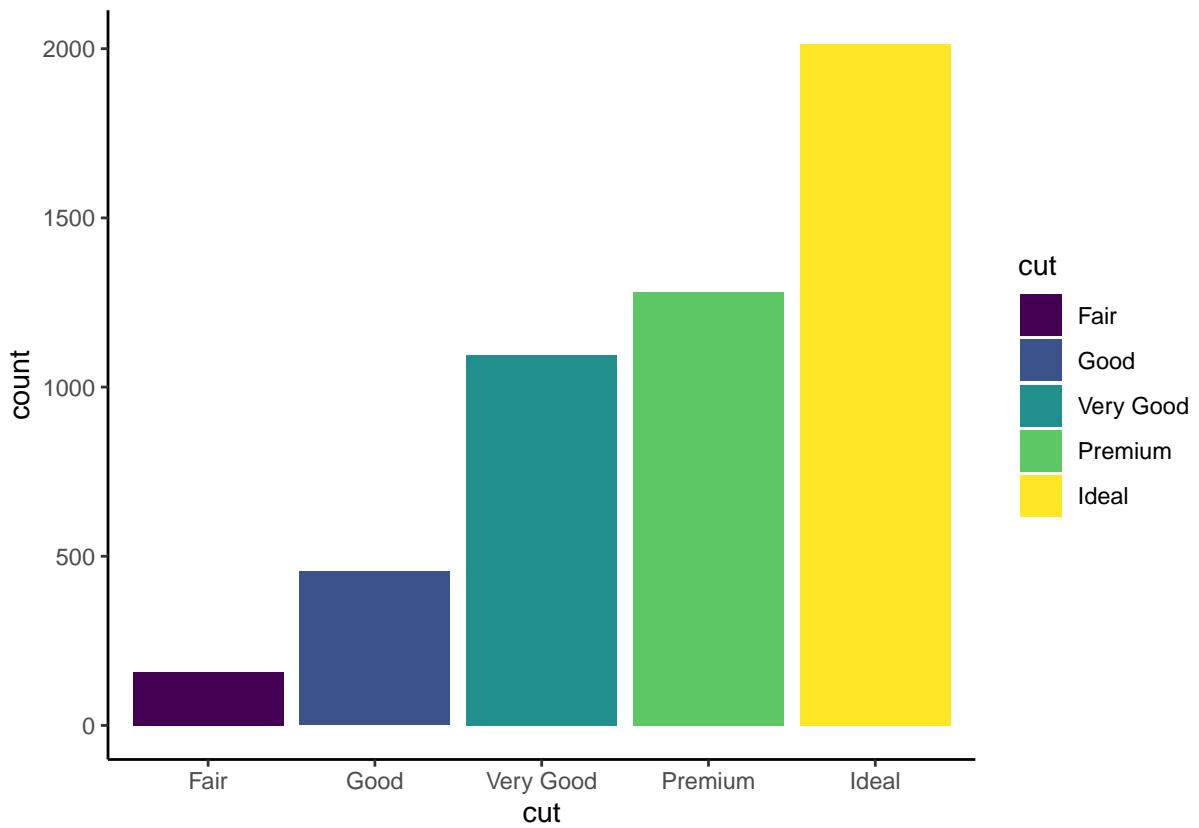
TABLE 5: Exemples d'échelles

| aes                     | Discret  | Continu    |
|-------------------------|----------|------------|
| Couleur (color et fill) | brewer   | gradient   |
| -                       | grey     | gradient2  |
| -                       | hue      | gradientn  |
| -                       | identity |            |
| -                       | manual   |            |
| Position (x et y)       | discrete | continuous |
| -                       |          | date       |
| Forme                   | shape    |            |
| -                       | identity |            |
| -                       | manual   |            |
| Taille                  | identity | size       |
| -                       | manual   |            |

Nous présentons quelques exemples d'utilisation des échelles :

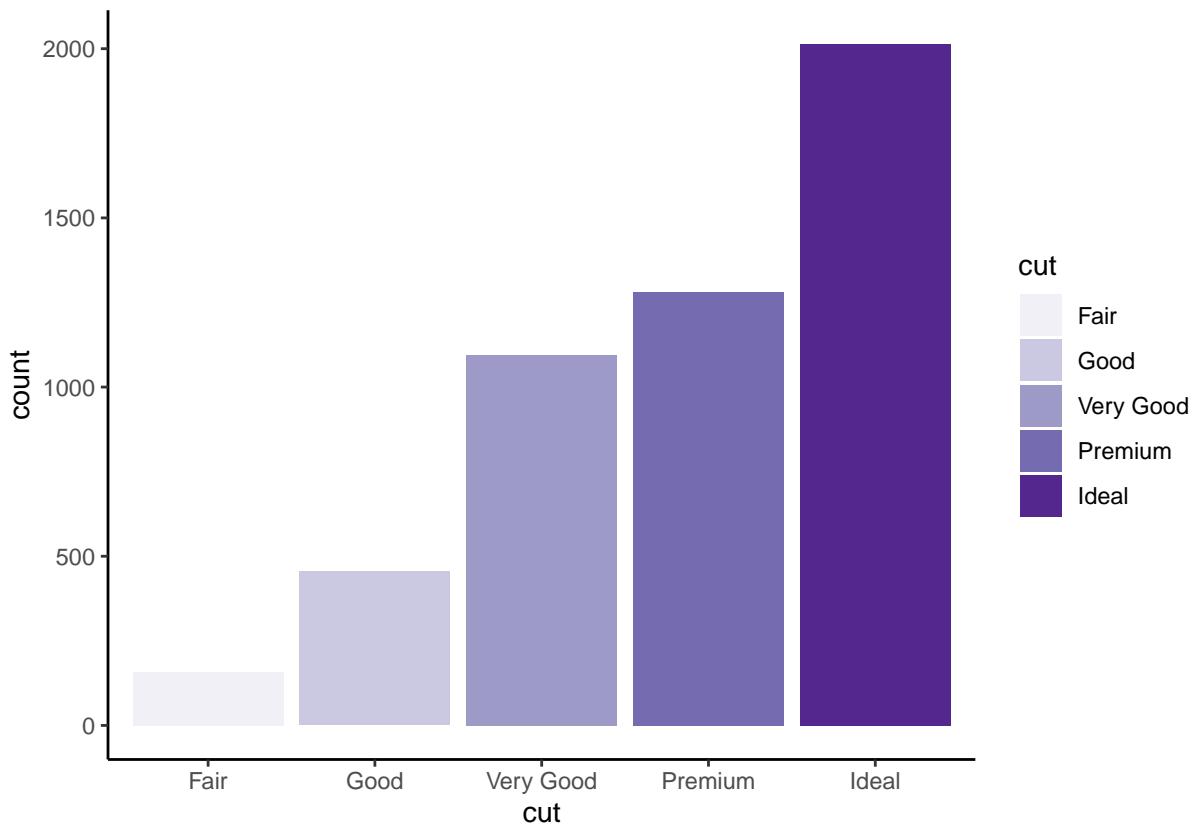
— Couleur dans un diagramme en barres

```
> p1 <- ggplot(diamonds2)+aes(x=cut)+geom_bar(aes(fill=cut))
> p1
```



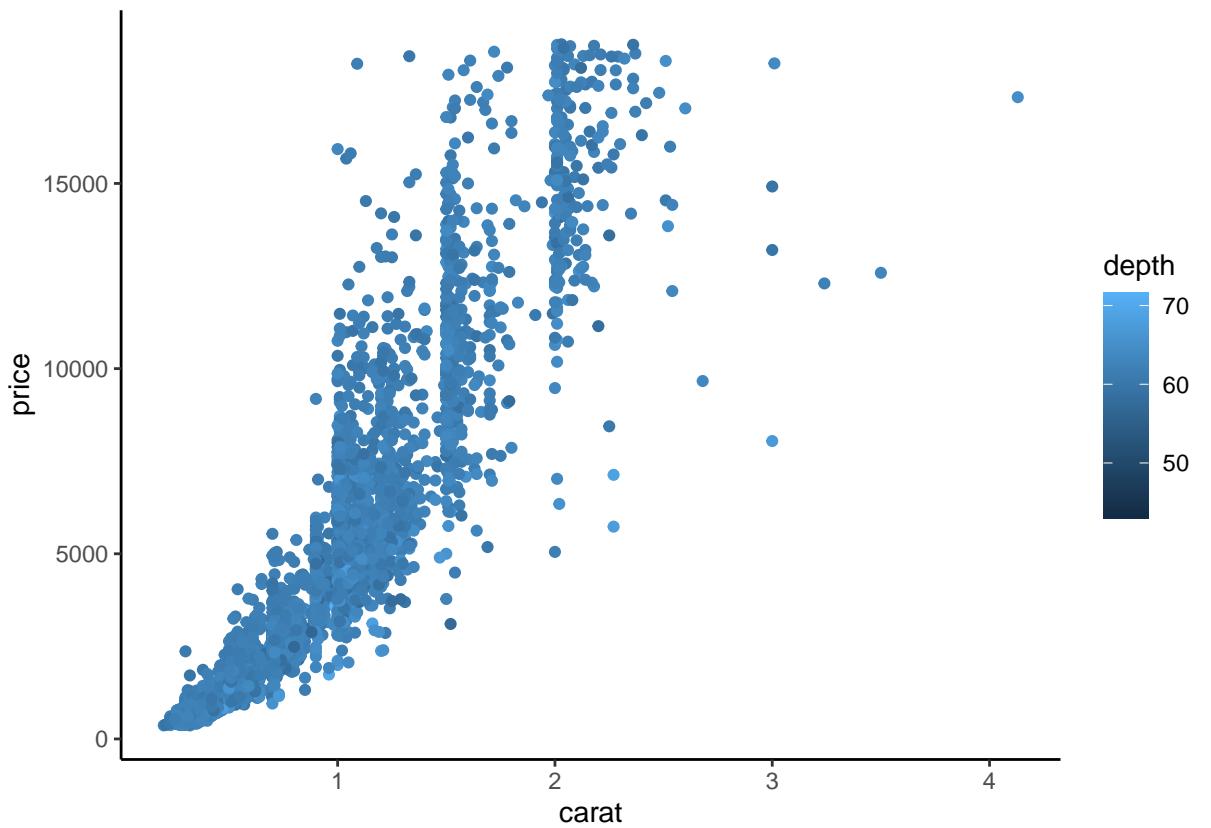
On change la couleur en utilisant la palette **Purples** :

```
> p1+scale_fill_brewer(palette="Purples")
```



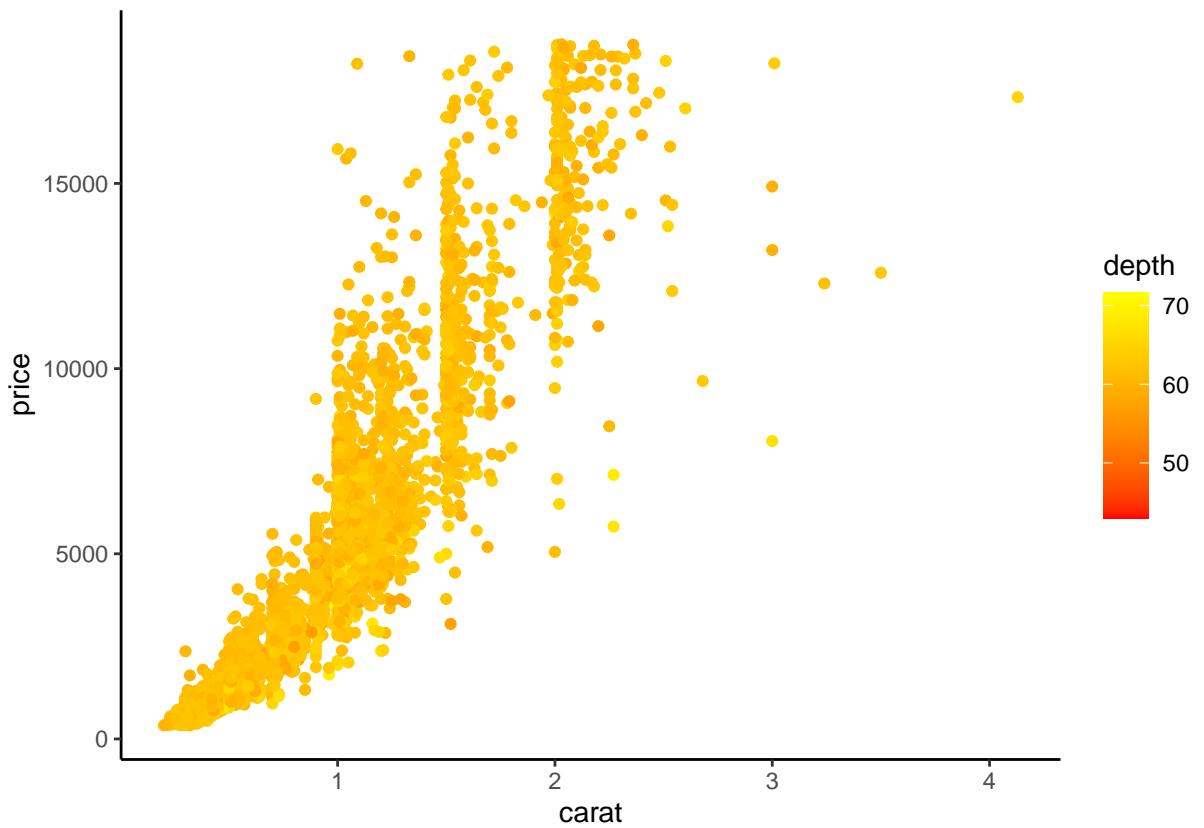
— Gradient de couleurs pour un nuage de points :

```
> p2 <- ggplot(diamonds2)+aes(x=carat,y=price)+geom_point(aes(color=depth))
> p2
```



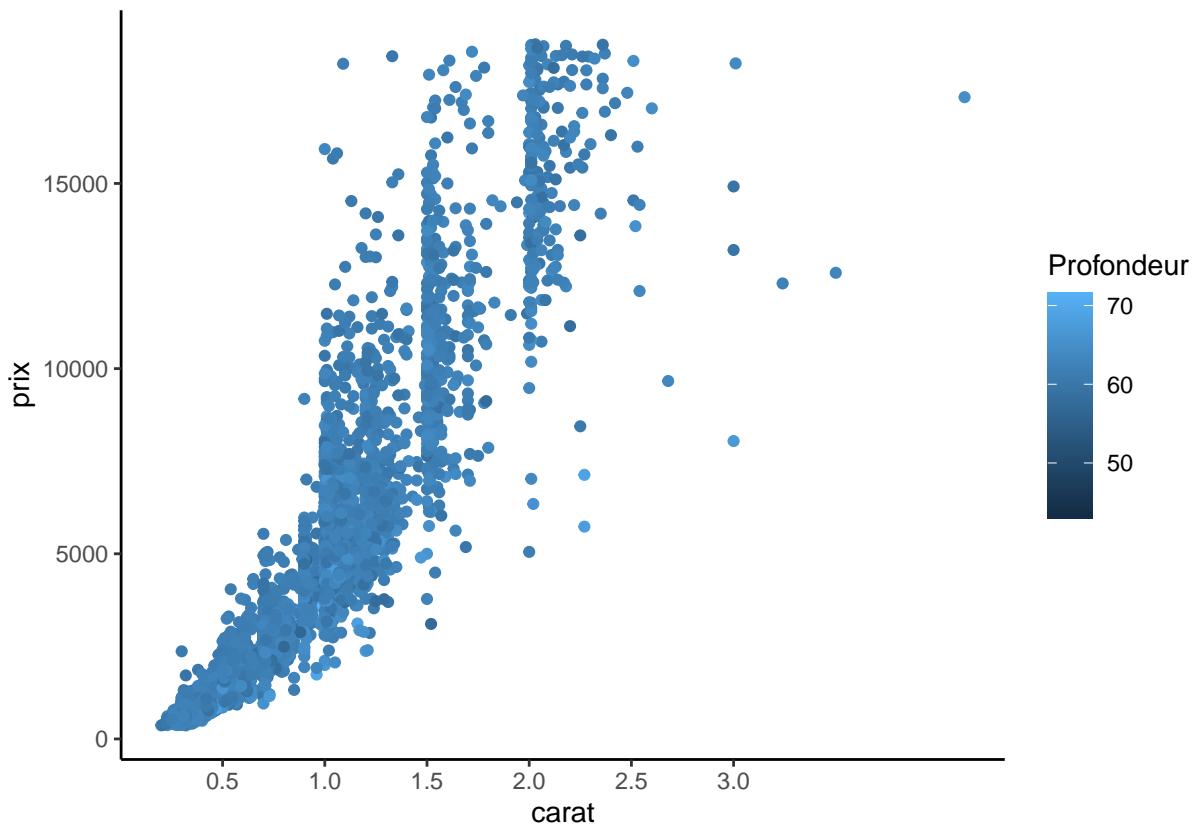
On change le gradient de couleur

```
> p2+scale_color_gradient(low="red",high="yellow")
```



— Modification sur les axes

```
> p2+scale_x_continuous(breaks=seq(0.5,3,by=0.5))+  
+   scale_y_continuous(name="prix") +  
+   scale_color_gradient("Profondeur")
```



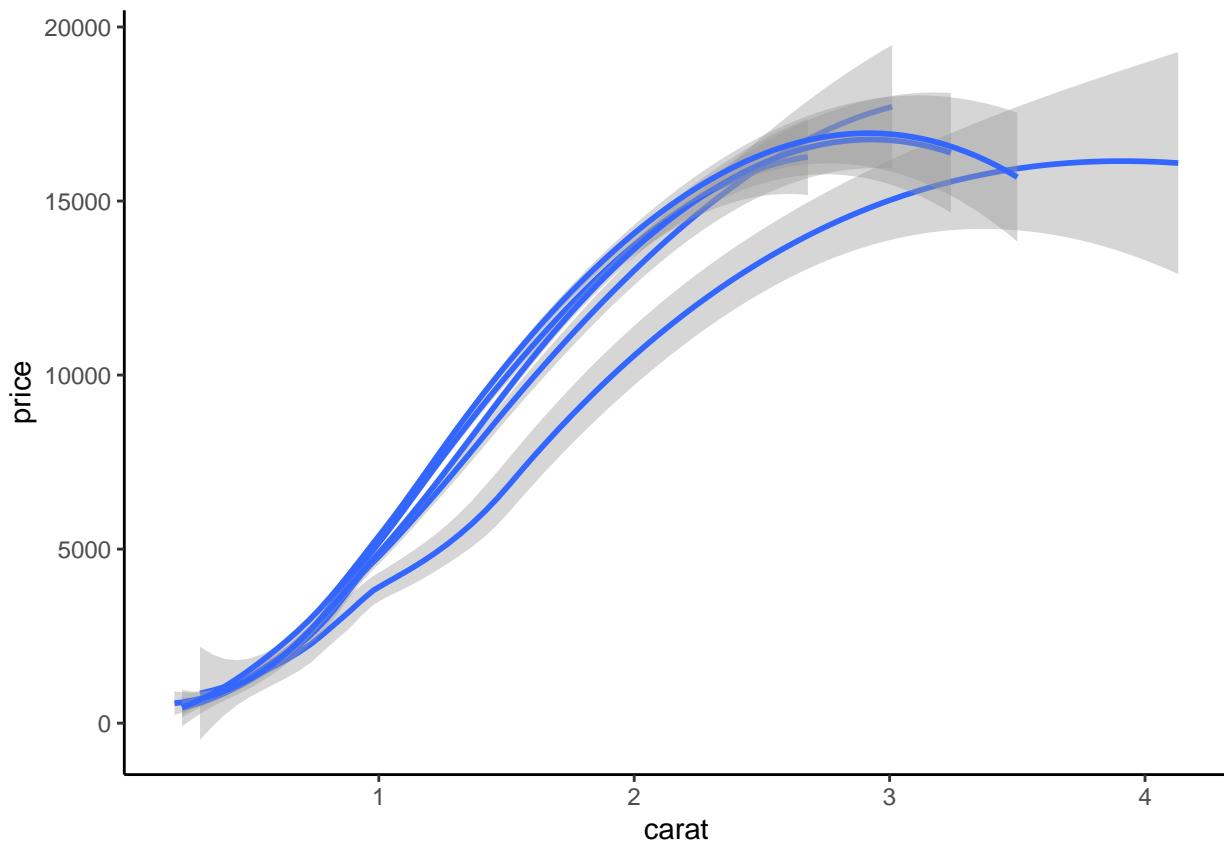
#### 4.2.6 Group et facets

`ggplot` permet de faire des représentations pour des groupes d'individus. On procède généralement de deux façons différentes :

- visualisation de sous groupes sur le même graphe, on utilise l'option `group` dans `aes` ;
- visualisation de sous groupes sur des graphes différents, on utilise le verbe `facets`.

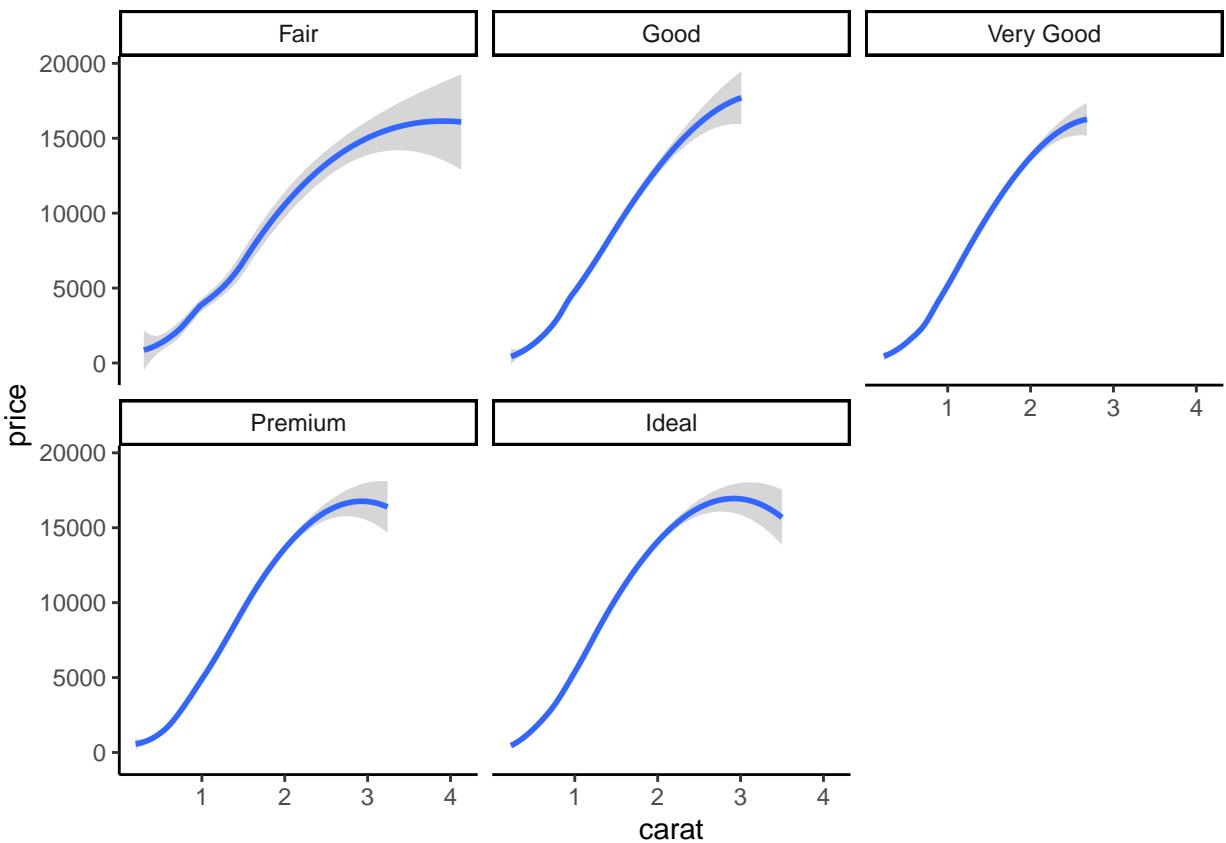
Représentons ici (sur le même graphe) le lissoir `price vs carat` pour chaque modalité de `cut`

```
> ggplot(diamonds2)+aes(x=carat,y=price,group=cut)+  
+   geom_smooth(method="loess")
```

