

Introduction à R et à la modélisation statistique

Laurent Rouvière

2020-09-11

Table des matières

Présentation	2
1 Introduction	2
1.1 R Script	2
1.2 Packages	3
1.3 Markdown	3
2 Les objets R	8
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	13
2.7 Exercices complémentaires	14
3 Manipuler les données avec dplyr	15
3.1 Importer des données	15
3.2 Le package dplyr	17
4 Visualisation avec ggplot2	29
4.1 Fonctions graphiques conventionnelles	29
4.2 La grammaire ggplot2	38
4.3 Compléments	57
4.4 Quelques exercices supplémentaires	65
5 Faire des cartes interactives avec leaflet	70
5.1 Présentation de leaflet	70
5.2 Challenge : Visualisation des stations velib à Paris	77
6 Faire de la régression sur R	78
6.1 Modèle linéaire : fonctions lm et predict	79
6.2 Sélection de variables	80
6.3 Régression logistique et arbre	80
7 Estimation et intervalles de confiance	81
7.1 Générer des observations selon des lois de probabilités	81
7.2 Une étude numérique de la moyenne empirique	82
7.3 Intervalles de confiance	84

Présentation

Ce tutoriel présente une introduction au logiciel R. On pourra trouver