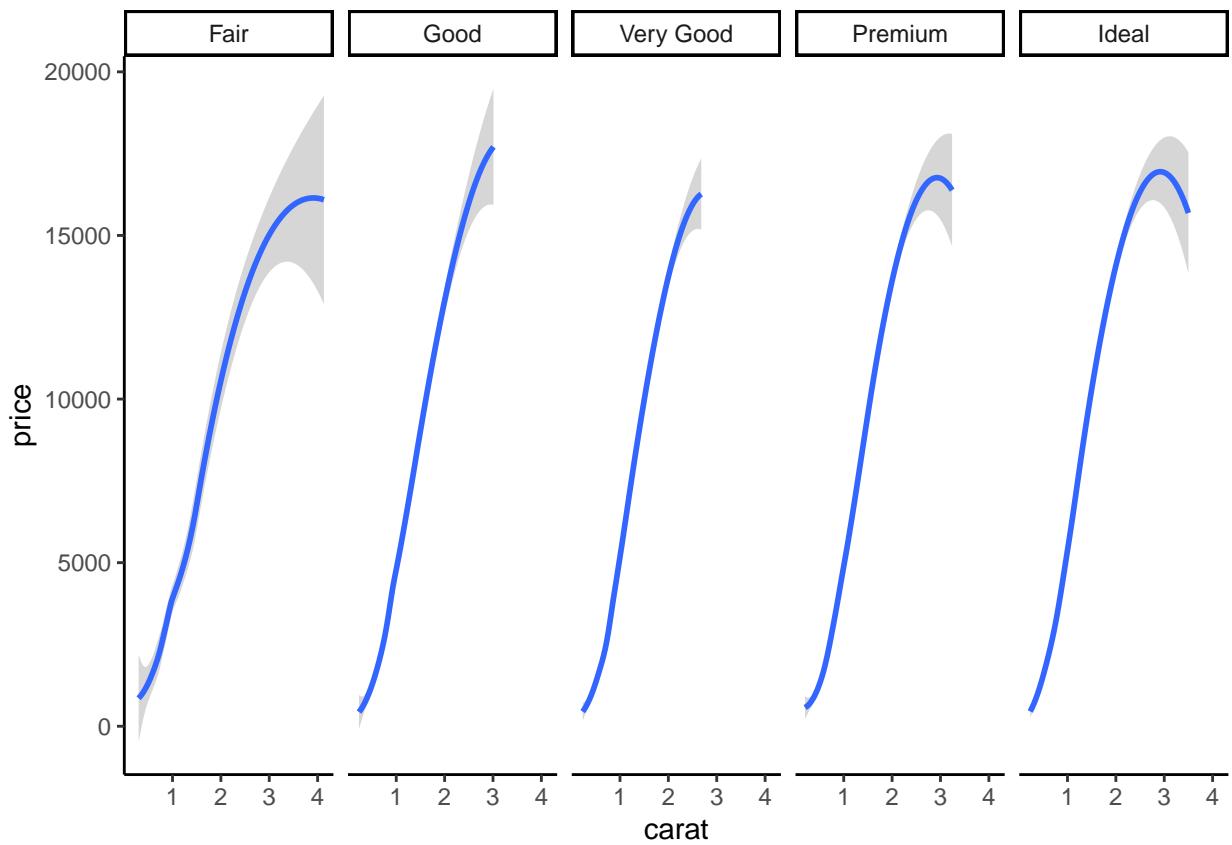


Pour obtenir cette représentation sur plusieurs fenêtres, on utilise

```
> ggplot(diamonds2)+aes(x=carat,y=price)+  
+   geom_smooth(method="loess")+facet_wrap(~cut)
```

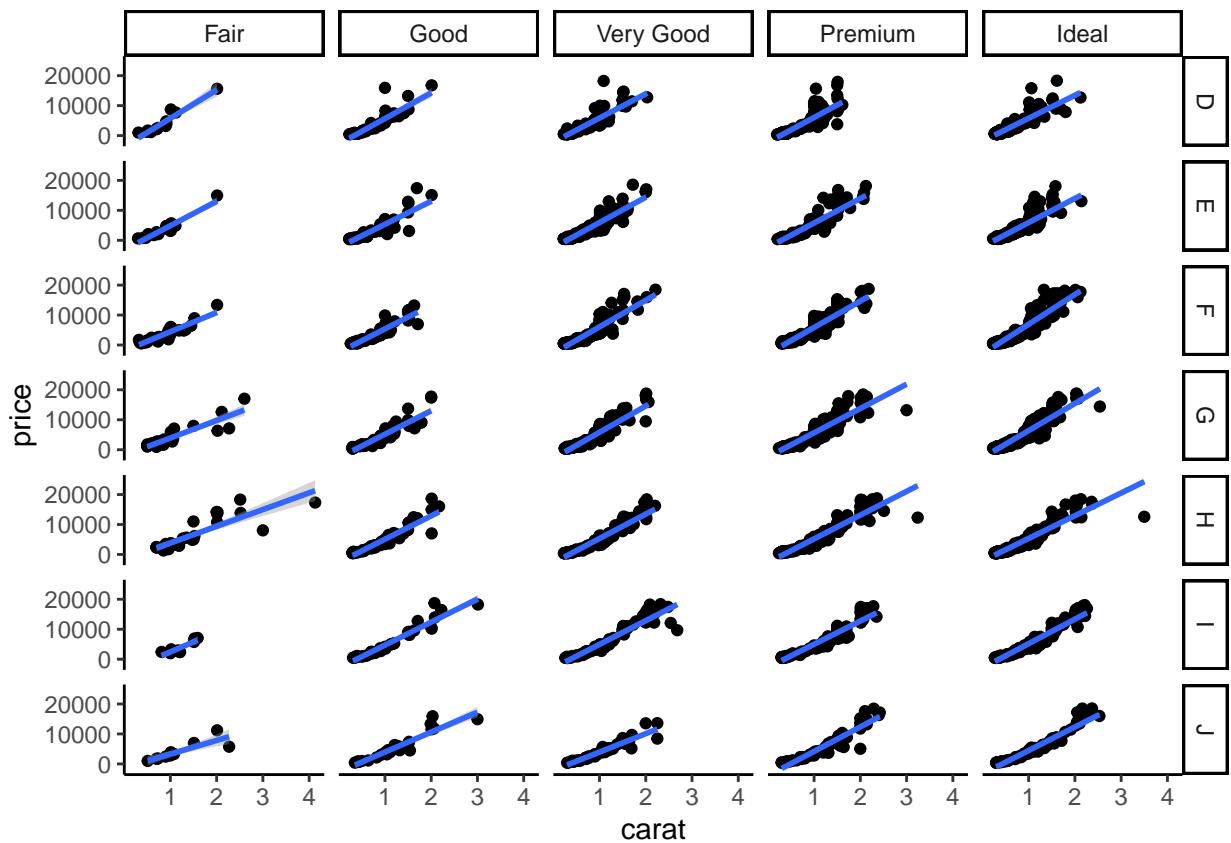


```
> ggplot(diamonds2)+aes(x=carat,y=price)+  
+   geom_smooth(method="loess")+facet_wrap(~cut,nrow=1)
```

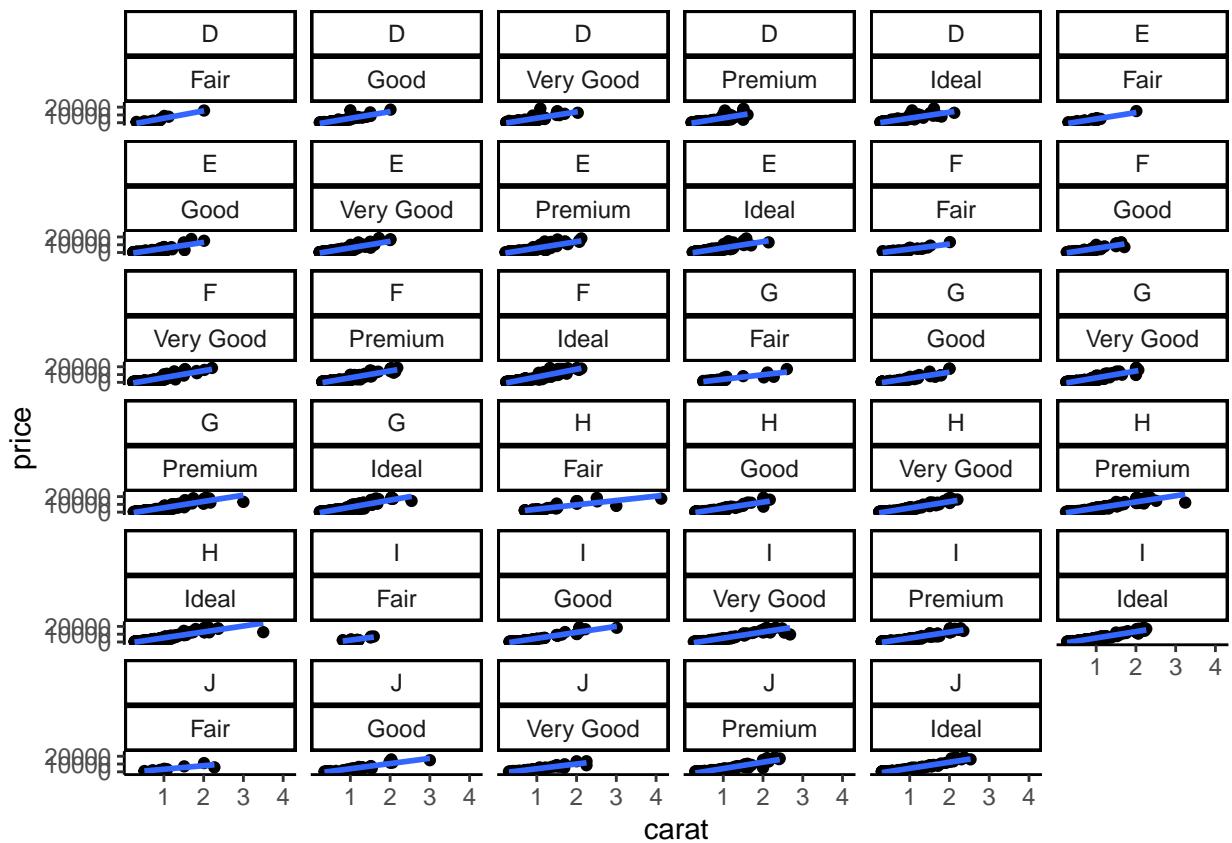


*facet\_grid* et *facet\_wrap* font des choses proches mais divisent la fenêtre de façon différente :

```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point()+
+   geom_smooth(method="lm")+facet_grid(color~cut)
```



```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point()+
+   geom_smooth(method="lm")+facet_wrap(color~cut)
```



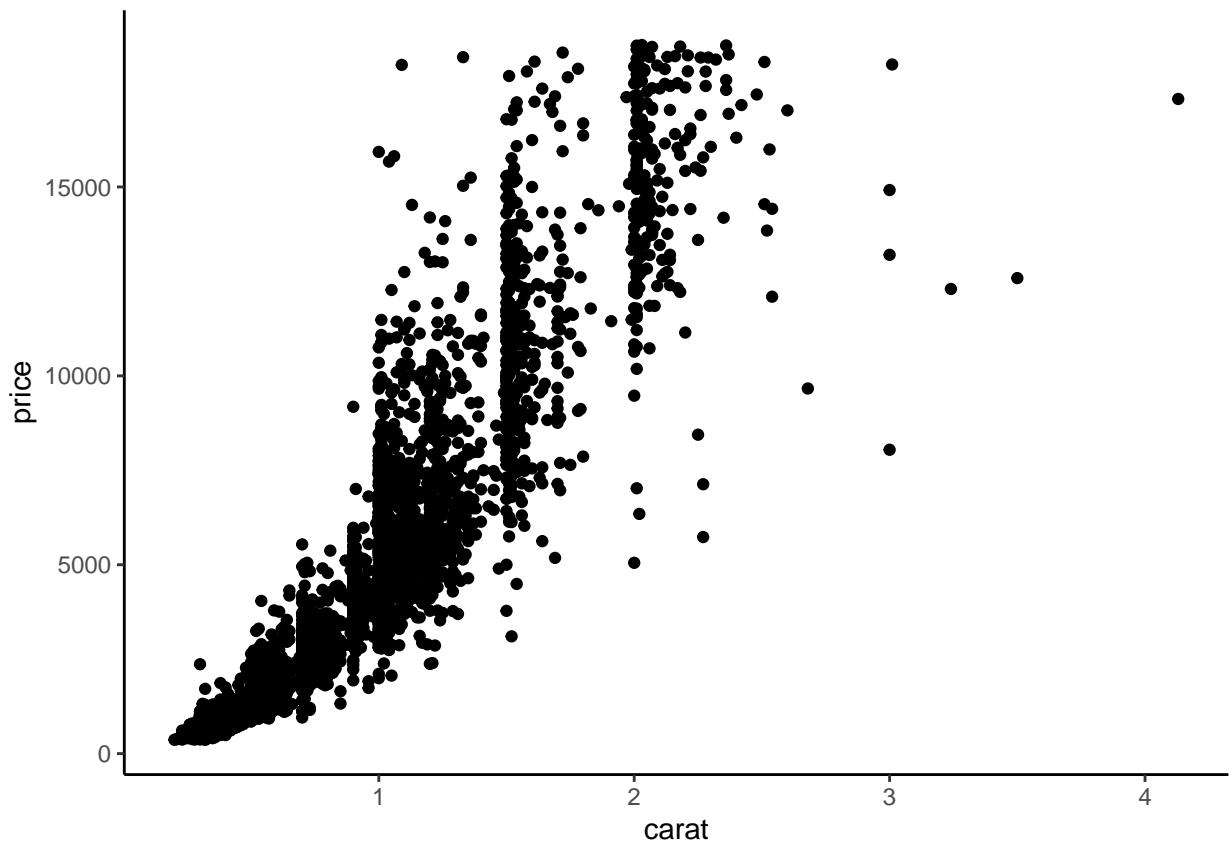
### 4.3 Compléments

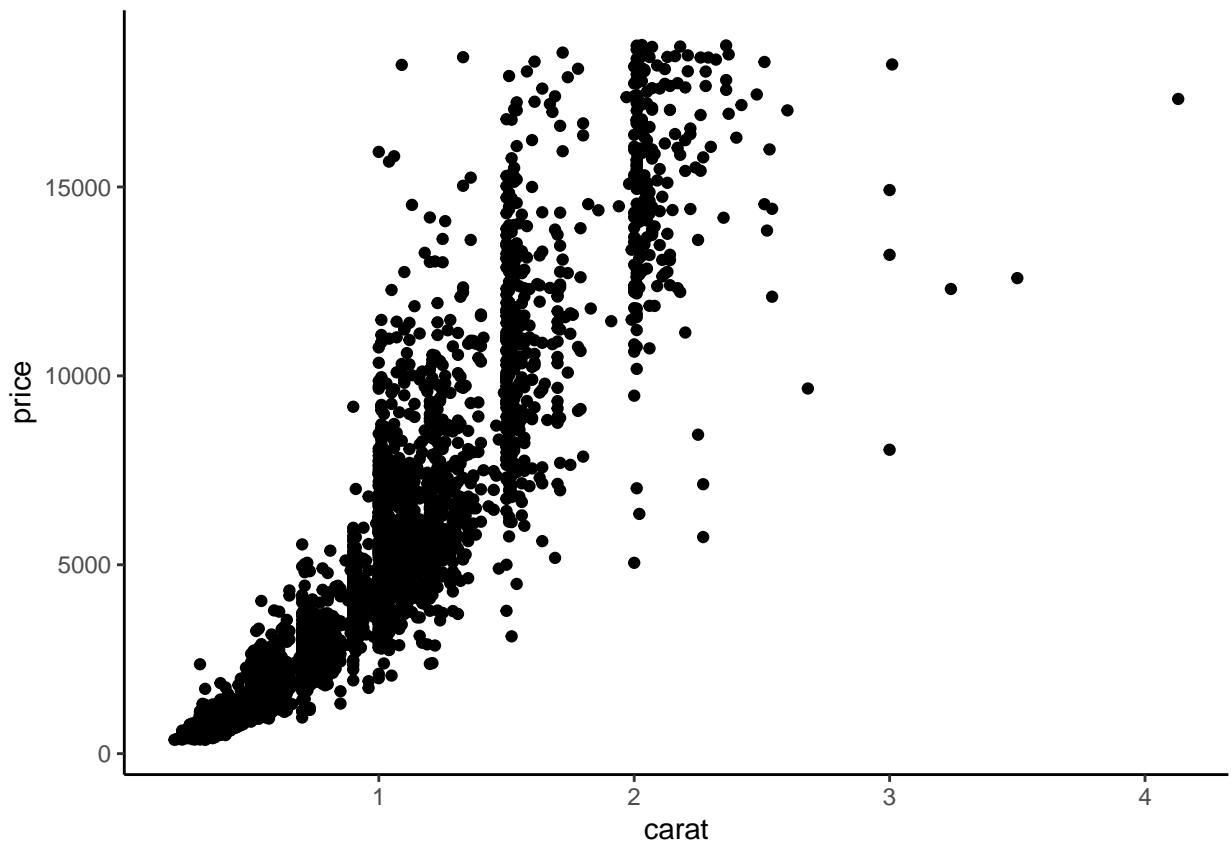
La syntaxe **ggplot** est définie selon le schéma :

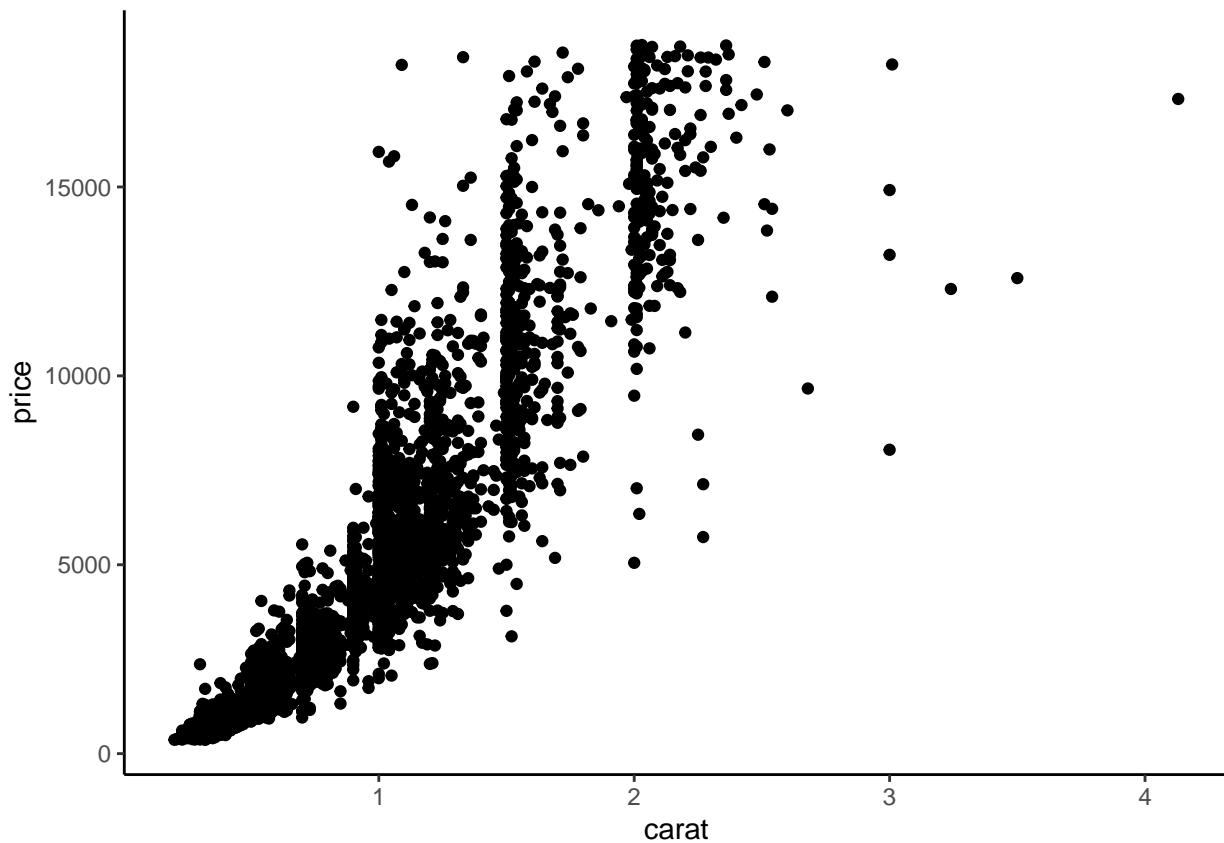
```
> ggplot() + aes() + geom_() + scale_()
```

Elle est très flexible, on peut par exemple spécifier les variables de **aes** dans les verbes **ggplot** ou **geom\_** :

```
> ggplot(diamonds2) + aes(x=carat, y=price) + geom_point()
```



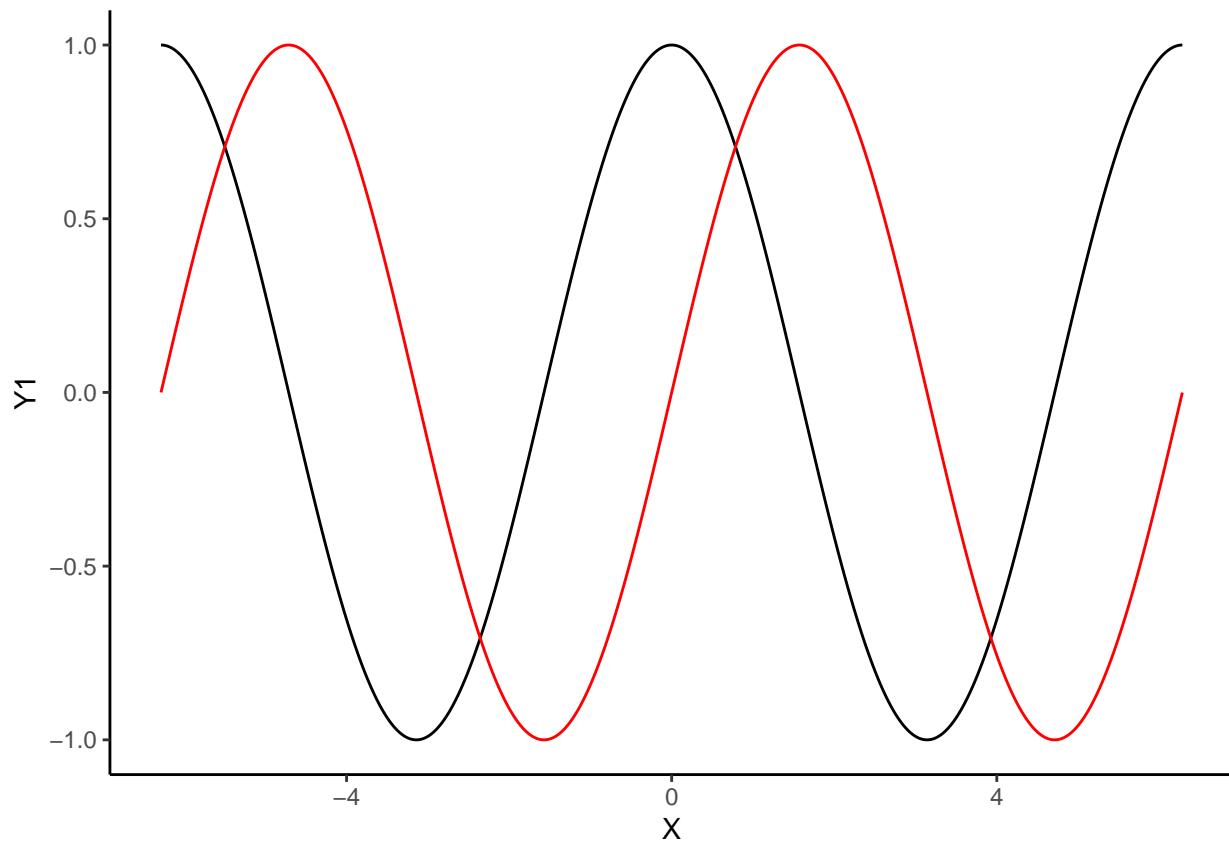




Ceci peut se révéler très utile lorsqu'on utilise des **aes** différents dans les **geom\_**.

On peut aussi construire un graphe à l'aide de différents jeux de données :

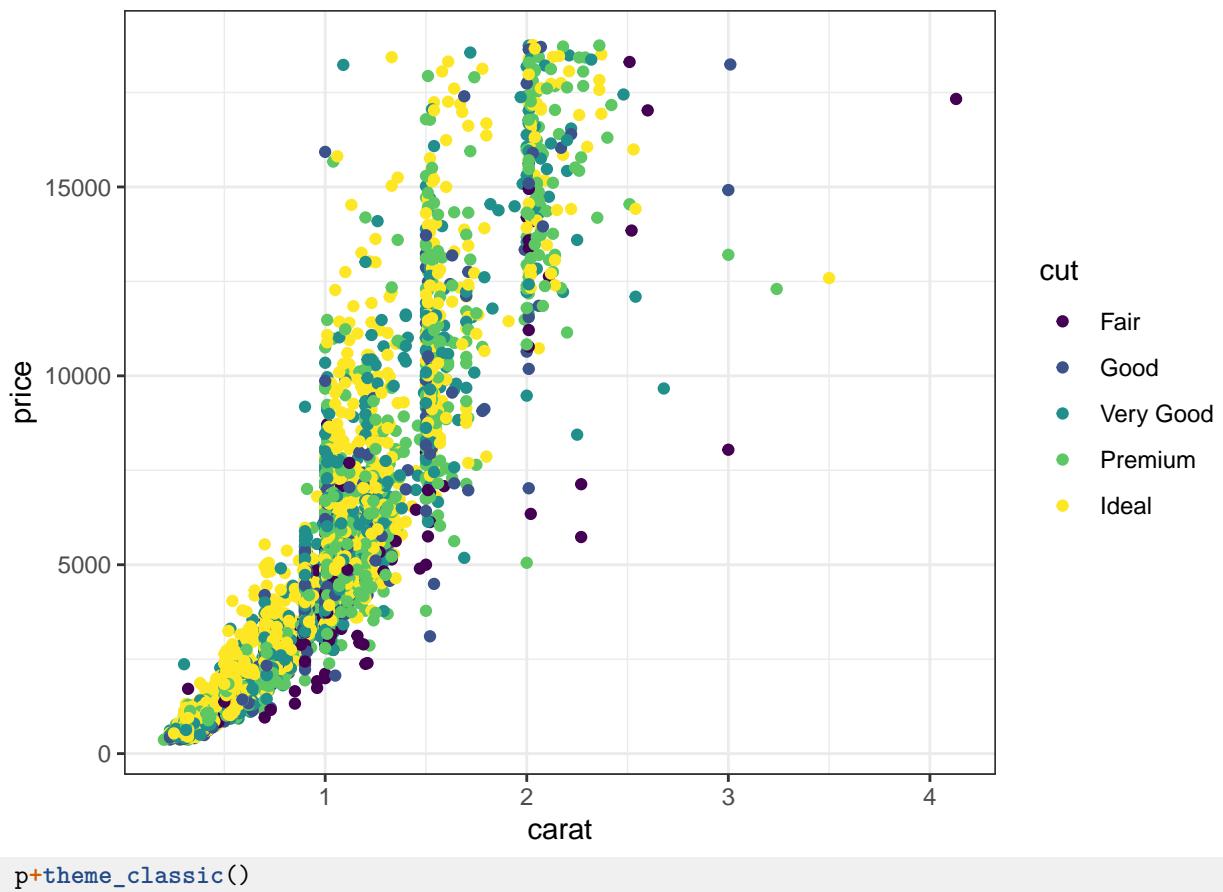
```
> X <- seq(-2*pi,2*pi,by=0.001)
> Y1 <- cos(X)
> Y2 <- sin(X)
> donnees1 <- data.frame(X,Y1)
> donnees2 <- data.frame(X,Y2)
> ggplot(donnees1)+geom_line(aes(x=X,y=Y1))+ 
+   geom_line(data=donnees2,aes(x=X,y=Y2),color="red")
```

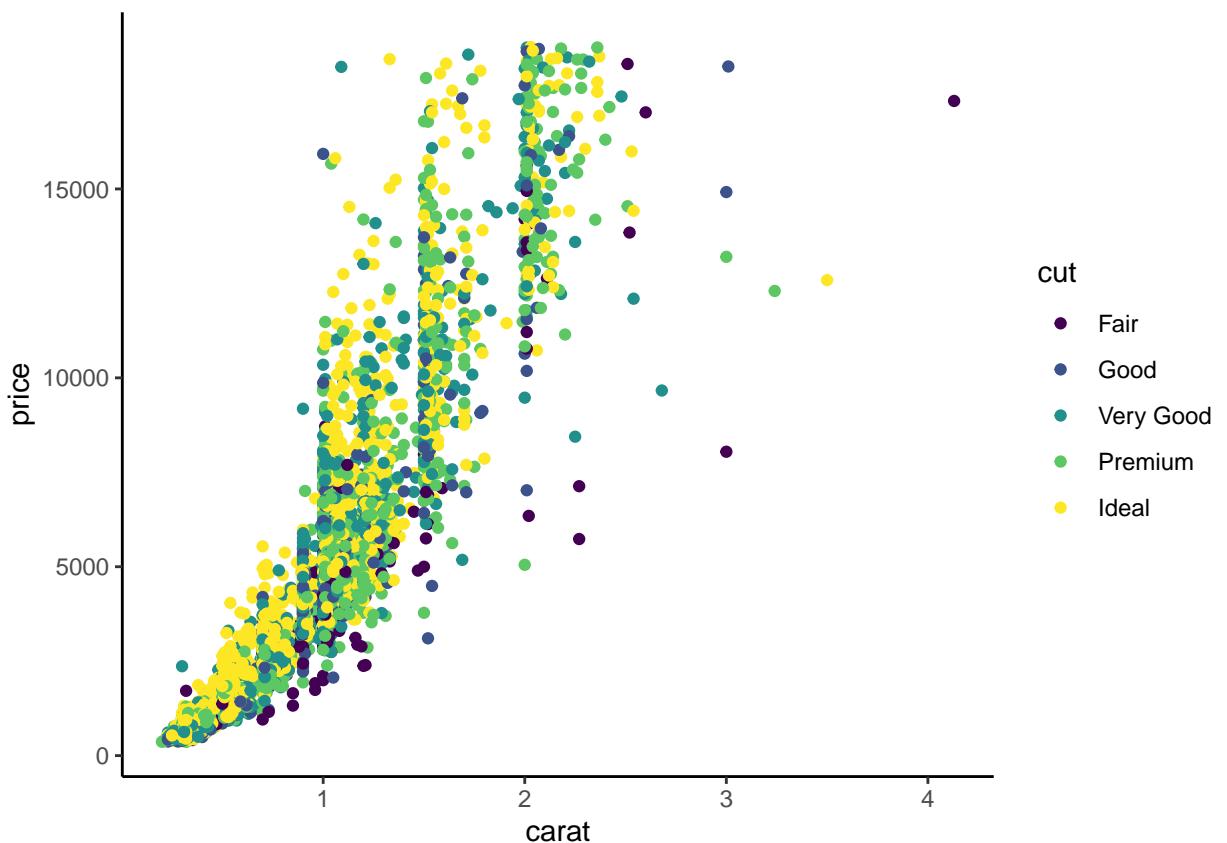


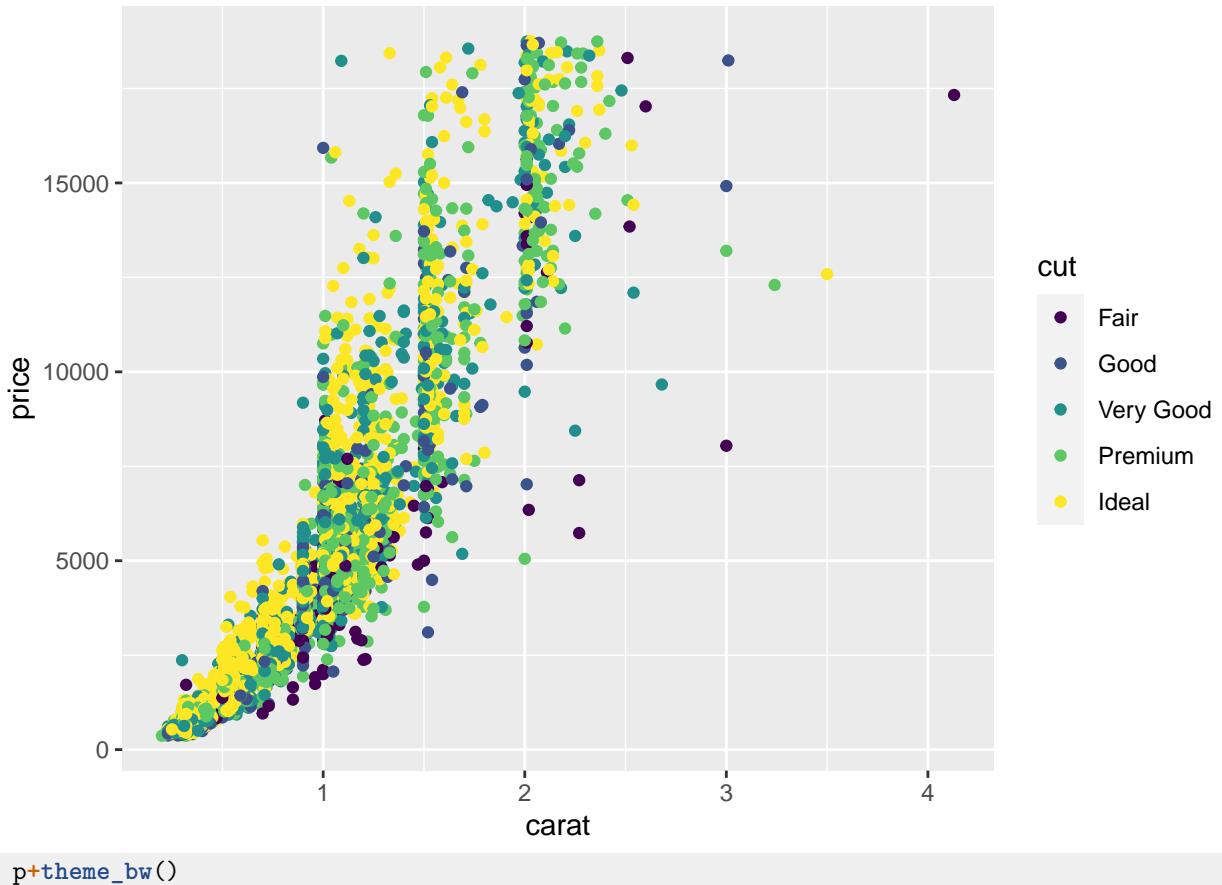
Il existe d'autres fonctions **ggplot** :

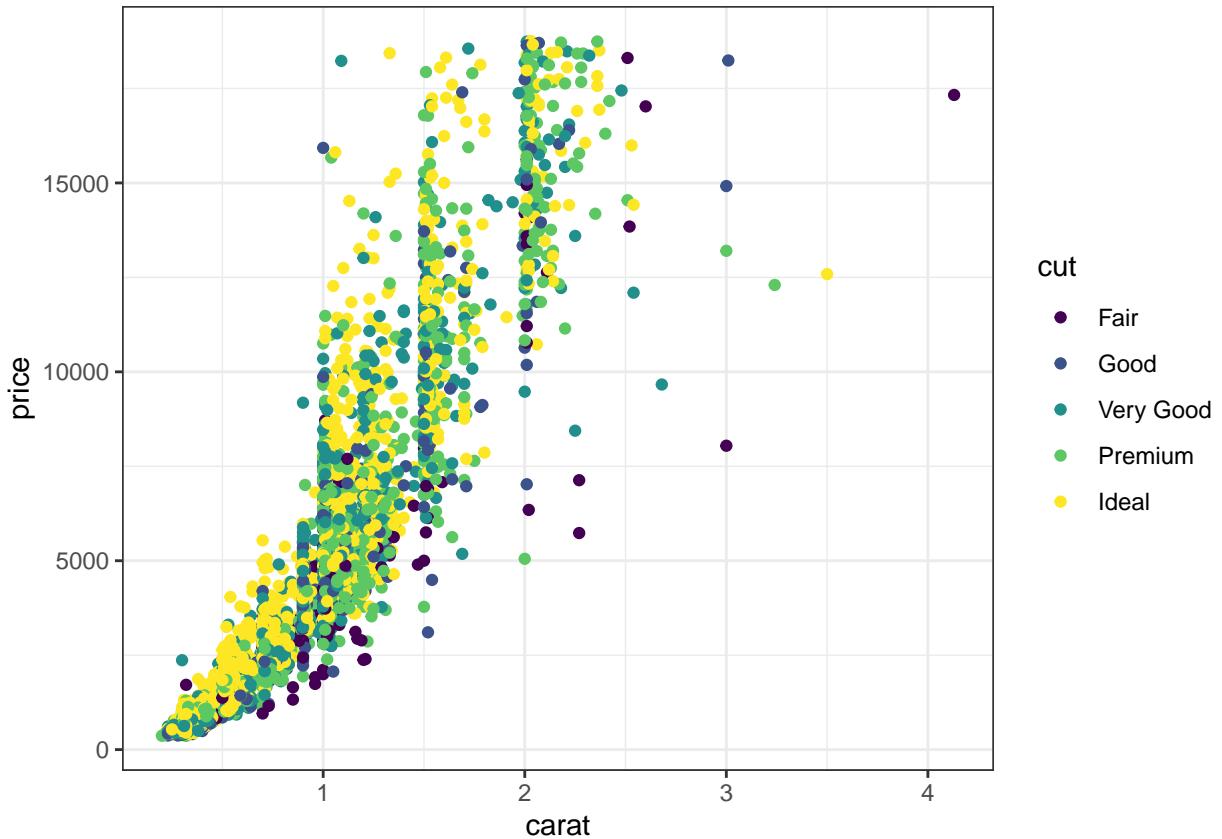
- **ggtitle** pour ajouter un titre.
- **ggsave** pour sauver un graphe.
- **theme\_** pour changer le theme du graphe.

```
> p <- ggplot(diamonds2)+aes(x=carat,y=price,color=cut)+geom_point()  
> p+theme_bw()
```









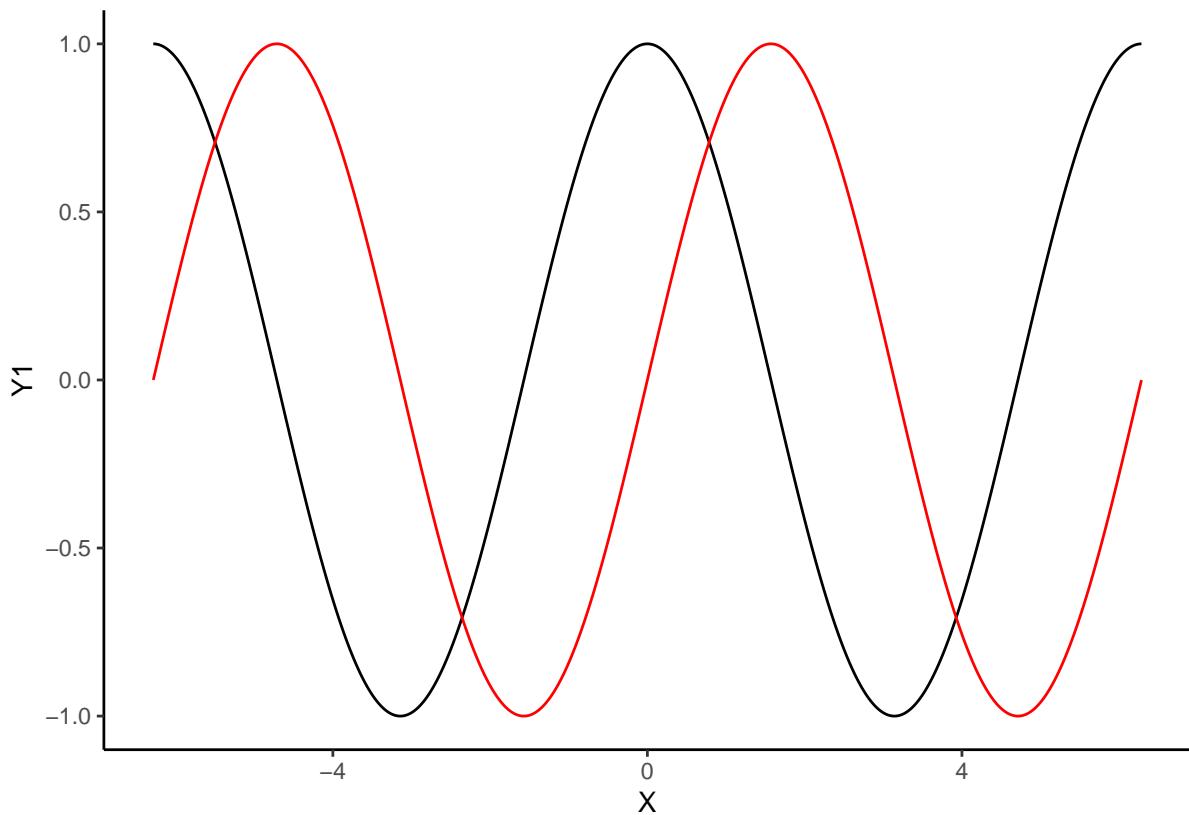
D'autres thèmes sont disponibles dans le package `ggtheme`. On pourra également parler de la fonction `set_theme` qui permet de préciser modifier le thème par défaut pour un document **Markdown**.

#### 4.4 Quelques exercices supplémentaires

**Exercice 4.9** (Fonctions cosinus et sinus).

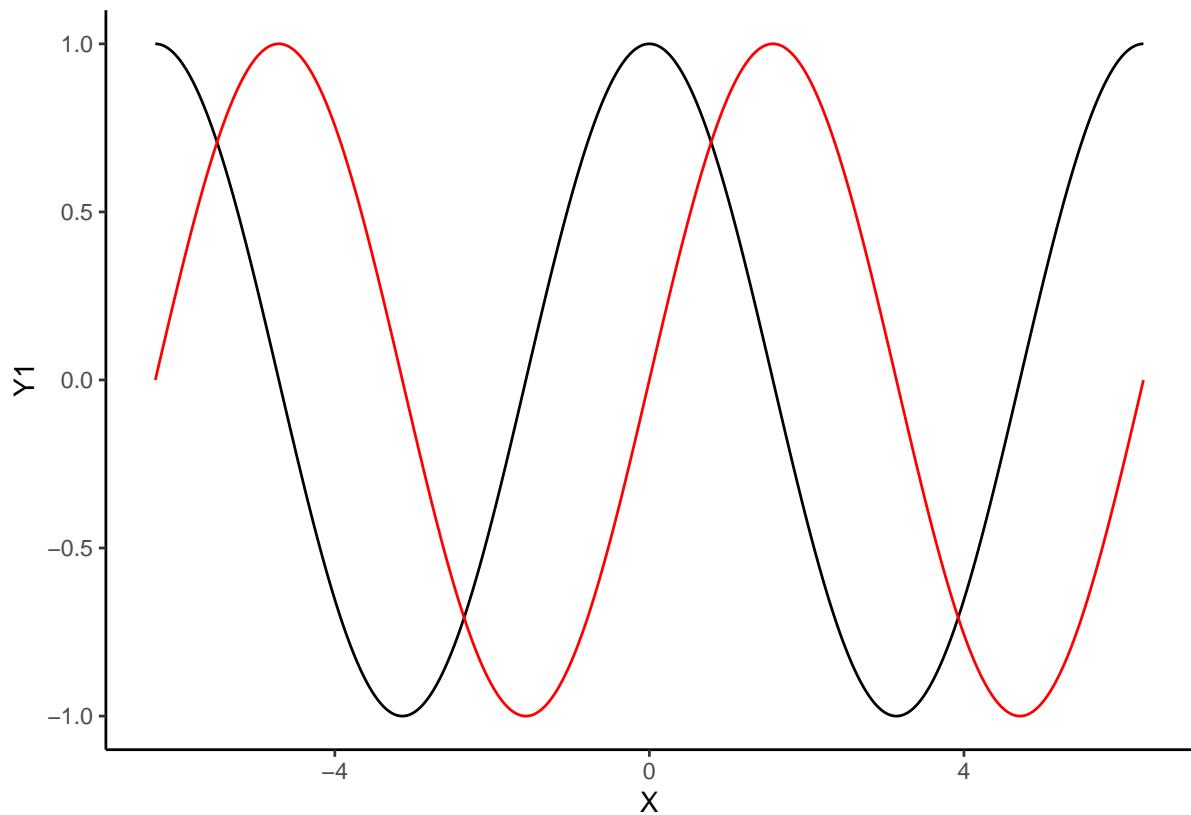
- Tracer les fonctions sinus et cosinus. On utilisera tout d'abord deux jeux de données : un pour le sinus, l'autre pour le cosinus.

```
> X <- seq(-2*pi, 2*pi, by=0.001)
> Y1 <- cos(X)
> Y2 <- sin(X)
> donnees1 <- data.frame(X,Y1)
> donnees2 <- data.frame(X,Y2)
> ggplot(donnees1)+geom_line(aes(x=X,y=Y1))+ 
+   geom_line(data=donnees2,aes(x=X,y=Y2),color="red")
```

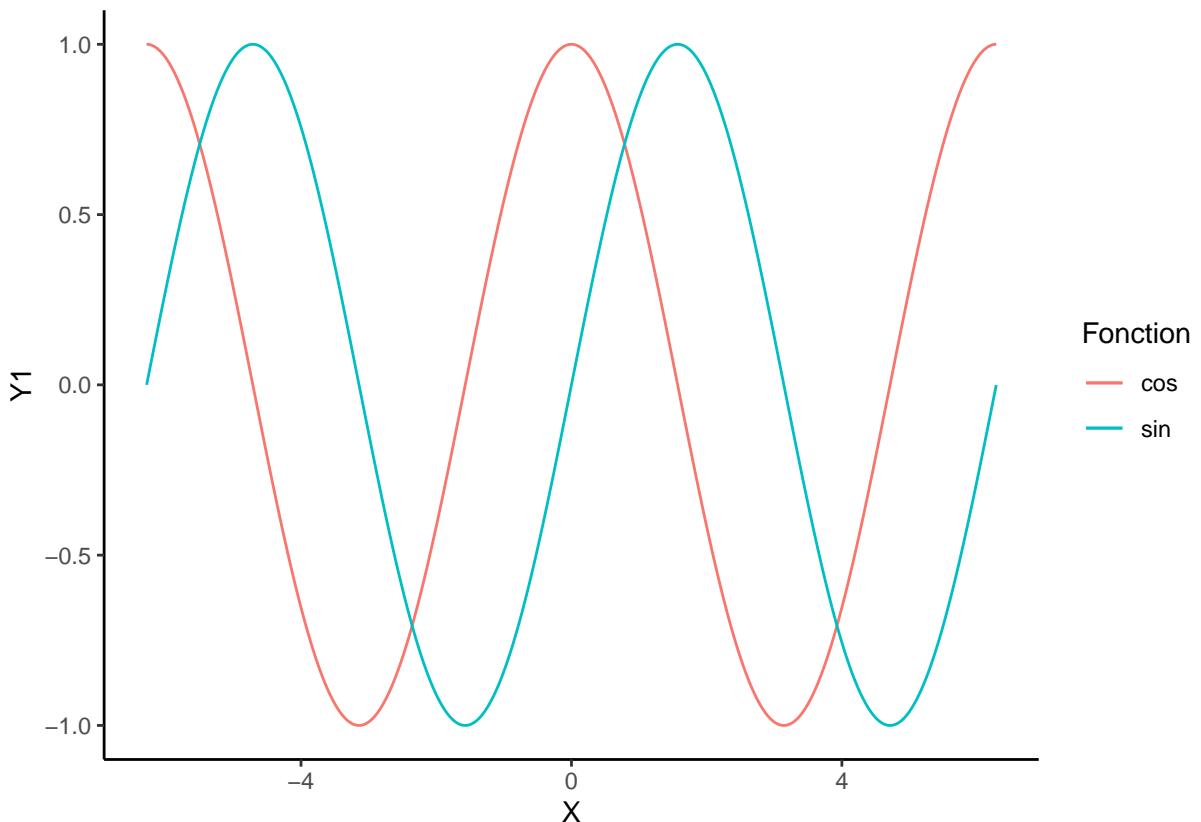


2. Faire la même chose avec un jeu de données et deux appels à la fonction `geom_line`. On pourra ajouter une légende.

```
> donnees <- data.frame(X,Y1,Y2)
> ggplot(donnees)+aes(x=X,y=Y1)+geom_line()+
+   geom_line(aes(y=Y2),color="red")
```



```
> #ou pour la légende  
> ggplot(donnees)+aes(x=X,y=Y1)+geom_line(aes(color="cos"))+  
+   geom_line(aes(y=Y2,color="sin"))+labs(color="Fonction")
```

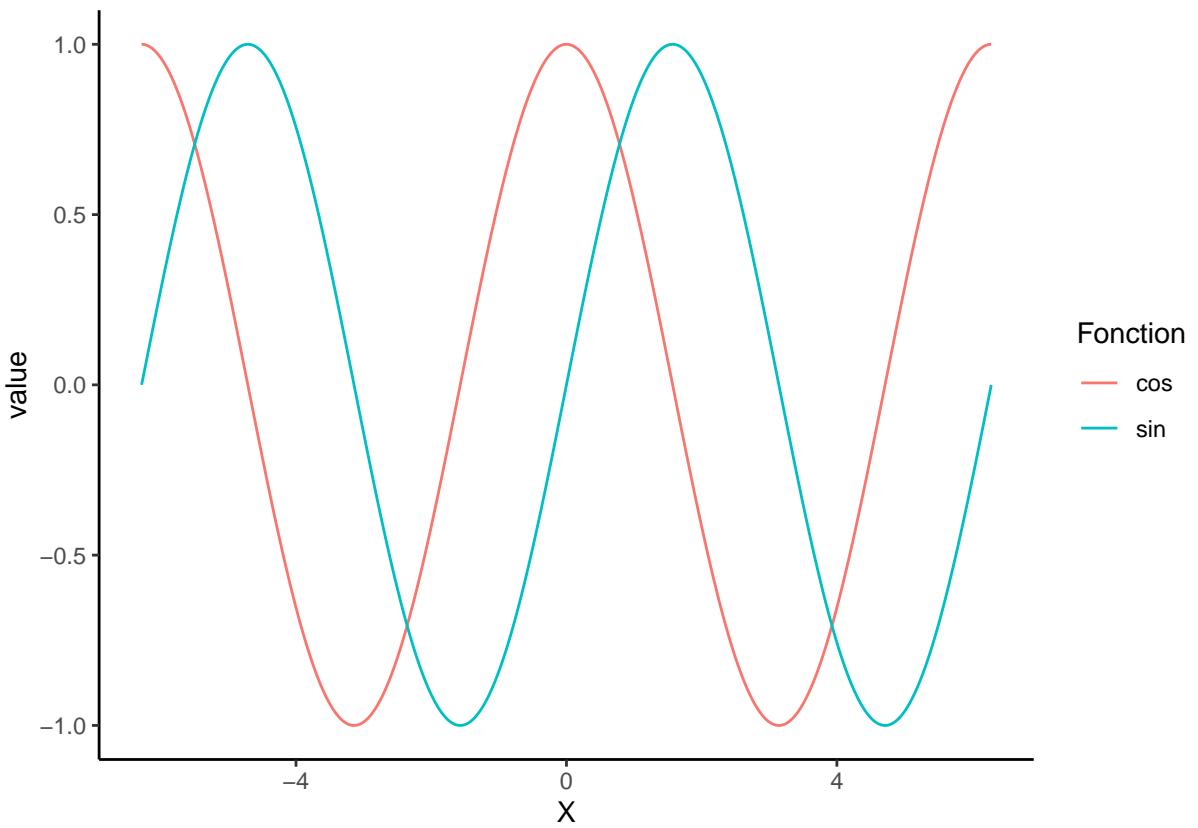


3. Faire la même chose avec un jeu de données et un seul appel à `geom_line`. On pourra utiliser la fonction `pivot_longer` du `tidyverse`.

```

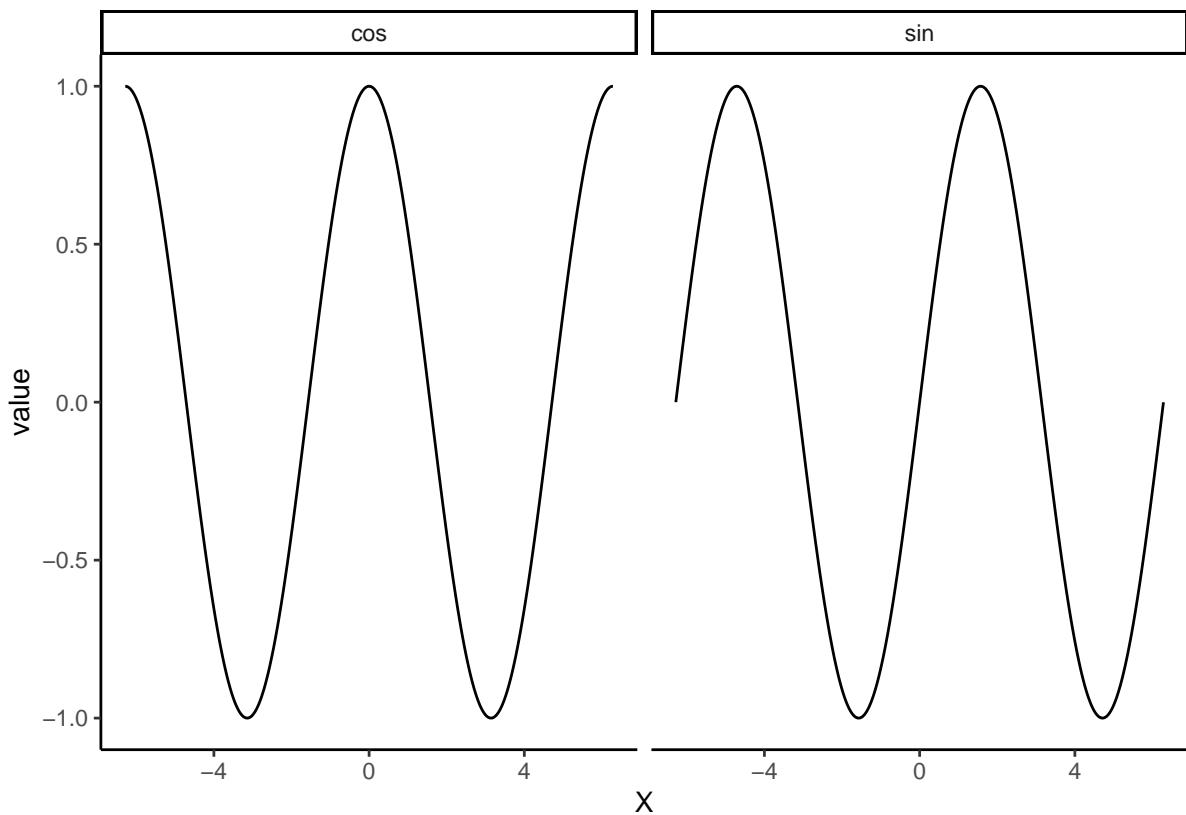
> df <- data.frame(X,cos=Y1,sin=Y2)
> df1 <- df %>% pivot_longer(cols=c(cos,sin),
+                               names_to = "Fonction",
+                               values_to = "value")
> #ou
> df1 <- df %>% pivot_longer(cols=-X,
+                               names_to = "Fonction",
+                               values_to = "value")
> ggplot(df1)+aes(x=X,y=value,color=Fonction)+geom_line()

```



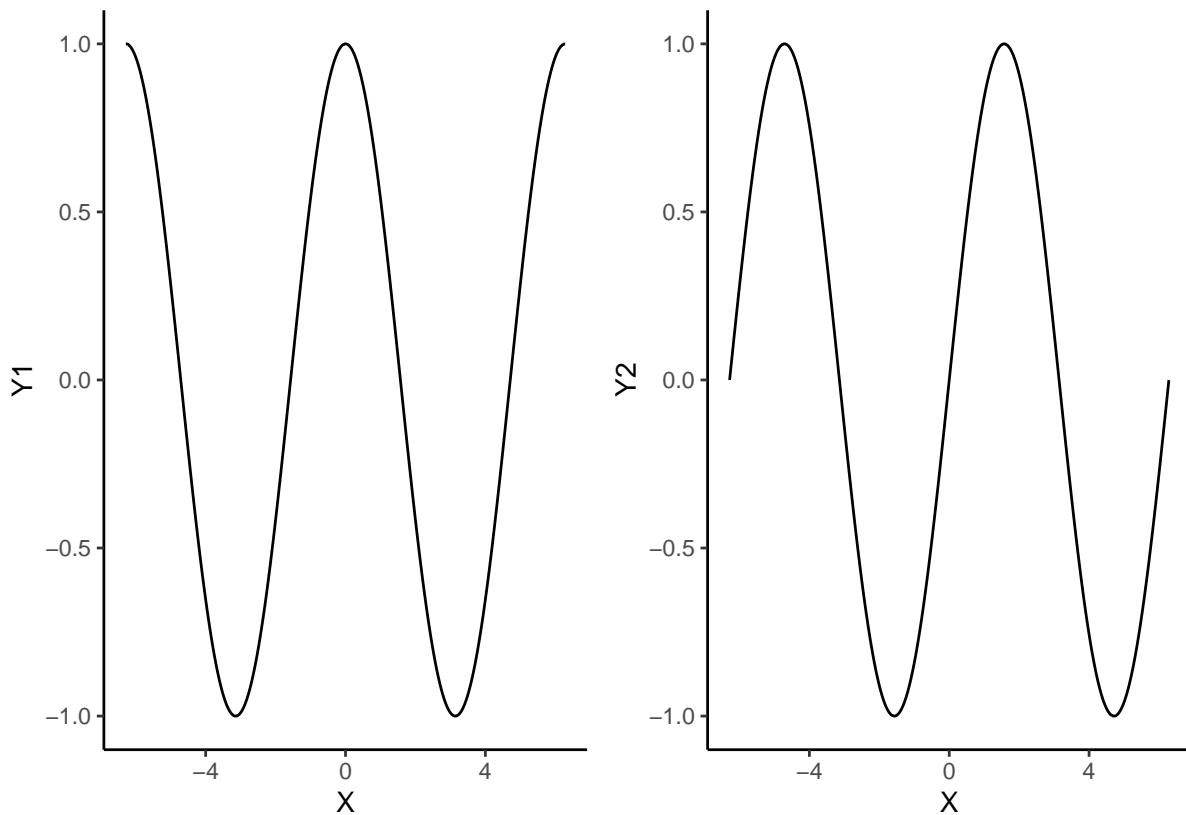
4. Tracer les deux fonctions sur deux fenêtres graphiques (utiliser `facet_wrap`).

```
> ggplot(df1)+aes(x=X,y=value)+geom_line()+facet_wrap(~Fonction)
```



5. Faire la même chose avec la fonction `grid.arrange` du package `gridExtra`.

```
> library(gridExtra)
> p1 <- ggplot(donnees1)+aes(x=X,y=Y1)+geom_line()
> p2 <- ggplot(donnees2)+aes(x=X,y=Y2)+geom_line()
> grid.arrange(p1,p2,nrow=1)
```



### Exercice 4.10 (Différents graphes).

On considère les données **mtcars**

```
> data(mtcars)
> summary(mtcars)

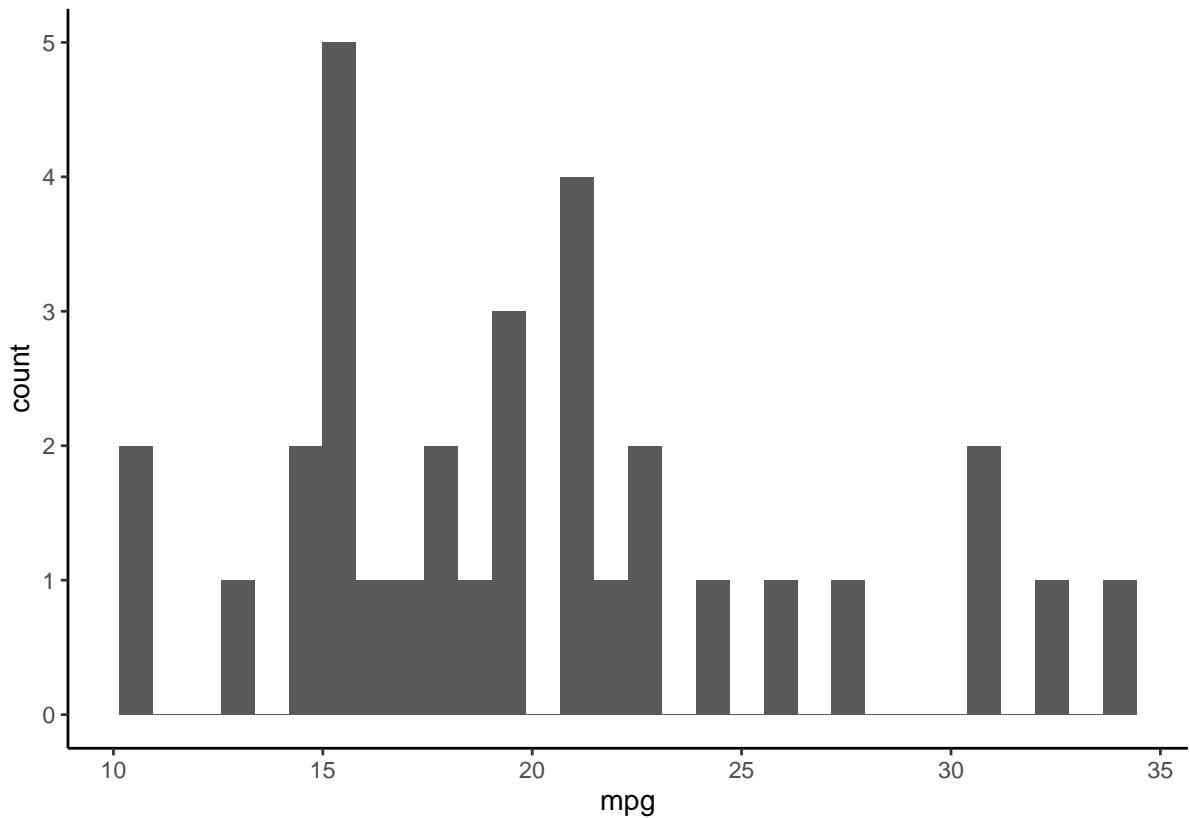
      mpg          cyl          disp          hp
Min.   :10.40   Min.   :4.000   Min.   :71.1   Min.   :52.0
1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.:96.5
Median :19.20   Median :6.000   Median :196.3   Median :123.0
Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0

      drat         wt          qsec          vs
Min.   :2.760   Min.   :1.513   Min.   :14.50  Min.   :0.0000
1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89  1st Qu.:0.0000
Median :3.695   Median :3.325   Median :17.71  Median :0.0000
Mean   :3.597   Mean   :3.217   Mean   :17.85  Mean   :0.4375
3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90  3rd Qu.:1.0000
Max.   :4.930   Max.   :5.424   Max.   :22.90  Max.   :1.0000

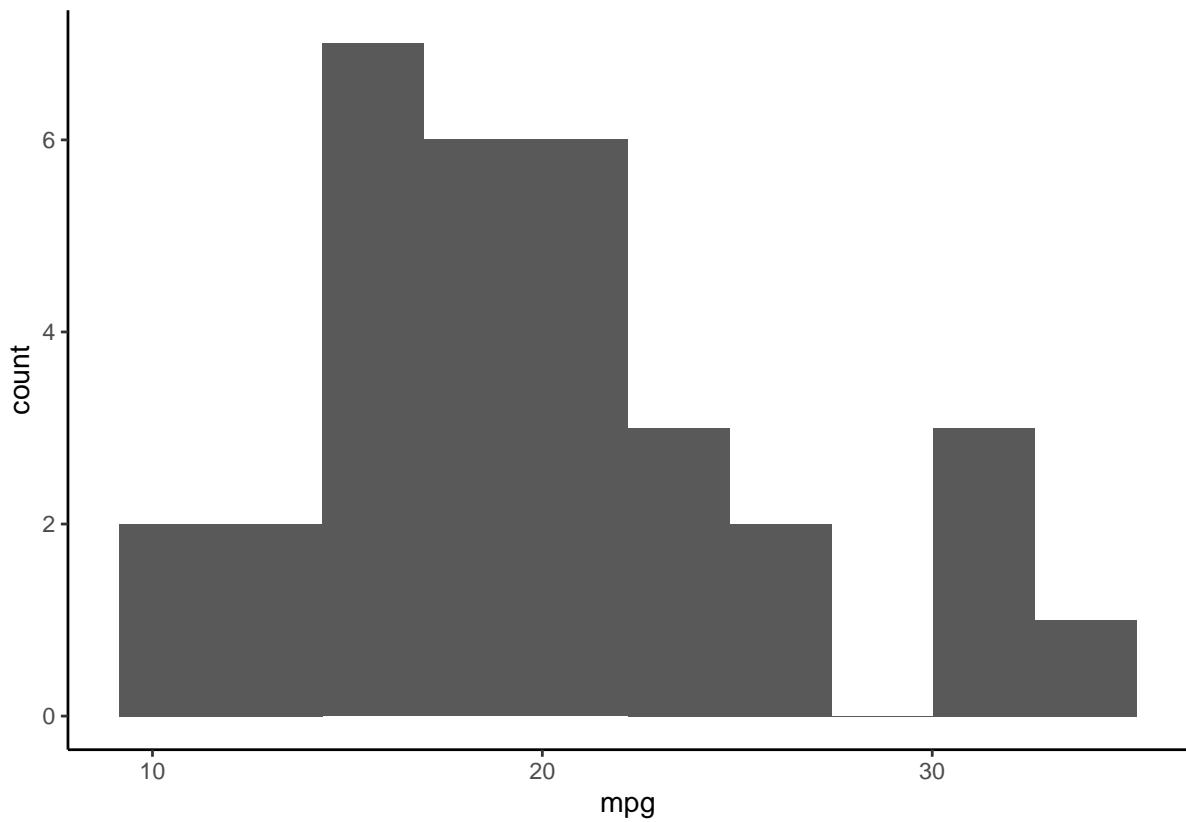
      am          gear          carb
Min.   :0.0000  Min.   :3.000  Min.   :1.000
1st Qu.:0.0000  1st Qu.:3.000  1st Qu.:2.000
Median :0.0000  Median :4.000  Median :2.000
Mean   :0.4062  Mean   :3.688  Mean   :2.812
3rd Qu.:1.0000  3rd Qu.:4.000  3rd Qu.:4.000
Max.   :1.0000  Max.   :5.000  Max.   :8.000
```

- Tracer l'histogramme de **mpg** (on fera varier le nombre de classes).

```
> ggplot(mtcars)+aes(x=mpg)+geom_histogram()
```

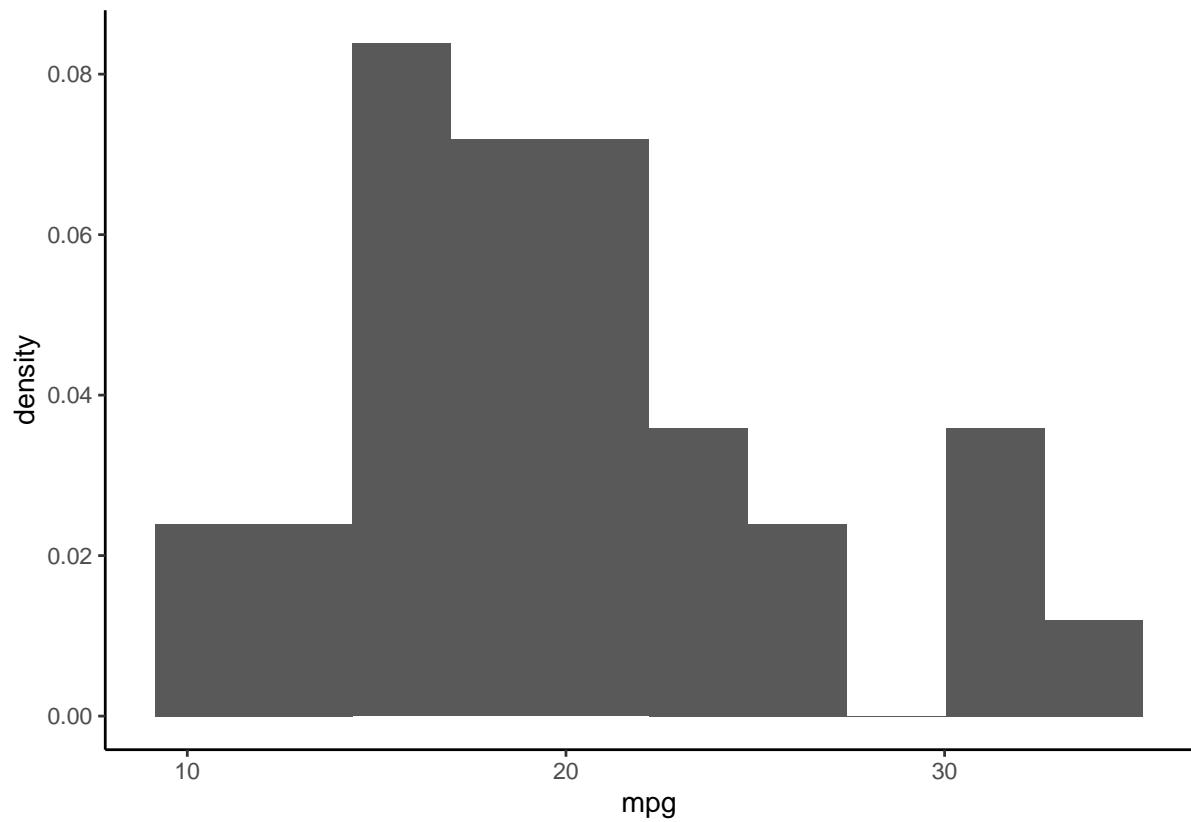


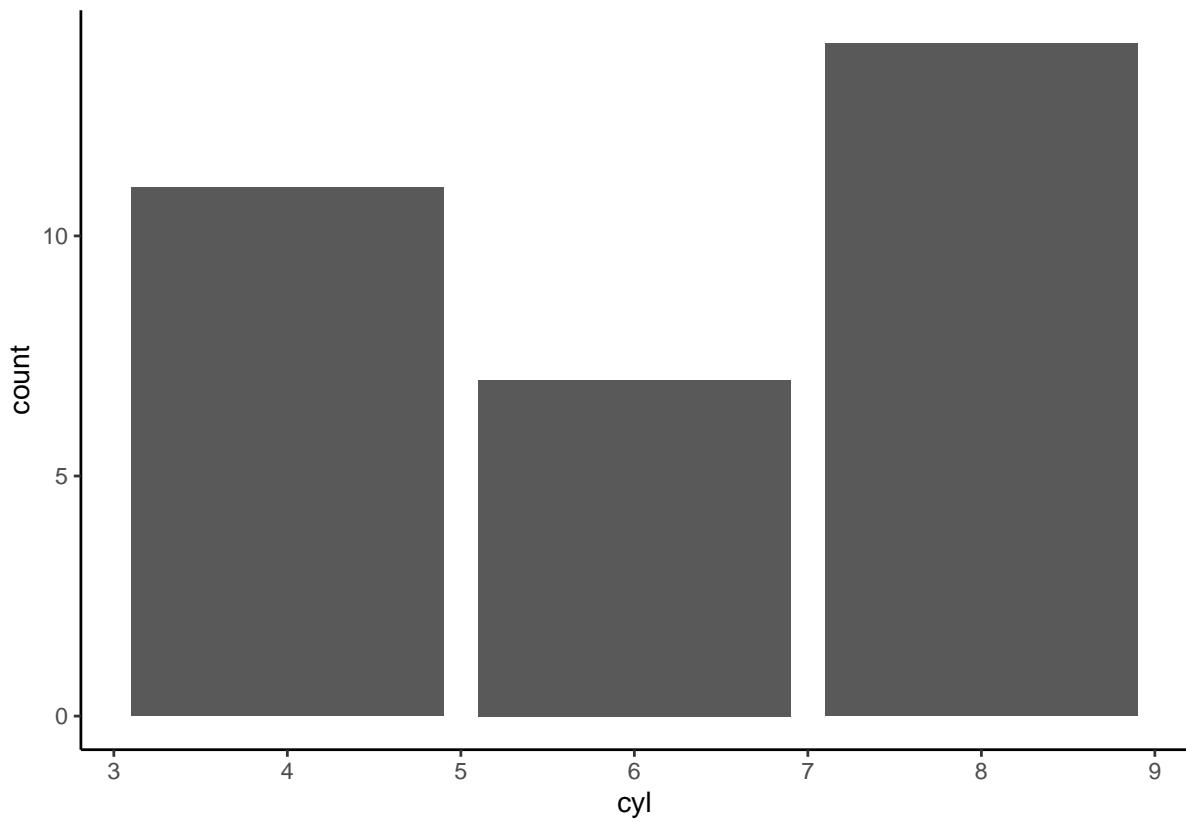
```
> ggplot(mtcars)+aes(x=mpg)+geom_histogram(bins=10)
```



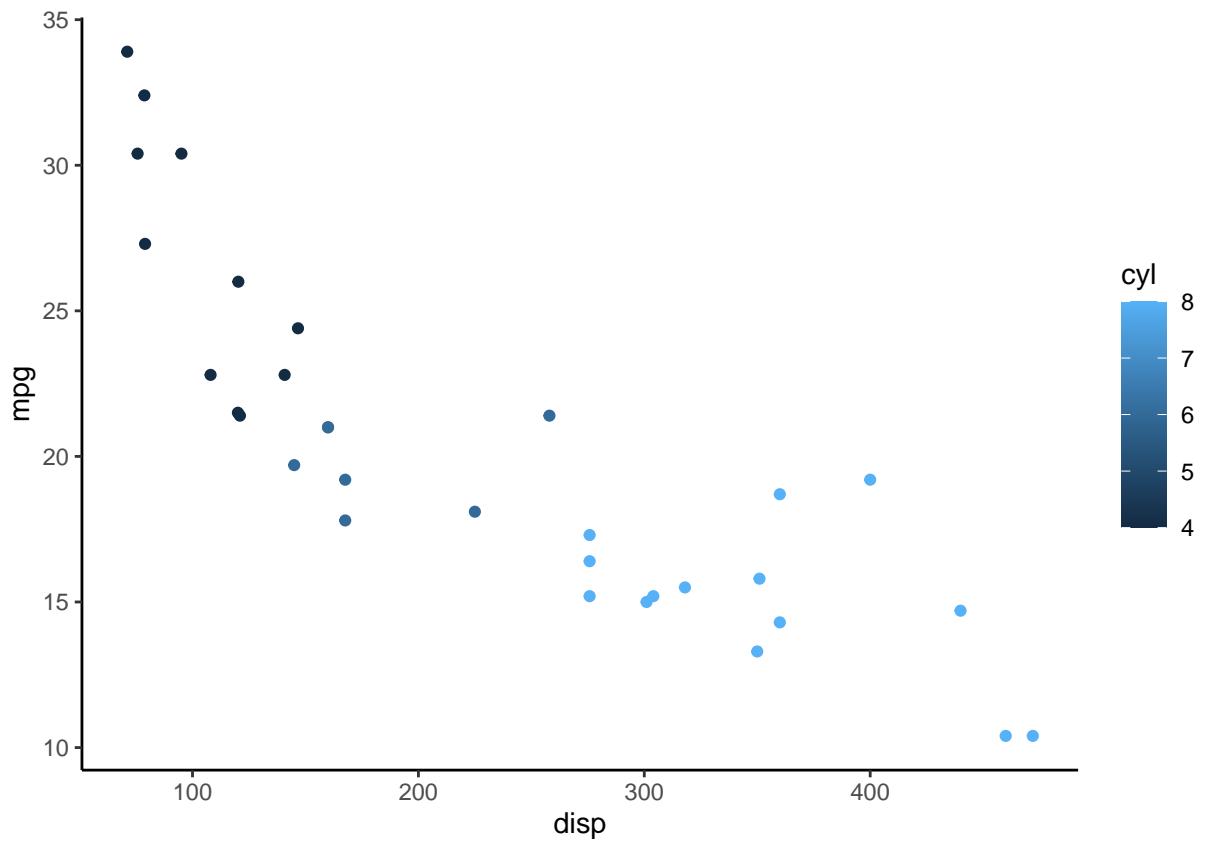
2. Tracer l'histogramme de la densité.

```
> ggplot(mtcars)+aes(x=mpg,y=..density..)+geom_histogram(bins=10)
```

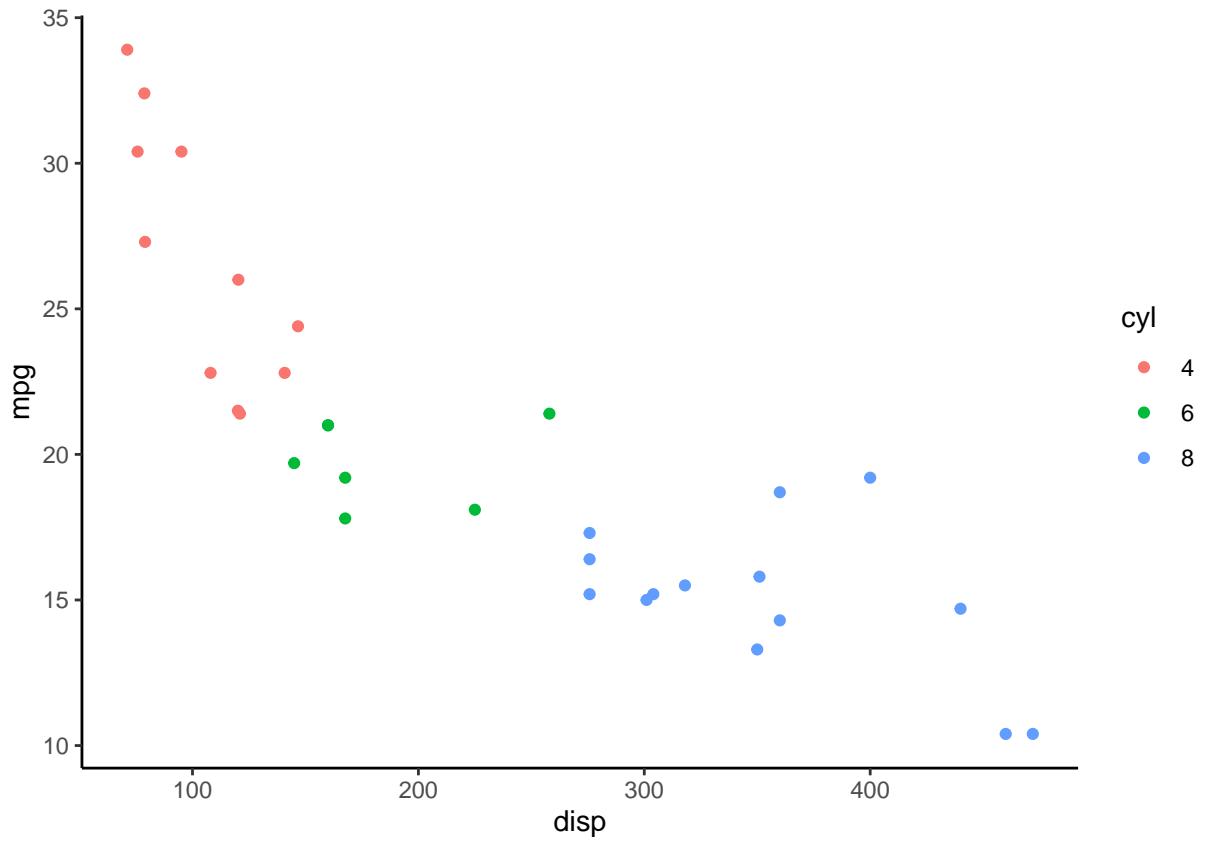




4. Tracer le nuage de points **disp** vs **mpg** en utilisant une couleur différente pour chaque valeur de **cyl**.
- ```
> ggplot(mtcars)+aes(x=disp,y=mpg,color=cyl)+geom_point()
```

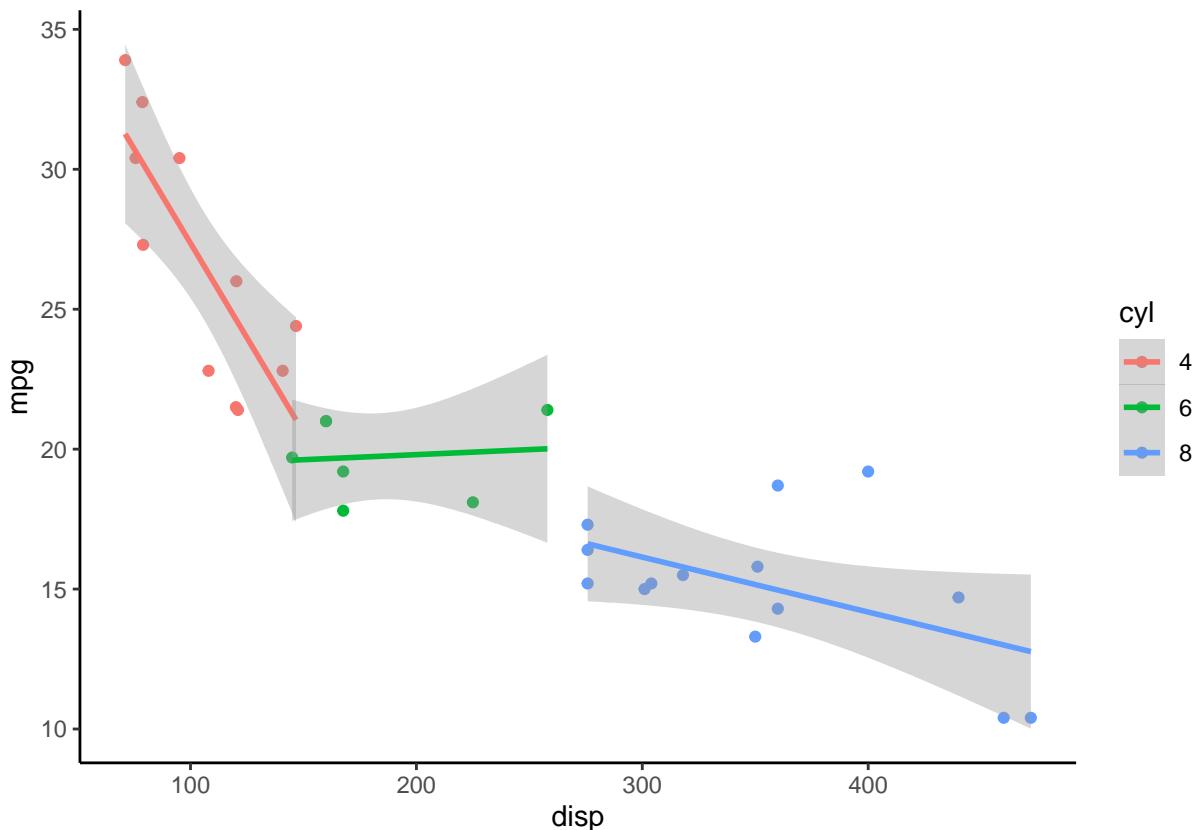


```
> ggplot(mtcars)+aes(x=disp,y=mpg,color=as.factor(cyl))+geom_point()+labs(color="cyl")
```



5. Ajouter le lisseur linéaire sur le graphe.

```
> ggplot(mtcars)+aes(x=disp,y=mpg,color=as.factor(cyl))+geom_point()+
+   geom_smooth(method="lm")+labs(color="cyl")
```



**Exercice 4.11** (Résidus pour régression simple).

1. Générer un échantillon  $(x_i, y_i), i = 1, \dots, 100$  selon le modèle linéaire

$$Y_i = 3 + X_i + \varepsilon_i$$

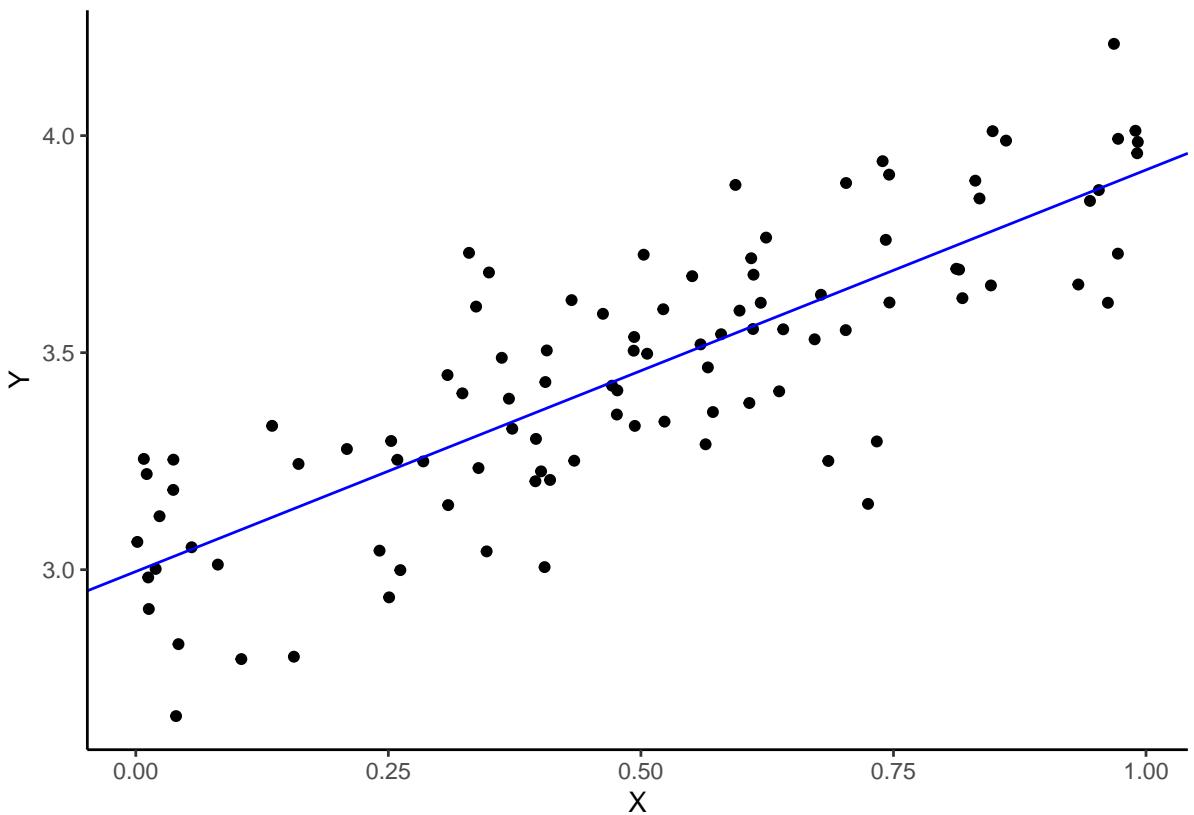
où  $X_i$  sont i.i.d. de loi uniforme sur  $[0, 1]$  et  $\varepsilon_i$  sont i.i.d. de loi gaussienne  $N(0, 0.2^2)$  (utiliser `runif` et `rnorm`).

```
> n <- 100
> X <- runif(n)
> eps <- rnorm(n, sd=0.2)
> Y <- 3+X+eps
> D <- data.frame(X,Y)
```

2. Tracer le nuage de points **Y** vs **X** et ajouter le lisseur linéaire.

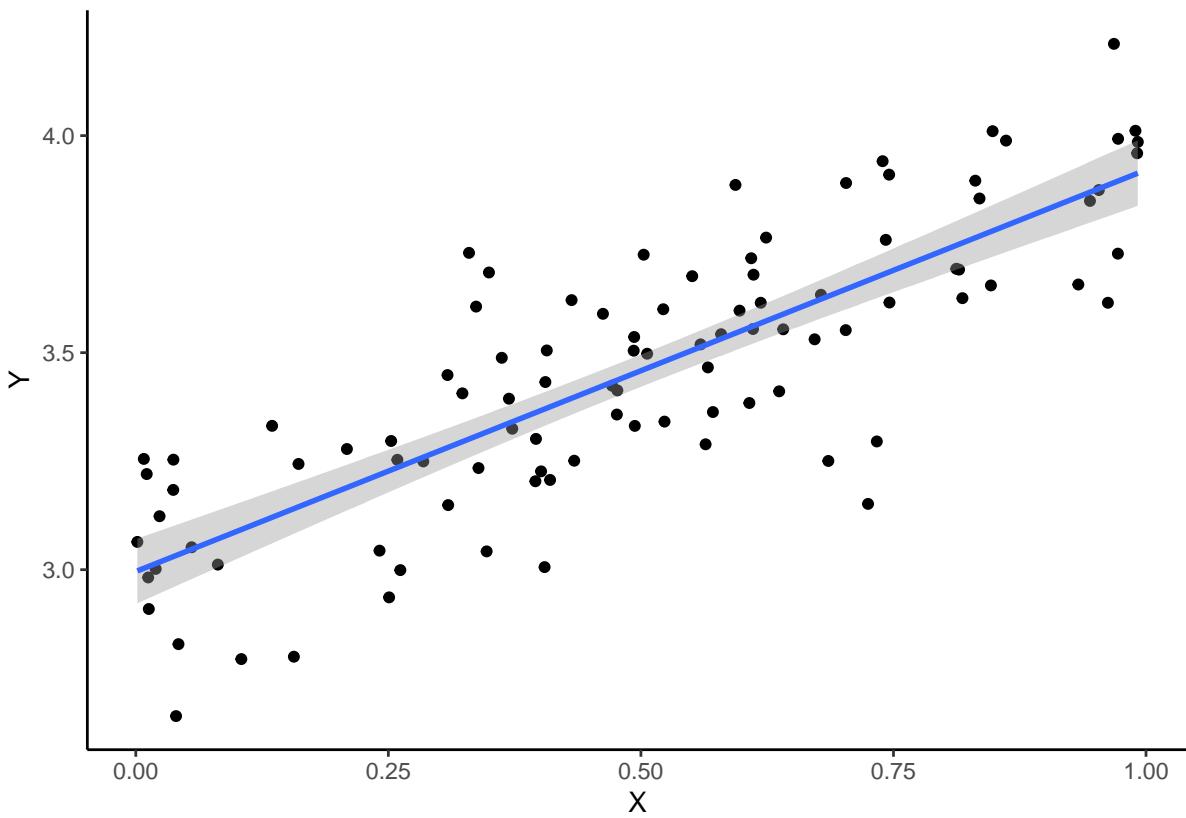
On le fait d'abord “à la main” en calculant l'équation de la droite de régression.

```
> model <- lm(Y~.,data=D)
> co <- coef(model)
> D$fit <- predict(model)
> co <- coef(lm(Y~.,data=D))
> ggplot(D)+aes(x=X,y=Y)+geom_point()+
+ geom_abline(slope=co[2],intercept=co[1],color="blue")
```



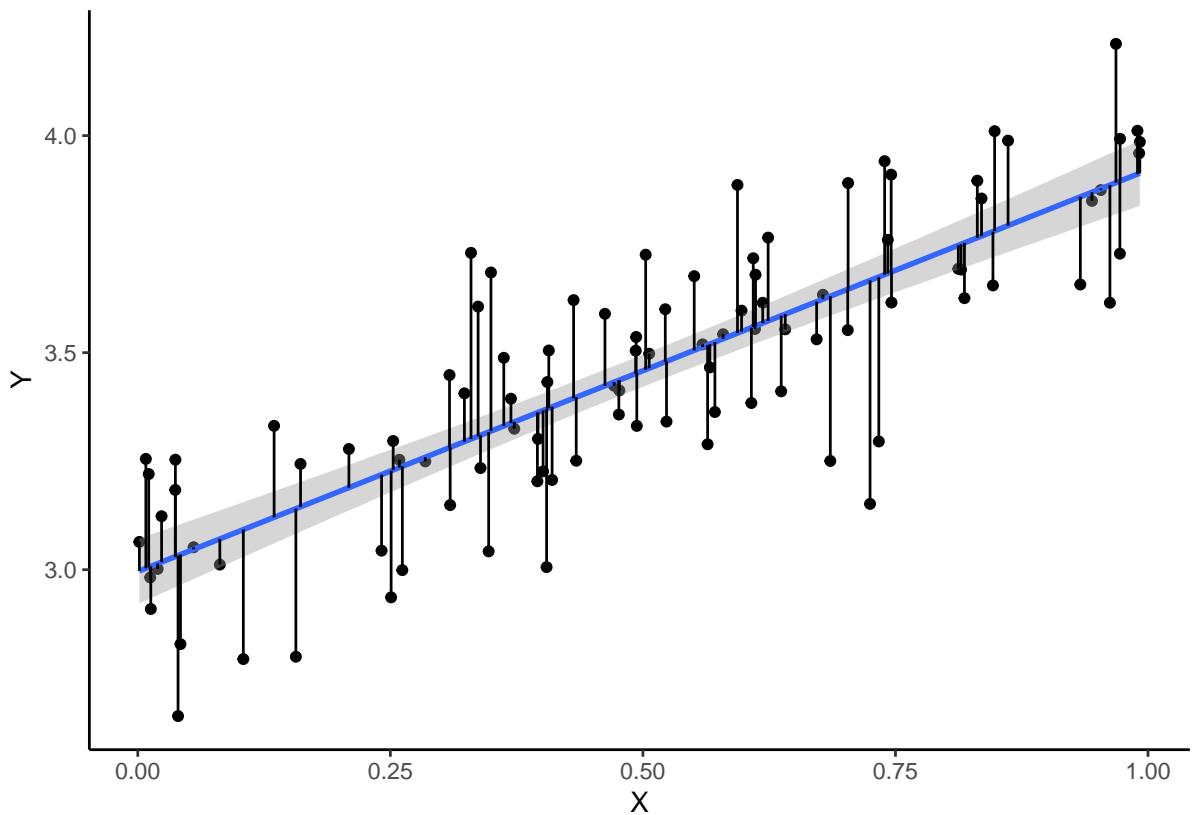
On peut avoir le tracé directement avec `geom_smooth`.

```
> ggplot(D)+aes(x=X,y=Y)+geom_point() +geom_smooth(method="lm")
```



3. Représenter les résidus : on ajoutera une ligne verticale entre chaque point et la droite de lissage (utiliser `geom_segment`).

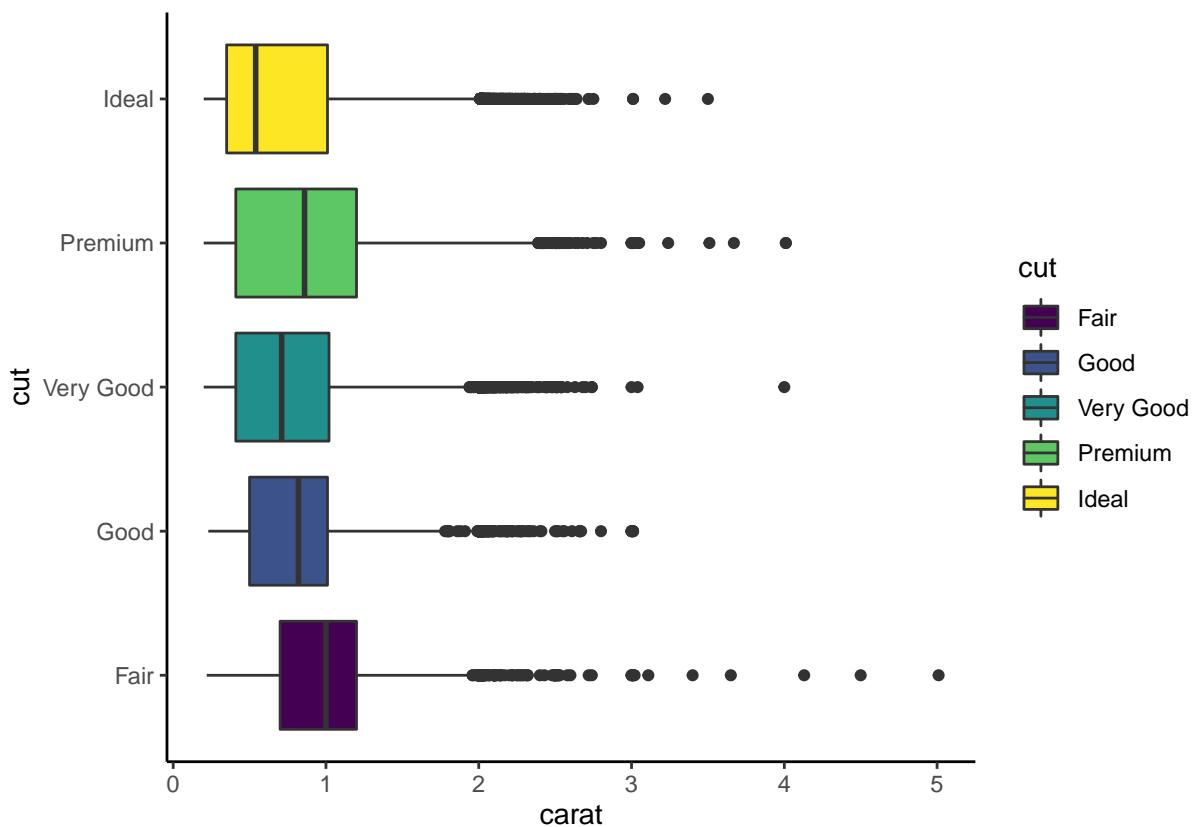
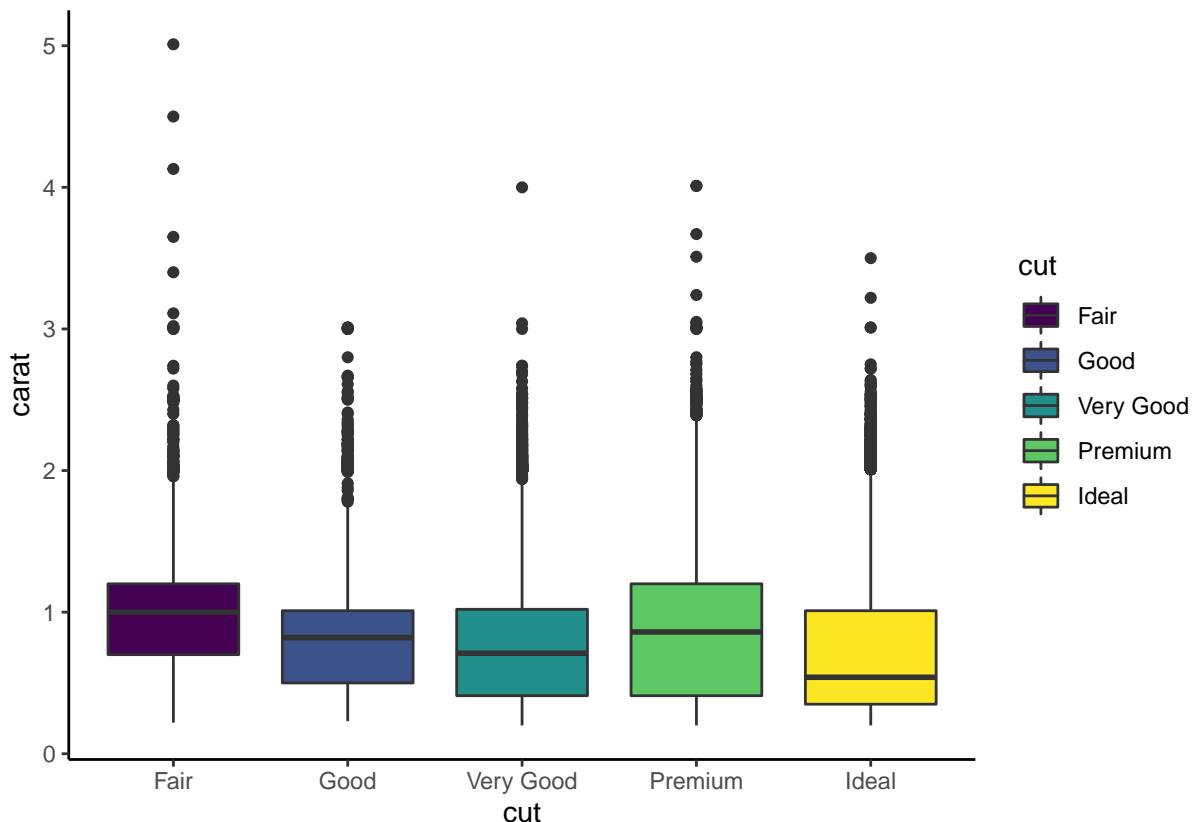
```
> ggplot(D)+aes(x=X,y=Y)+geom_point() +geom_smooth(method="lm")+
+  geom_segment(aes(xend=X,yend=fit))
```

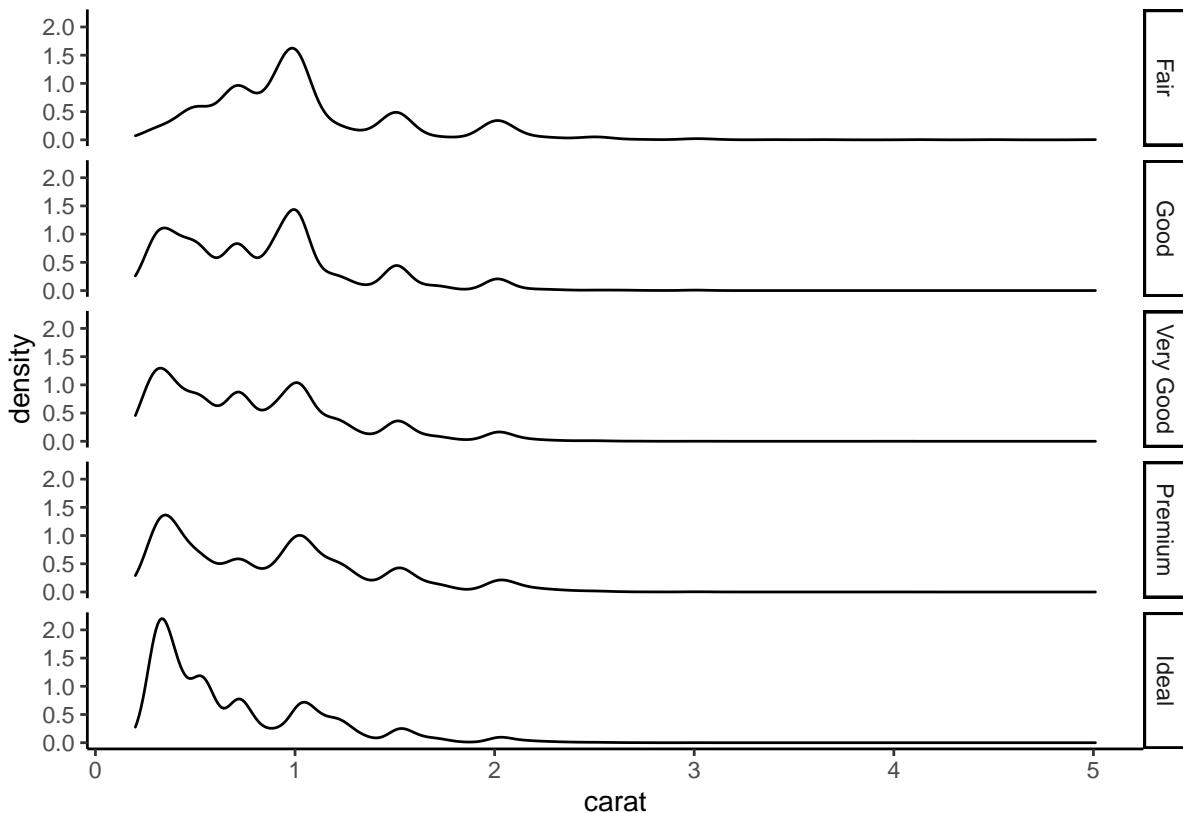


**Exercice 4.12** (Challenge).

On considère les données **diamonds**.

1. Tracer les graphes suivants (utiliser `coord_flip` pour le second).

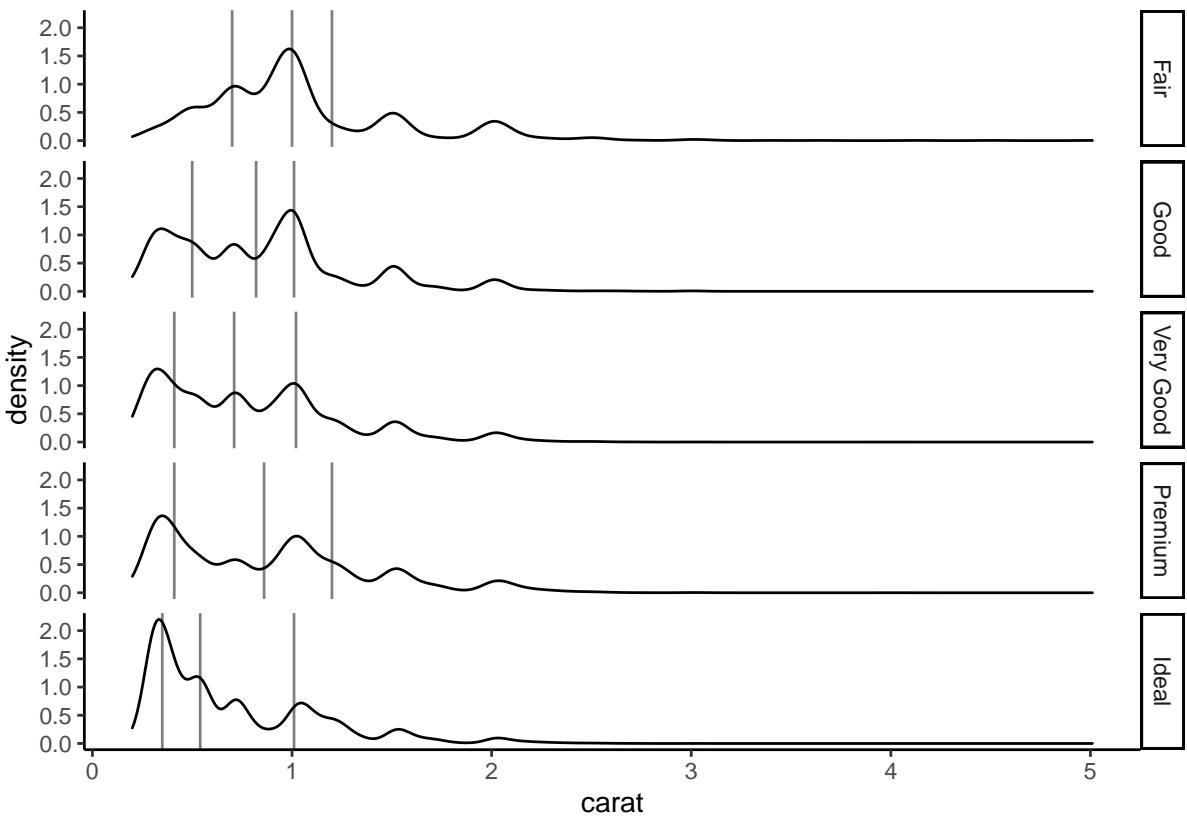




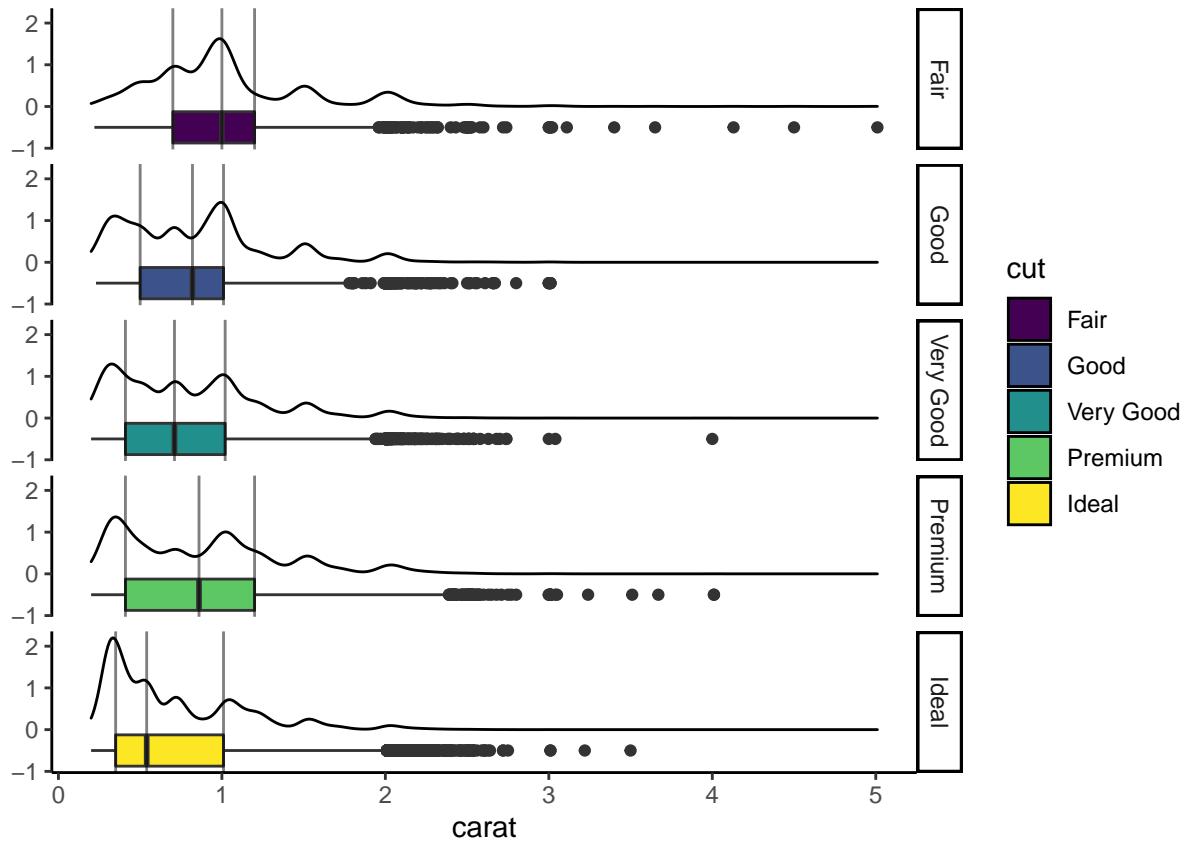
```
> ggplot(data=diamonds) + geom_boxplot(aes(x=cut,y=carat,fill=cut))
> ggplot(data=diamonds) + geom_boxplot(aes(x=cut,y=carat,fill=cut))+coord_flip()
> ggplot(data=diamonds) + geom_density(aes(x=carat,y=..density..)) + facet_grid(cut~.)
```

2. Ajouter sur le troisième graphe les quartiles de la variable **carat** pour chaque valeur de **cut**. On utilisera une ligne verticale.

```
> Q1 <- diamonds %>% group_by(cut) %>%
+   summarize(q1=quantile(carat,c(0.25)),q2=quantile(carat,c(0.5)),
+             q3=quantile(carat,c(0.75)))
> quantildf <- Q1%>% gather(key="alpha",value="quantiles",-cut)
> ggplot(data=diamonds) + geom_density(aes(x=carat,y=..density..))+ 
+   facet_grid(cut~.) +
+   geom_vline(data=quantildf,aes(xintercept=quantiles),col=alpha("black",1/2))
```



3. En déduire le graphe suivant (on utilisera le package `ggstance`).



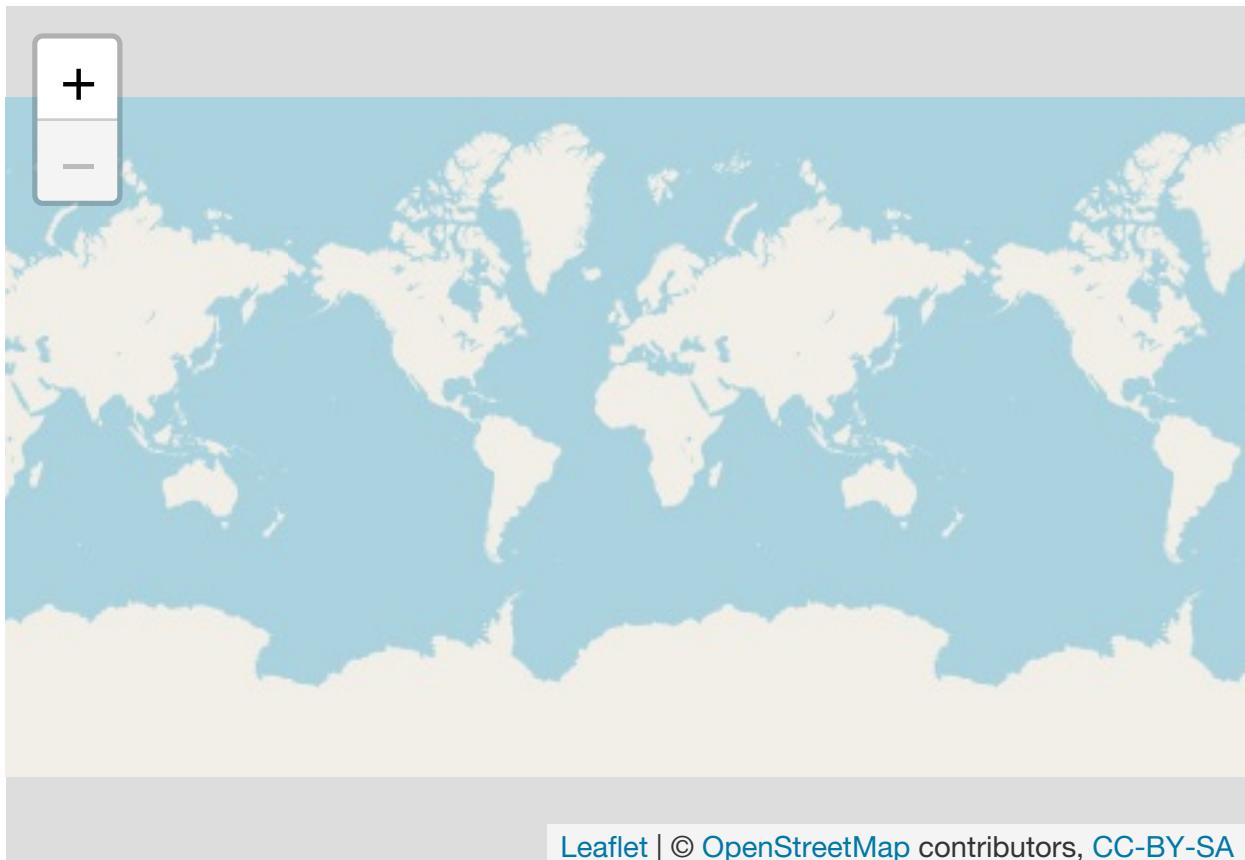
```
> library(ggstance)
> ggplot(data=diamonds) +
+   geom_boxplot(data=diamonds,aes(y=-0.5,x=carat,fill=cut)) +
+   geom_density(aes(x=carat,y=..density..)) + facet_grid(cut~.) +
+   geom_vline(data=quantildf,aes(xintercept=quantiles),col=alpha("black",1/2))+ 
+   ylab("")
```

## 5 Faire des cartes interactives avec leaflet

### 5.1 Présentation de leaflet

Leaflet est un package permettant de faire de la *cartographie interactive*. On pourra consulter un descriptif synthétique [ici](#) ainsi que le tutoriel [https://lrouviere.github.io/TUTO\\_VISU/faire-des-cartes-avec-r.html](https://lrouviere.github.io/TUTO_VISU/faire-des-cartes-avec-r.html) pour des compléments sur la cartographie avec R. Le principe est proche de **ggplot2** : les cartes sont construites à partir de couches qui se superposent. Un fond de carte s'obtient avec les fonctions **leaflet** et **addTiles**

```
> library(leaflet)
> leaflet() %>% addTiles()
```



On dispose de plusieurs styles de fonds de cartes (quelques exemples [ici](#)) :

```
> Paris <- c(2.351462,48.8567)
> m2 <- leaflet() %>% setView(lng = Paris[1], lat = Paris[2], zoom = 12) %>%
+   addTiles()
> m2 %>% addProviderTiles("Stamen.Toner")
```



> m2 %>% addProviderTiles("Wikimedia")



```
> m2 %>% addProviderTiles("Esri.NatGeoWorldMap")
```

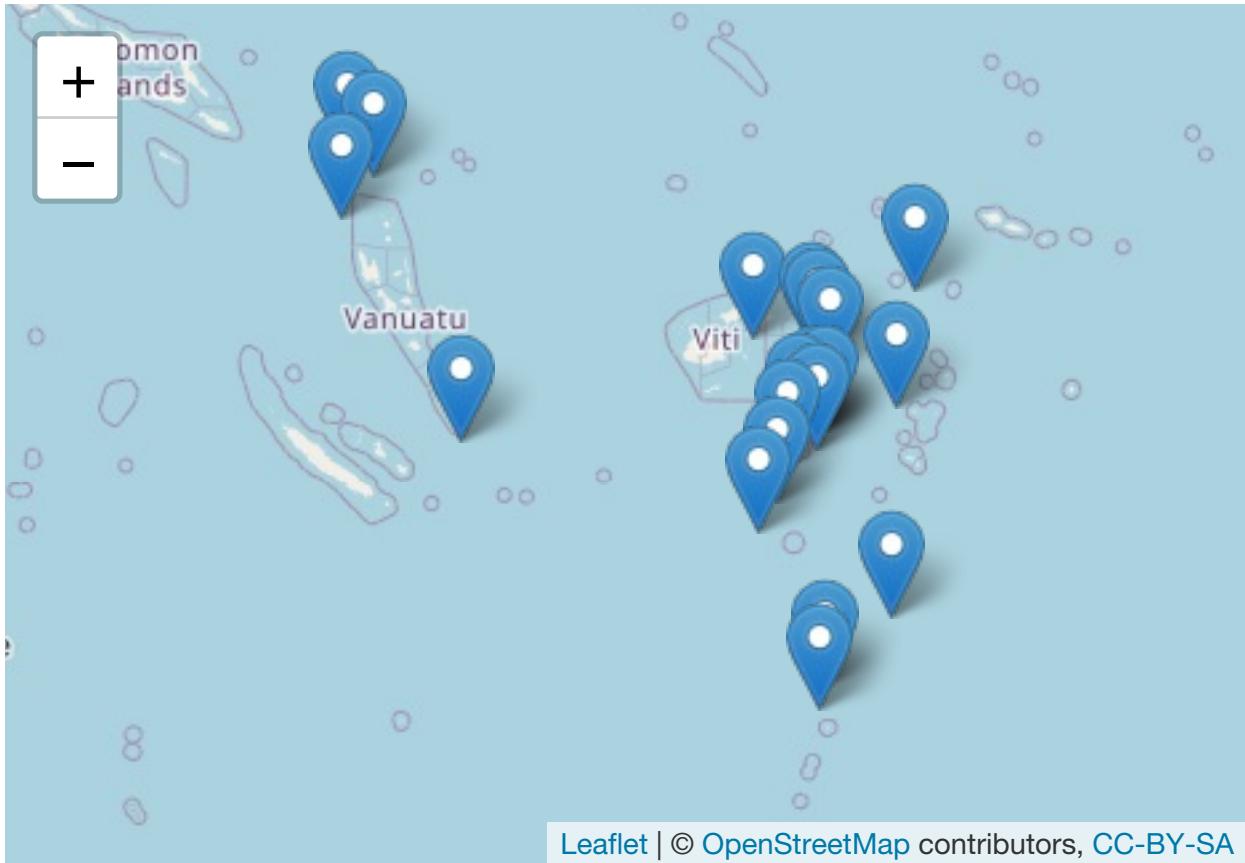


```
> m2 %>%
+   addProviderTiles("Stamen.Watercolor") %>%
+   addProviderTiles("Stamen.TonerHybrid")
```



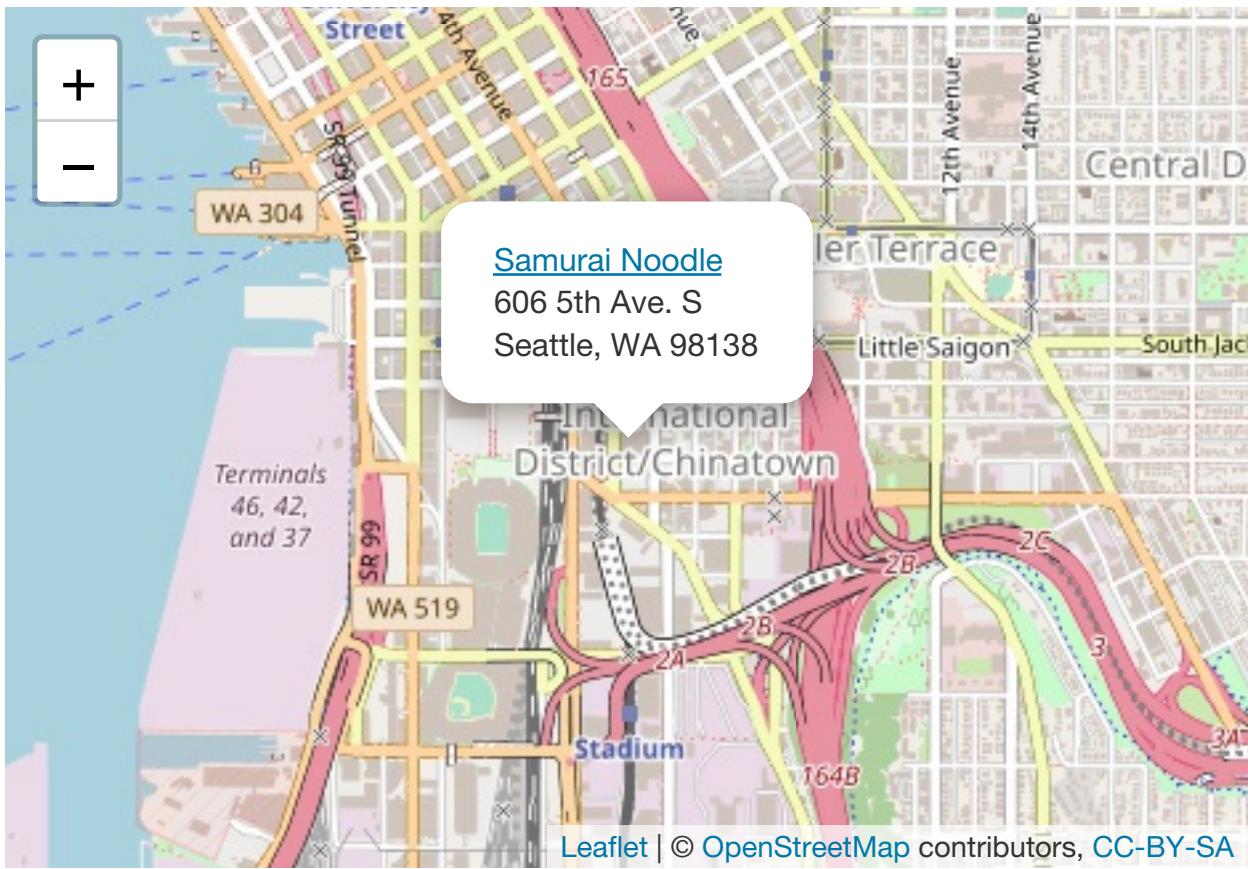
Il est fréquemment utile de repérer des lieux sur une carte à l'aide de symboles. On pourra effectuer cela à l'aide des fonctions `addMarkers` et `addCircles`...

```
> data(quakes)
> leaflet(data = quakes[1:20,]) %>% addTiles() %>%
+   addMarkers(~long, ~lat, popup = ~as.character(mag))
```



Le caractère interactif de la carte permet d'ajouter de l'information lorsqu'on clique sur un marker (grâce à l'option **popup**). On peut également ajouter des **popups** qui contiennent plus d'information, voire des liens vers des sites web :

```
> content <- paste(sep = "<br/>",
+   "<b><a href='http://www.samurainoodle.com'>Samurai Noodle</a></b>",
+   "606 5th Ave. S",
+   "Seattle, WA 98138"
+ )
>
> leaflet() %>% addTiles() %>%
+   addPopups(-122.327298, 47.597131, content,
+   options = popupOptions(closeButton = FALSE)
+ )
```



### Exercice 5.1 (Popup avec leaflet).

Placer un **popup** localisant l'Université Rennes 2 (Campus Villejean). On ajoutera un lien renvoyant sur le site de l'Université. On pourra utiliser la fonction **mygeocode** ce-dessous qui permet de géolocaliser des lieux à partir d'adresses.

```
> if (!require(jsonlite)) install.packages("jsonlite")
> mygeocode <- function(adresses){
+ # adresses est un vecteur contenant toutes les adresses sous forme de chaîne de caractères
+ nominatim_osm <- function(address = NULL){
+   ## details: http://wiki.openstreetmap.org/wiki/Nominatim
+   ## fonction nominatim_osm proposée par D.Kisler
+   if(suppressWarnings(is.null(address)))  return(data.frame())
+   tryCatch(
+     d <- jsonlite::fromJSON(
+       gsub('@@addr@', gsub('\s+', '\%20', address),
+             'http://nominatim.openstreetmap.org/search/@addr@?format=json&addressdetails=0&limit=1')
+     ), error = function(c)  return(data.frame())
+   )
+   if(length(d) == 0) return(data.frame())
+   return(c(as.numeric(d$lon), as.numeric(d$lat)))
+ }
+ tableau <- t(sapply(adresses,nominatim_osm))
+ colnames(tableau) <- c("lon","lat")
+ return(tableau)
+ }
```

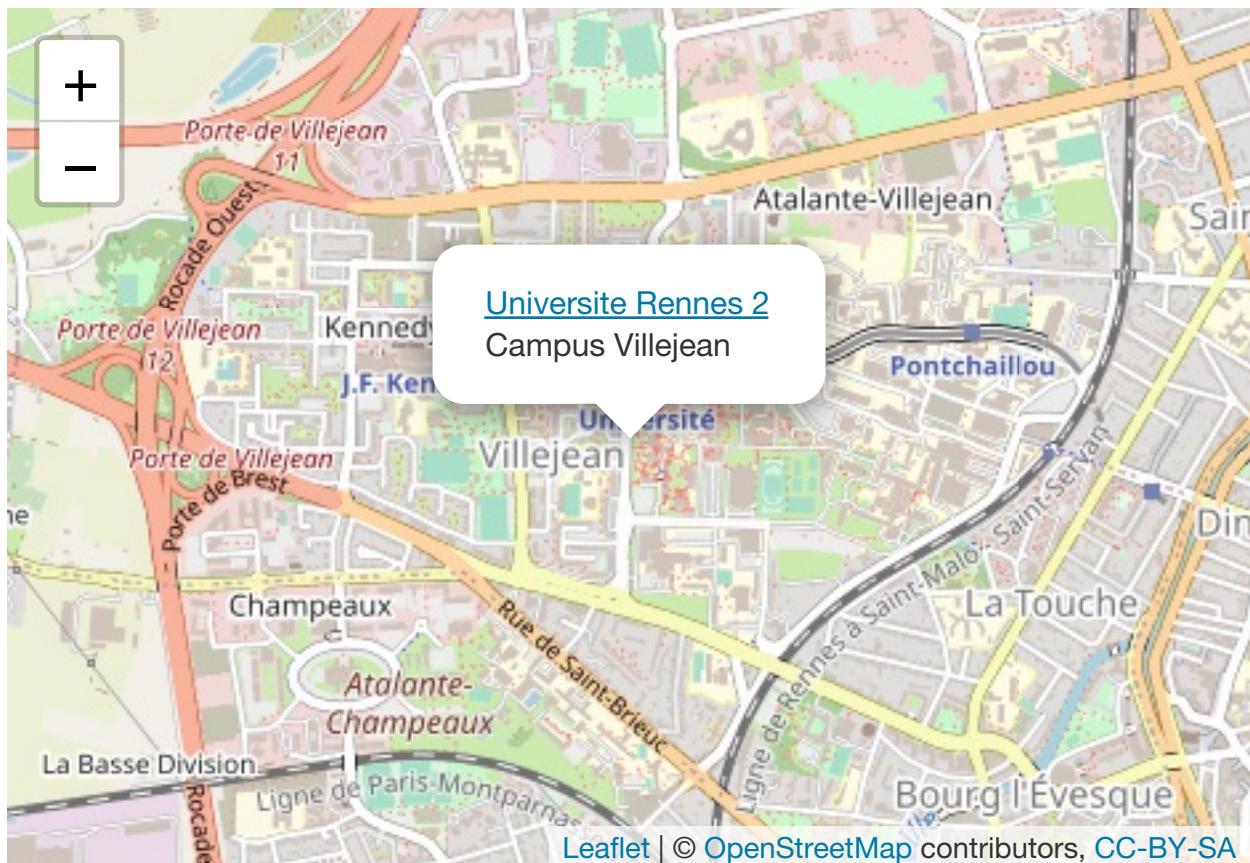
Par exemple

```

> mygeocode("Paris")
  lon      lat
Paris 2.351462 48.8567

> R2 <- mygeocode("Universite Rennes 2 Villejean") %>% as_tibble()
> info <- paste(sep = "<br/>",
+   "<b><a href='https://www.univ-rennes2.fr'>Universite Rennes 2</a></b>",
+   "Campus Villejean")
>
>
> leaflet() %>% addTiles() %>%
+   addPopups(R2[1]$lon, R2[2]$lat, info, options = popupOptions(closeButton = FALSE))

```



## 5.2 Challenge : Visualisation des stations velib à Paris

Plusieurs villes dans le monde ont accepté de mettre en ligne les données sur l'occupation des stations velib. Ces données sont facilement accessibles et mises à jour en temps réel. On dispose généralement de la taille et la localisation des stations, la proportion de vélos disponibles... Il est possible de requérir (entre autres) :

- sur les données Decaux
- sur Open data Paris
- sur vlstats pour des données mensuelles ou historiques ou encore sur Velib pour obtenir des fichiers qui sont rafraîchis régulièrement.

1. Récupérer les données actuelles de velib disponibles pour la ville de Paris : <https://opendata.par>

<https://opendata.paris.fr/explore/dataset/velib-disponibilite-en-temps-reel/information/>. On pourra utiliser la fonction `read_delim` avec l'option `delim=";"`.

```
> lien <- "https://opendata.paris.fr/explore/dataset/velib-disponibilite-en-temps-reel/
+ download/?format=csv&timezone=Europe/Berlin&use_labels_for_header=true"
> sta.Paris <- read_delim(lien,delim=";")
```

2. Décrire les variables du jeu de données.

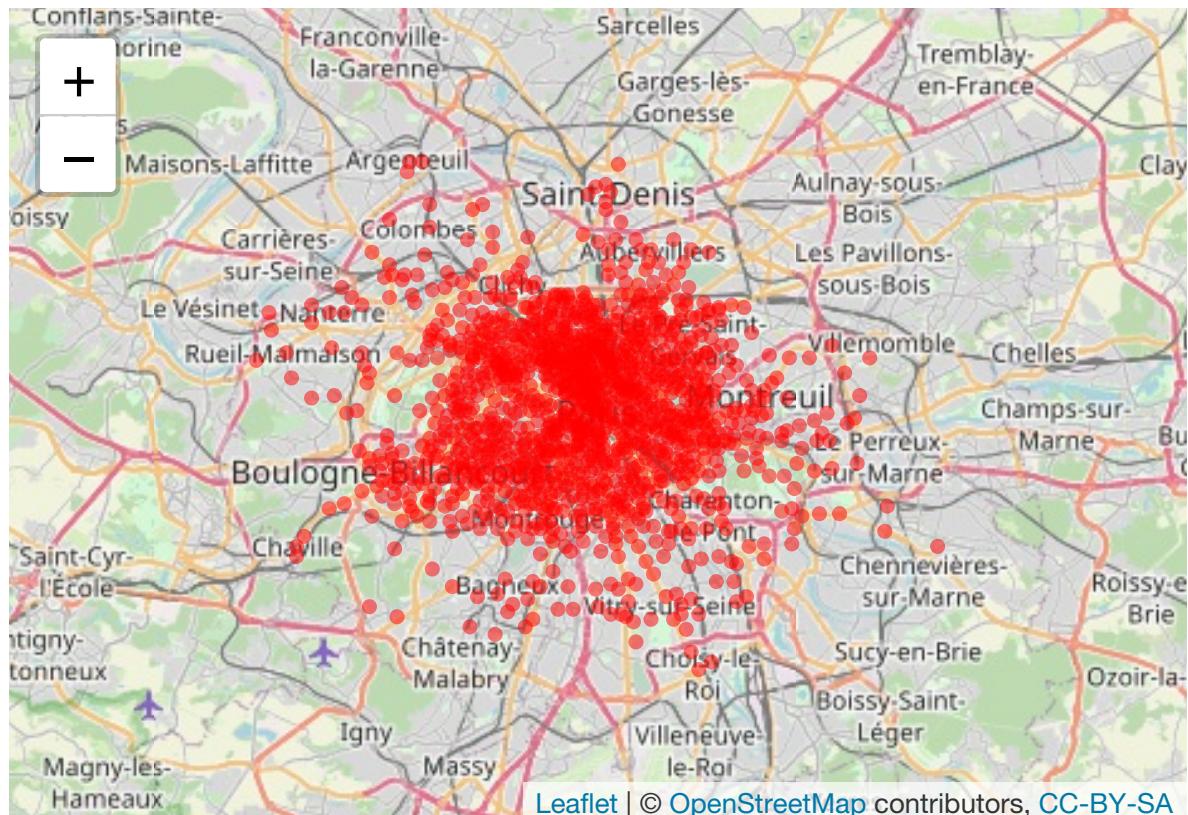
Nous avons de l'information sur la disponibilité, le remplissage... de stations velib parisiennes.

3. Créer une variable **latitude** et une variable **longitude** à partir de la variable **geo**.

```
> sta.Paris1 <- sta.Paris %>% separate(`Coordonnées géographiques`,
+   into=c("lat","lon"),sep=",") %>%
+   mutate(lat=as.numeric(lat),lon=as.numeric(lon))
```

4. Visualiser les positions des stations sur une carte leaflet.

```
> map.velib1 <- leaflet(data = sta.Paris1) %>%
+   addTiles() %>%
+   addCircleMarkers(~ lon, ~ lat,radius=3,
+   stroke = FALSE, fillOpacity = 0.5,color="red"
+   )
>
> map.velib1
```



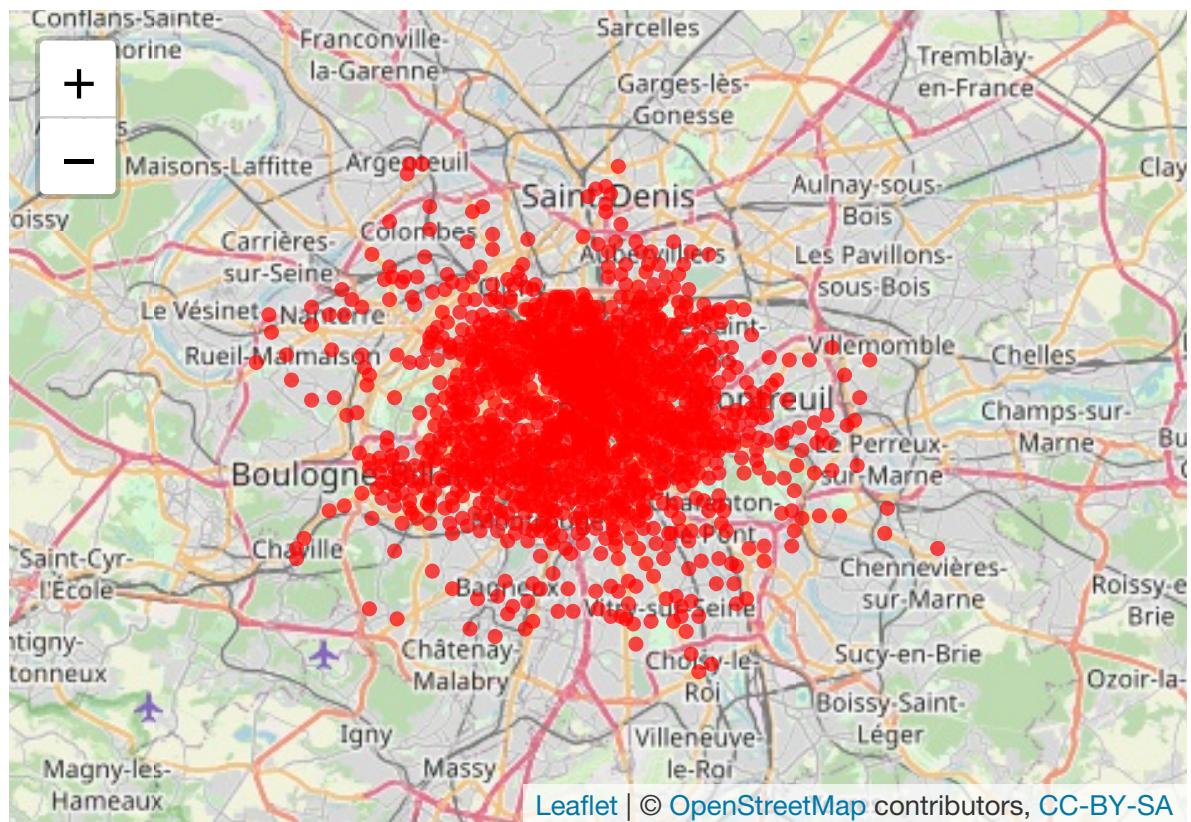
5. Ajouter un popup qui permet de connaître le nombre de vélos disponibles (électriques+mécanique) quand on clique sur la station (on pourra utiliser l'option **popup** dans la fonction **addCircleMarkers**).

```
> map.velib2 <- leaflet(data = sta.Paris1) %>%
+   addTiles() %>%
```

```

+   addCircleMarkers(~ lon, ~ lat, radius=3, stroke = FALSE,
+                     fillOpacity = 0.7, color="red",
+                     popup = ~ sprintf("<b> Vélos dispos: %s</b>",
+                                       as.character(`Nombre total vélos disponibles`)))
+
>
> #ou sans sprintf
>
> map.velib2 <- leaflet(data = sta.Paris1) %>%
+   addTiles() %>%
+   addCircleMarkers(~ lon, ~ lat, radius=3, stroke = FALSE, fillOpacity = 0.7, color="red",
+                     popup = ~ paste("Vélos dispos :",
+                                       as.character(`Nombre total vélos disponibles`)))
+
>
> map.velib2

```

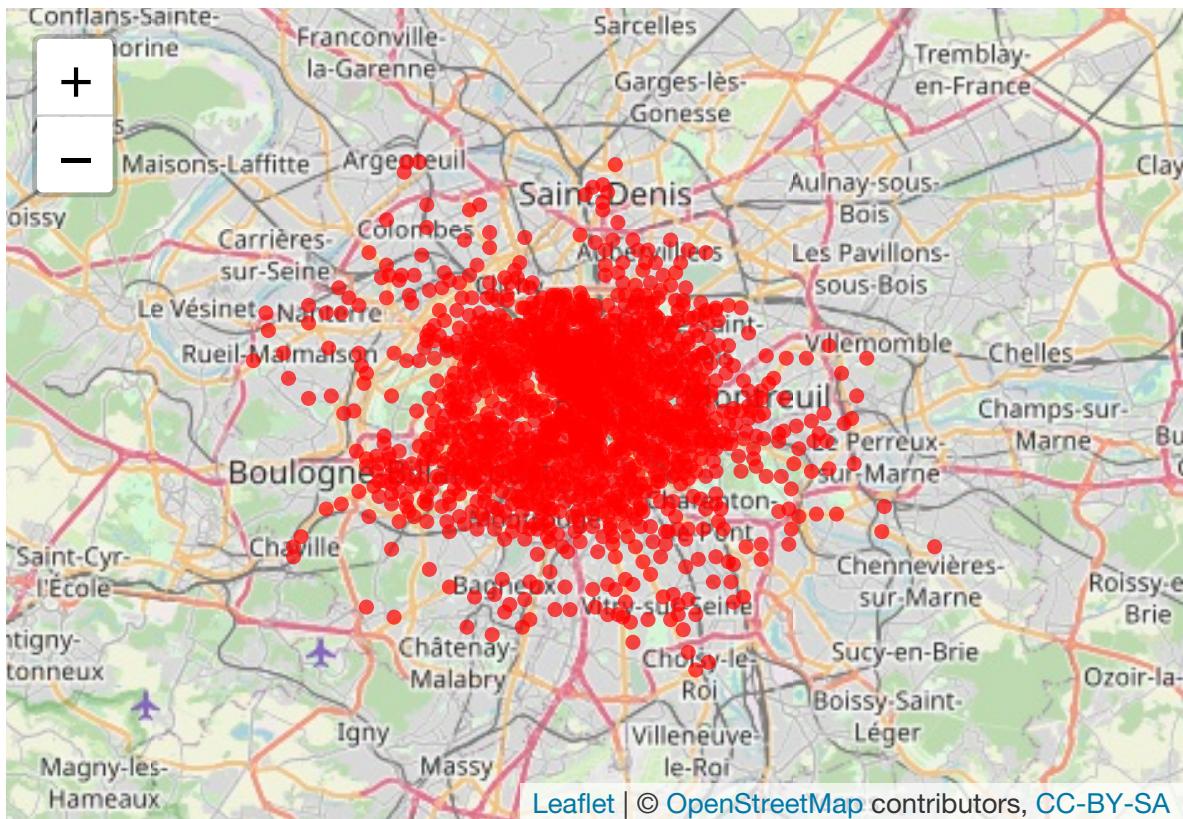


- Ajouter le nom de la station dans le popup.

```

> map.velib3 <- leaflet(data = sta.Paris1) %>%
+   addTiles() %>%
+   addCircleMarkers(~ lon, ~ lat, radius=3, stroke = FALSE,
+                     fillOpacity = 0.7, color="red",
+                     popup = ~ paste(as.character(`Nom station`),", Vélos dispos :",
+                                       as.character(`Nombre total vélos disponibles`),
+                                       sep=""))
+
>
> map.velib3

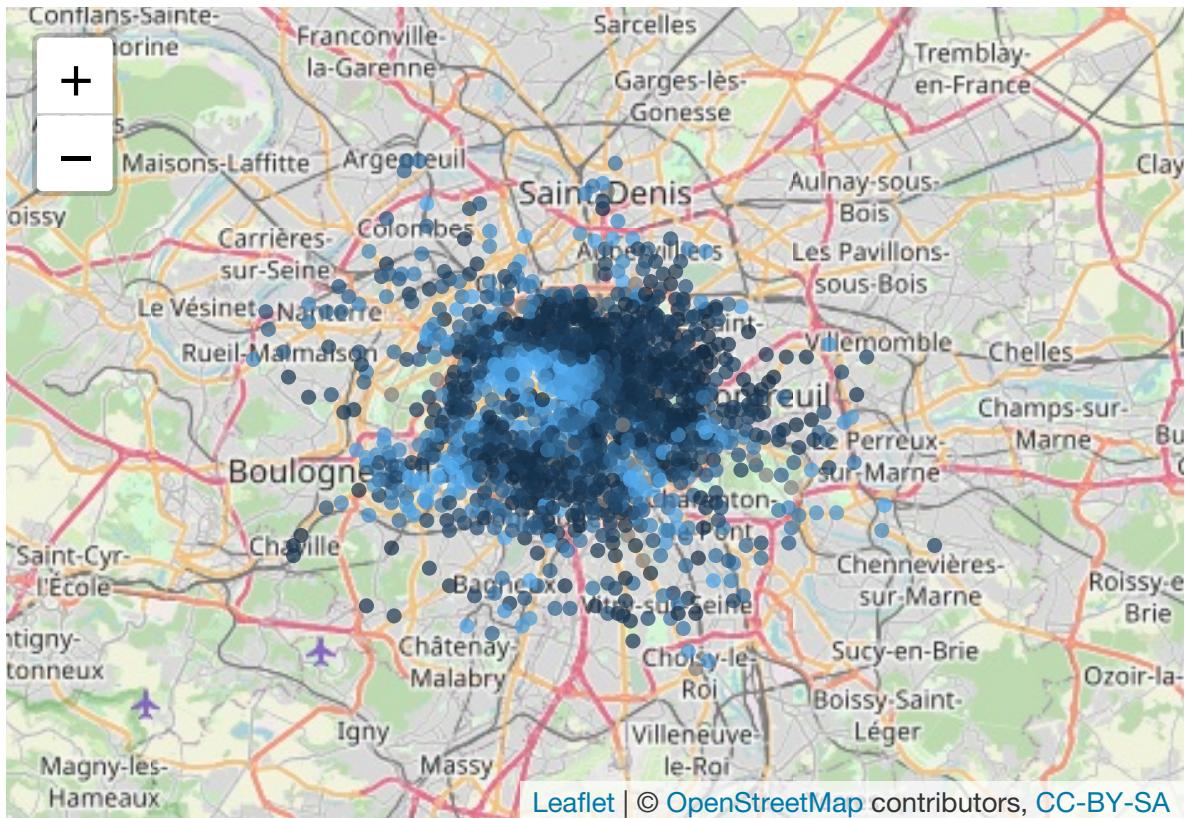
```



7. Faire de même en utilisant des couleurs différentes en fonction de la proportion de vélos disponibles dans la station. On pourra utiliser les palettes de couleur

```
> ColorPal1 <- colorNumeric(scales::seq_gradient_pal(low = "#132B43", high = "#56B1F7",
+   space = "Lab"), domain = c(0,1))
> ColorPal2 <- colorNumeric(scales::seq_gradient_pal(low = "red", high = "black",
+   space = "Lab"), domain = c(0,1))

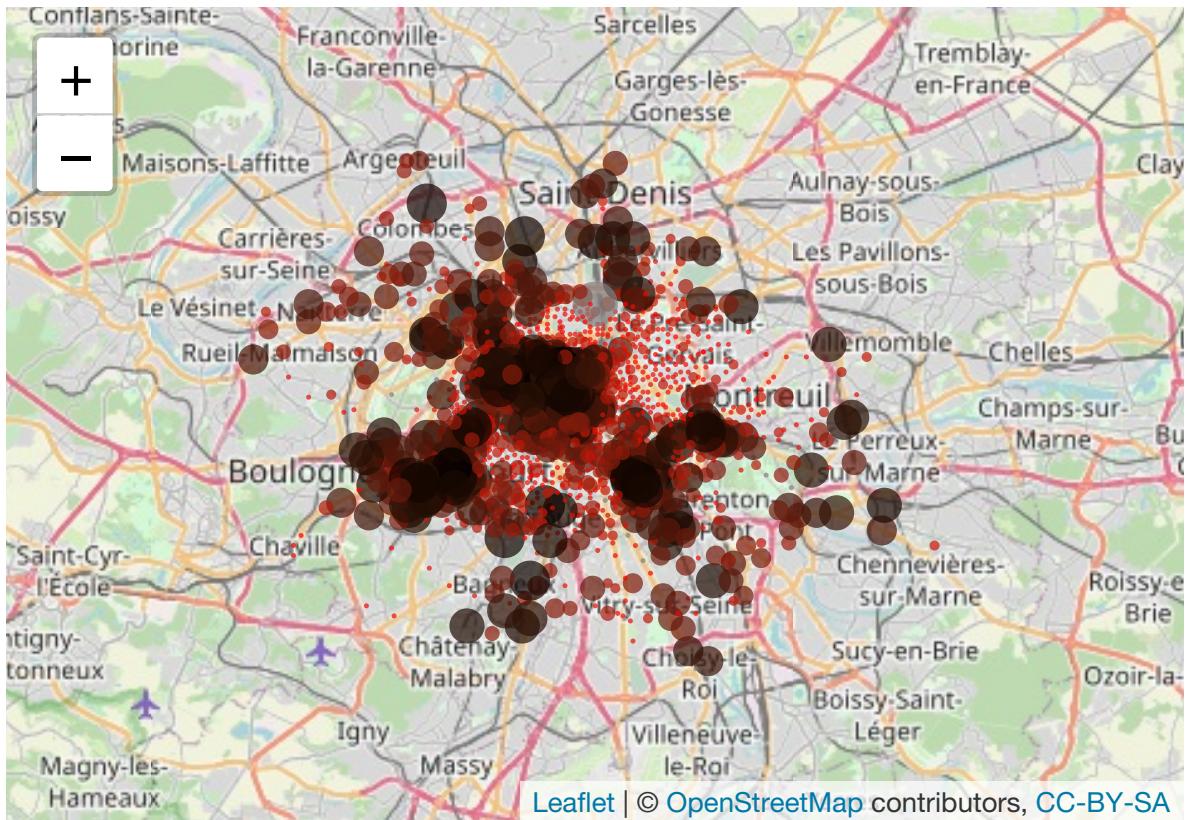
> map.velib4 <- leaflet(data = sta.Paris1) %>%
+   addTiles() %>%
+   addCircleMarkers(~ lon, ~ lat, radius=3, stroke = FALSE, fillOpacity = 0.7,
+                   color=~ColorPal1(`Nombre total vélos disponibles`/
+                                     `Capacité de la station`),
+                   popup = ~ paste(as.character(`Nom station`),", Vélos dispos :",
+                                 as.character(`Nombre total vélos disponibles`),
+                                 sep=""))
>
> map.velib4
```



```

> map.velib5 <- leaflet(data = sta.Paris1) %>%
+   addTiles() %>%
+   addCircleMarkers(~ lon, ~ lat, stroke = FALSE, fillOpacity = 0.7,
+     color=~ColorPal2(`Nombre total vélos disponibles`/
+       `Capacité de la station`),
+     radius=~(`Nombre total vélos disponibles`/
+       `Capacité de la station`)*8,
+     popup = ~ paste(as.character(`Nom station`),", Vélos dispos :",
+                   as.character(`Nombre total vélos disponibles`),
+                   sep=""))
>
> map.velib5

```

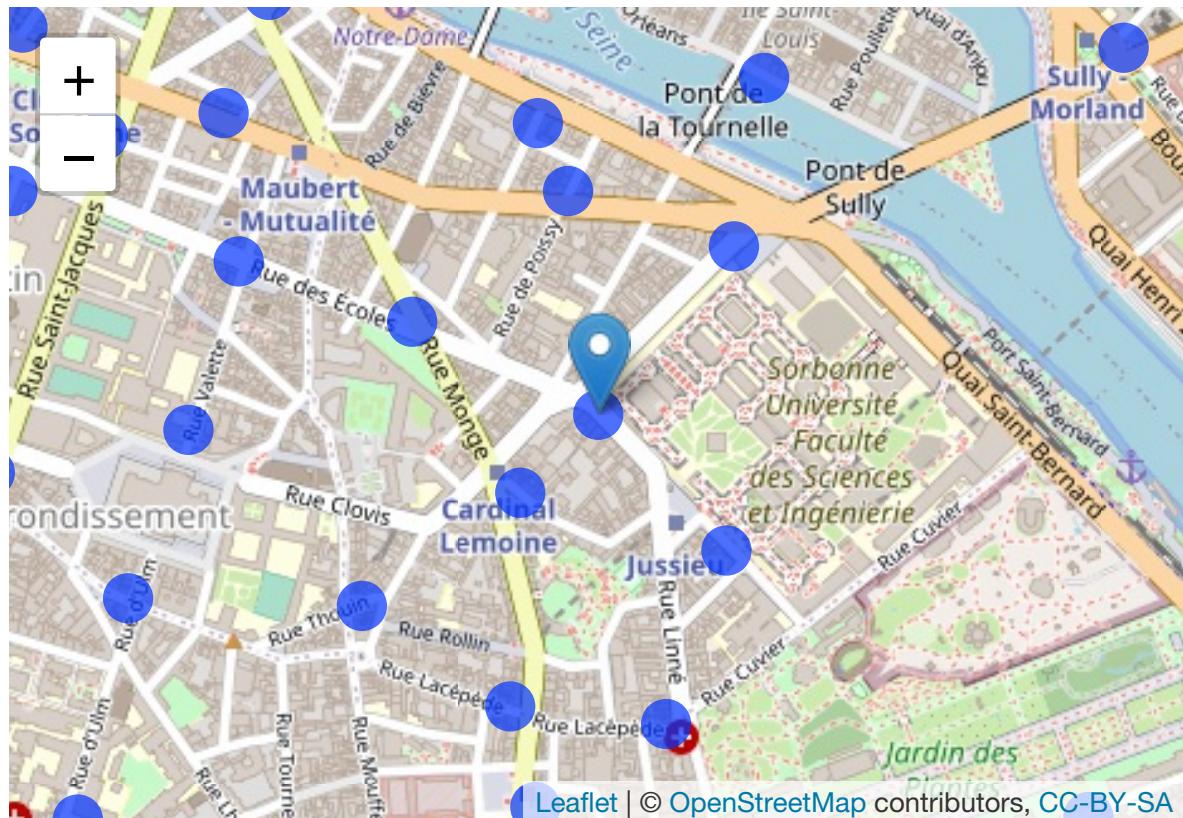


8. Créer une fonction `local.station` qui permette de visualiser quelques stations autour d'une station choisie.

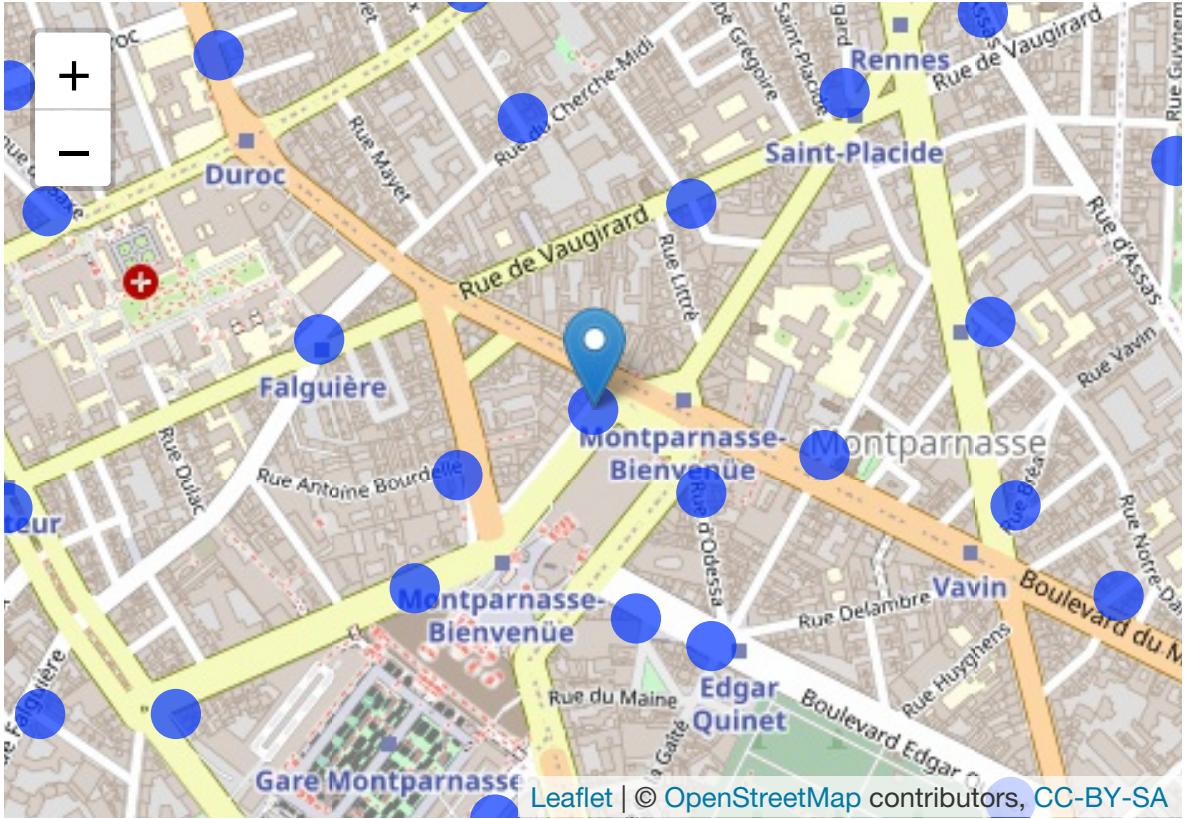
```
> nom.station <- "Jussieu - Fossés Saint-Bernard"
> local.station <- function(nom.station){
+   df <- sta.Paris1 %>% filter(`Nom station` == nom.station)
+   leaflet(data = sta.Paris1) %>% setView(lng=df$lon, lat=df$lat, zoom=15) %>%
+   addTiles() %>%
+   addCircleMarkers(~ lon, ~ lat, stroke = FALSE, fillOpacity = 0.7,
+                   popup = ~ paste(as.character(`Nom station`), ", Vélos dispos :",
+                                   as.character(`Nombre total vélos disponibles`),
+                                   sep=""))
+   addMarkers(lng=df$lon, lat=df$lat,
+             popup = ~ paste(nom.station, ", Vélos dispos :",
+                             as.character(df$`Nombre total vélos disponibles`),
+                             sep=""),
+             popupOptions = popupOptions(noHide = T))
+ }
```

La fonction devra par exemple renvoyer

```
> local.station("Jussieu - Fossés Saint-Bernard")
```



```
> local.station("Gare Montparnasse - Arrivée")
```



## 6 Faire de la régression sur R

Les problèmes de régression et de classification supervisée consistent à expliquer et/ou prédire une sortie  $y \in \mathcal{Y}$  avec

- $\mathcal{Y} = \mathbb{R}$  pour la régression
- $\mathcal{Y}$  de cardinal fini pour la classification supervisée,

par des entrées  $x \in \mathbb{R}^p$ . Il s'agit donc de trouver une fonction

$$m : \mathbb{R}^p \rightarrow \mathcal{Y}$$

à partir de données  $(X_1, Y_1), \dots, (X_n, Y_n)$ .

Ces données sont souvent collectées dans un **dataframe** df de la forme

$Y$	$X_1$	$X_2$	$\dots$	$X_p$
$y_1$	$x_{1,1}$	$x_{1,2}$	$\dots$	$x_{1,p}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$y_n$	$x_{n,1}$	$x_{n,2}$	$\dots$	$x_{n,p}$

Le protocole pour construire un algorithme de régression sur **R** est toujours le même. Il faut spécifier :

- la méthode (ou l'algorithme)

- la variable à expliquer
- les variables explicatives
- le jeu de données
- les éventuelles options de la méthode considérée.

Par exemple la commande

```
> method(Y~X1+X3,data=df,...)
```

ajustera le modèle **method** pour expliquer  $Y$  par  $X_1$  et  $X_3$  avec les données dans **df** (les points représentent d'éventuelles options). Voici quelques exemples de méthodes :

fonction R	algorithme	Package	Problème
<b>lm</b>	modèle linéaire		Reg
<b>glm</b>	modèle logistique		Class
<b>lda</b>	analyse discriminante linéaire	MASS	Class
<b>svm</b>	Support Vector Machine	e1071	Class
<b>knn.reg</b>	plus proches voisins	FNN	Reg
<b>knn</b>	plus proches voisins	class	Class
<b>rpart</b>	arbres	rpart	Reg et Class
<b>glmnet</b>	ridge et lasso	glmnet	Reg et Class
<b>gbm</b>	boosting	gbm	Reg et Class
<b>randomForest</b>	forêts aléatoires	randomForest	Reg et Class

**Remarque** : pour **glmnet**, on ne peut pas utiliser de formule de la forme  $Y \sim \dots$ . Il faut spécifier une matrice pour les  $X$  et un vecteur pour  $Y$ . La fonction **model.matrix** peut se révéler très utile pour calculer la matrice des  $X$ .

Puisqu'il existe un grand nombre d'algorithmes pour répondre à un même problème de régression, il est important de définir des critères de performance afin de les comparer. Ces critères sont généralement inconnus et doivent être estimés à l'aide de procédure de type **apprentissage/validation** ou **validation croisée**. On a souvent besoin d'utiliser la fonction **predict** pour calculer ces critères. Cette fonction est une **fonction générique** : on peut utiliser **predict** pour une régression linéaire, logistique, un arbre, une forêt aléatoire... Pour obtenir l'aide de cette fonction pour

- la régression linéaire : taper **help(predict.lm)**
- la régression logistique : taper **help(predict.glm)**
- les régressions pénalisées : taper **help(predict.glmnet)**
- les arbres : taper **help(predict.rpart)**
- les forêts aléatoires : taper **help(predict.randomForest)**
- ...

Dans la suite on suppose que  $\mathcal{Y} = \mathbb{R}$  et on considère le modèle de régression

$$Y = m(X) + \varepsilon.$$

La performance d'un estimateur  $\hat{m}$  de  $m$  sera mesurée par son erreur quadratique de prédiction :

$$E[(Y - \hat{m}(X))^2].$$

## 6.1 Modèle linéaire : fonctions lm et predict

On considère le modèle de régression linéaire

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p + \varepsilon$$

où  $X_1, \dots, X_p$  sont les variables explicatives,  $Y$  la variable à expliquer et  $\varepsilon$  le terme d'erreur. On fixe  $p = 5$  et on considère les données suivantes :

```
> n <- 1000
> p <- 5
> set.seed(1234)
> X.mat <- matrix(rnorm(n*p), ncol=p)
> eps <- rnorm(n, mean = 0, sd=0.5)
> df <- data.frame(X.mat, eps)
> df <- df %>% mutate(Y=X1+X2+X3+X4+X5+eps) %>% select(-eps)
```

- Construire un modèle linéaire permettant d'expliquer  $Y$  par  $X_1, \dots, X_5$  (utiliser la fonction **lm**) et afficher les estimateurs de  $\beta_0, \dots, \beta_5$  (on pourra utiliser les fonctions **coef** et **summary**).

```
> mod1 <- lm(Y ~ ., data=df)
> coef(mod1)
(Intercept)          X1          X2          X3          X4          X5 
  0.0228707   1.0111903   1.0000752   1.0034085   1.0071250   0.9962842 

> summary(mod1)

Call:
lm(formula = Y ~ ., data = df)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.44876 -0.33840 -0.00769  0.33308  1.76883 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  0.02287    0.01543   1.482   0.139    
X1          1.01119    0.01550   65.258 <2e-16 ***  
X2          1.00008    0.01575   63.479 <2e-16 ***  
X3          1.00341    0.01524   65.829 <2e-16 ***  
X4          1.00712    0.01552   64.908 <2e-16 ***  
X5          0.99628    0.01589   62.702 <2e-16 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4872 on 994 degrees of freedom
Multiple R-squared:  0.9556,    Adjusted R-squared:  0.9554 
F-statistic: 4279 on 5 and 994 DF,  p-value: < 2.2e-16
```

- On considère le jeu de données test suivant.

```
> m <- 500
> p <- 5
> set.seed(12345)
> X.mat <- matrix(rnorm(m*p), ncol=5)
> eps <- rnorm(m, mean = 0, sd=0.5)
> df.test <- data.frame(X.mat, eps)
> df.test <- df.test %>% mutate(Y=X1+X2+X3+X4+X5+eps) %>% select(-eps)
```

Calculer, pour chaque individu de ce nouveau jeu de données, les prédictions faites par le modèle de la question précédente (utiliser la fonction **predict** avec l'option *newdata*).

```
> pred <- predict(mod1, newdata=df.test)
> head(pred)
```

```

1          2          3          4          5          6
0.09630147 -1.25027415 -0.52549286  0.19569041  3.72923032 -5.79419545

```

3. Créer un nouveau dataframe qui contiennent les valeurs prédictes  $\hat{y}_i$  à la question précédente sur une colonne et les valeurs observées  $y_i$  du jeu de données `df.test` sur une autre colonne.

```
> pred.df <- data.frame(pred, obs=df.test$Y)
```

4. A l'aide du verbe **summarize**, calculer l'erreur quadratique moyenne (estimée) du modèle linéaire :

$$\frac{1}{m} \sum_{i \in test} (\hat{y}_i - y_i)^2.$$

```

> pred.df %>% summarize(MSE=mean((pred-obs)^2))
      MSE
1 0.2326355

```

## 6.2 Sélection de variables

On considère les données suivantes

```

> n <- 1000
> p <- 105
> set.seed(1234)
> X.mat <- matrix(rnorm(n*p), ncol=p)
> eps <- rnorm(n, mean = 0, sd=0.5)
> df <- data.frame(X.mat, eps)
> df <- df %>% mutate(Y=X1+X2+X3+X4+X5+eps) %>% select(-eps)

```

issues du modèle

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p + \varepsilon$$

avec  $p = 105$ . On remarquera que seules les variables  $X_1, \dots, X_5$  sont explicatives.

1. Ajuster un modèle linéaire (fonction **lm**) sur `df` et afficher les estimateurs de  $\beta_0, \dots, \beta_{105}$ .

```

> mod2 <- lm(Y ~ ., data=df)
> summary(mod2)$coefficients %>% head()
    Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.01307274 0.01660197 -0.787421 4.312441e-01
X1           0.98461851 0.01656206 59.450240 7.137528e-313
X2           0.99625236 0.01668382 59.713691 3.032293e-314
X3           1.01858539 0.01628043 62.565035 0.000000e+00
X4           1.00691542 0.01643050 61.283315 2.371515e-322
X5           1.00752931 0.01718708 58.621324 1.561036e-308

```

2. On propose d'utiliser une procédure de sélection de variables **backward** à partir du critère **BIC**. Effectuer cette procédure à l'aide de la fonction **step** (on pourra utiliser les options `direction="backward"` et `k=log(n)`). On appellera ce modèle **mod.step**.

```

> mod.step <- step(mod2, direction=c("backward"), k=log(n), trace=0)
> summary(mod.step)

```

Call:

```
lm(formula = Y ~ X1 + X2 + X3 + X4 + X5 + X29 + X69 + X74, data = df)
```

Residuals:

Min	1Q	Median	3Q	Max
-1.63923	-0.34301	0.00179	0.32041	1.45661

```

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.002413  0.015749 -0.153  0.87828
X1          0.992339  0.015807 62.777 < 2e-16 ***
X2          0.991358  0.016097 61.588 < 2e-16 ***
X3          1.010115  0.015562 64.907 < 2e-16 ***
X4          1.006043  0.015830 63.552 < 2e-16 ***
X5          1.008520  0.016242 62.093 < 2e-16 ***
X29         -0.043358  0.015158 -2.860  0.00432 **
X69         0.042714  0.015292  2.793  0.00532 **
X74         -0.043792  0.016118 -2.717  0.00670 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4969 on 991 degrees of freedom
Multiple R-squared:  0.954, Adjusted R-squared:  0.9537
F-statistic: 2571 on 8 and 991 DF, p-value: < 2.2e-16

```

On a sélectionné un modèle avec 8 variables : les 5 explicatives et 3 variables de bruit.

### 3. Calculer les erreurs quadratiques de prévision

$$\frac{1}{m} \sum_{i \in test} (\hat{y}_i - y_i)^2$$

des deux modèles (le modèle complet et le modèle sélectionné) en utilisant le jeu de données test suivant.

```

> m <- 300
> p <- 105
> set.seed(12345)
> X.mat <- matrix(rnorm(m*p), ncol=p)
> eps <- rnorm(m, mean = 0, sd=0.5)
> df.test <- data.frame(X.mat, eps)
> df.test <- df.test %>% mutate(Y=X1+X2+X3+X4+X5+eps) %>% select(-eps)

```

On met calcules les prévisions et on les met dans un tibble :

```

> p.full <- predict(mod2, newdata=df.test)
> p.step <- predict(mod.step, newdata=df.test)
> pred.df <- tibble(full=p.full, step=p.step, obs=df.test$Y)

```

On en déduit les erreurs quadratiques moyennes :

```

> pred.df %>% summarize(MSE.full=mean((full-obs)^2), MSE.step=mean((step-obs)^2))
# A tibble: 1 x 2
  MSE.full MSE.step
    <dbl>     <dbl>
1 0.300     0.254
> #ou
> pred.df %>% summarize_at(1:2, ~mean((. - obs)^2))
# A tibble: 1 x 2
  full   step
    <dbl> <dbl>
1 0.300  0.254

```

## 6.3 Régression logistique et arbre

On considère le jeu de données **spam** disponible ici

```
> library(kernlab)
> data(spam)
```

Le problème est d'expliquer la variable **type** (un email est un spam ou non) par les 57 autres variables.

1. Séparer les données en un échantillon d'apprentissage **dapp** de taille 3000 et un échantillon test **dtest** de taille 1601. On pourra utiliser la fonction **sample**.

```
> set.seed(4321)
> perm <- sample(nrow(spam), 3000)
> dapp <- spam[perm,]
> dtest <- spam[-perm,]
```

2. Construire un modèle logistique permettant de répondre au problème en utilisant uniquement les données d'apprentissage. On utilisera la fonction **glm** avec l'option **family="binomial"**.

```
> m.logit <- glm(type~., data=dapp, family="binomial")
```

3. A l'aide de la fonction **step**, effectuer une sélection backward (ça peut prendre quelques minutes).

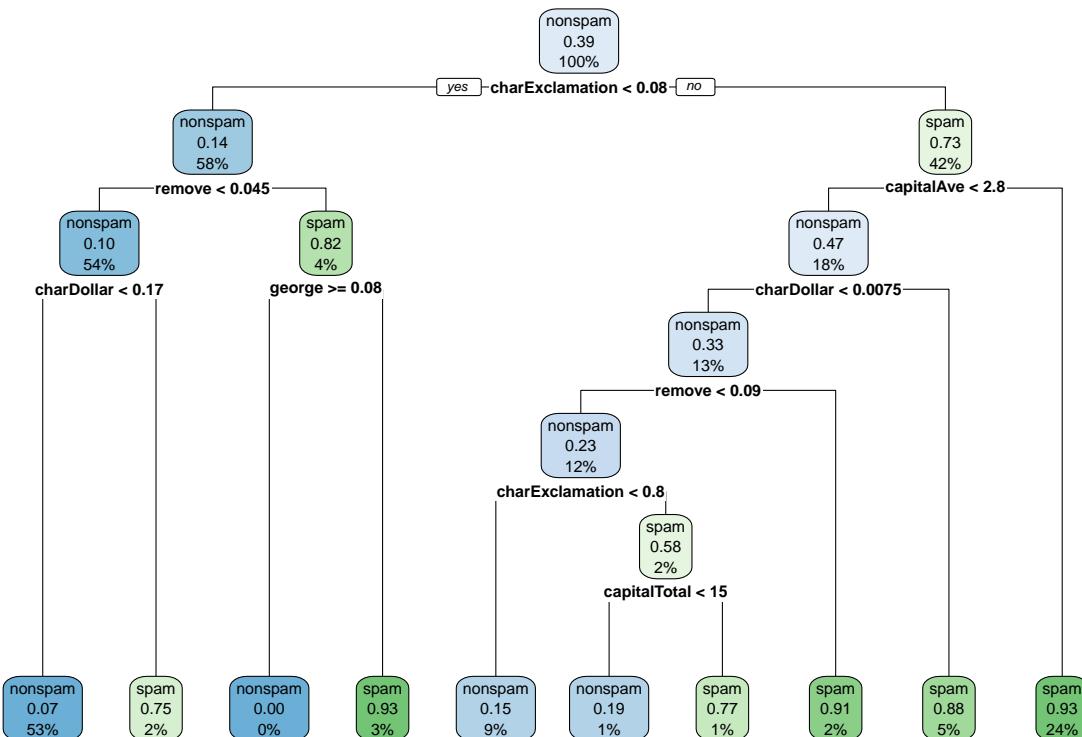
```
> m.step <- step(m.logit, direction="backward", trace=0)
```

4. A l'aide de la fonction **rpart** du package **rpart**, construire un arbre de régression (toujours sur les données d'apprentissage) pour répondre au problème. On utilisera les paramètres par défaut de la fonction.

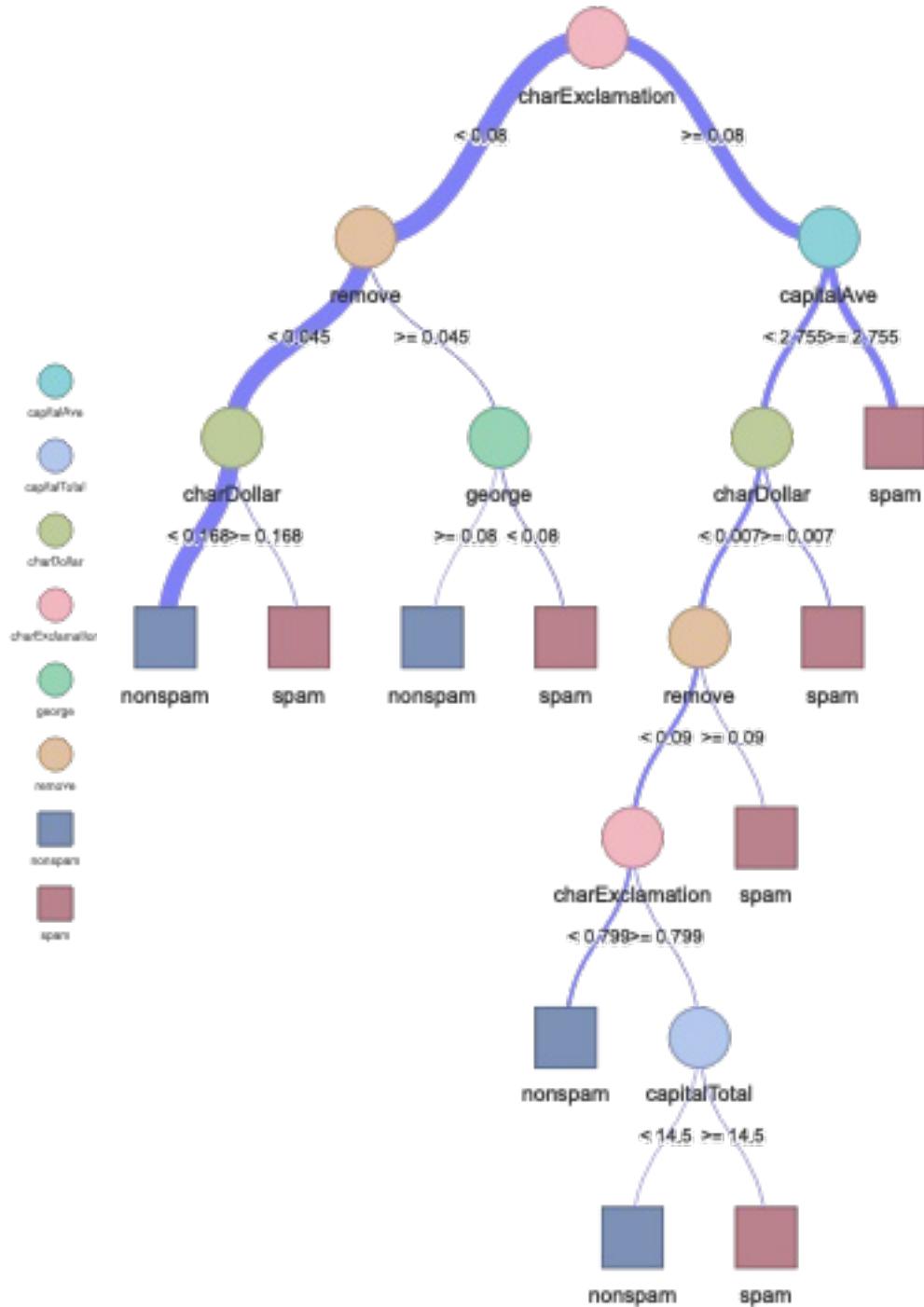
```
> library(rpart)
> arbre <- rpart(type~., data=dapp)
```

5. Visualiser l'arbre construit à l'aide des fonctions **rpart.plot** et **visTree** des packages **rpart.plot** et **visNetwork**

```
> library(rpart.plot)
> rpart.plot(arbre)
```



```
> library(visNetwork)
> visTree(arbre)
```



Export as png

6. Pour les 3 modèles construits (logistique, backward et arbre) calculer les prédictions de la variable `type` pour les individus de l'échantillon `dtest`. On pourra regrouper ces prévisions dans un data-frame à 3 colonnes.

```
> prev <- data.frame(
+   logit=predict(m.logit,newdata=dtest,type="response") %>% round() %>% recode_factor(`0`="nonspam"),
+   step=predict(m.step,newdata=dtest,type="response") %>% round() %>% recode_factor(`0`="nonspam"),
+   arbre=predict(arbre,newdata=dtest,type="class"))
```

7. Ajouter au data-frame précédent une colonne où on mettra les valeurs observées de la variable à expliquer.

```
> prev1 <- prev %>% mutate(obs=dtest$type)
```

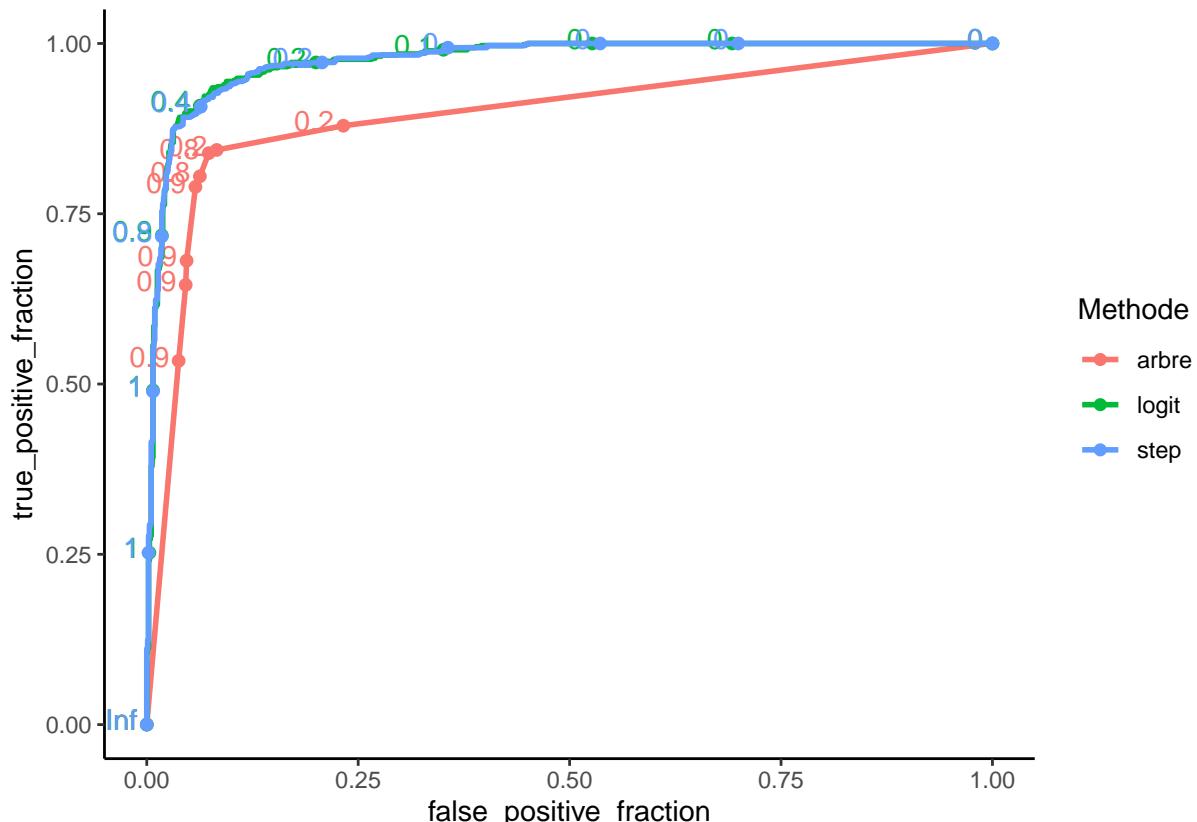
8. A l'aide de `summarize_at` calculer les erreurs de classification des 3 modèles.

```
> prev1 %>% summarize_at(1:3,~(mean(obs!=.))) %>% round(3)
  logit  step arbre
  1  0.07  0.074  0.109
```

9. Représenter les courbes **ROC** et calculer les **AUC**. On pourra consulter les pages 346 et 347 dans Cornillon et al. (2018) pour le tracé de courbes ROC sur R.

```
> score <- data.frame(
+   logit=predict(m.logit,newdata=dtest,type="response"),
+   step=predict(m.step,newdata=dtest,type="response"),
+   arbre=predict(arbre,newdata=dtest,type="prob")[,2]) %>%
+   mutate(obs=dtest$type) %>%
+   pivot_longer(-obs,names_to = "Methode",values_to="score")

> library(plotROC)
> ggplot(score)+aes(d=obs,m=score,color=Methode)+geom_roc()+theme_classic()
```



```

> score %>% group_by(Methode) %>%
+   summarize(AUC=as.numeric(pROC::auc(obs,score))) %>%
+   mutate(AUC=round(AUC,3)) %>%
+   arrange(desc(AUC))
# A tibble: 3 x 2
  Methode     AUC
  <chr>    <dbl>
1 logit    0.975
2 step     0.975
3 arbre    0.894

```

## Références

- Barnier, J. (2020). *Introduction à R et au tidyverse*.
- Cornillon, P., Guyader, A., Husson, F., Jégou, N., Josse, J., Klutchnikoff, N., Le Pennec, E., Matzner-Løber, E., Rouvière, L., and Thieurmel, B. (2018). *R pour la statistique et la science des données*. PUR.
- Wickham, A. and Grolemund, G. (2017). *R for Data Science*. O'Reilly.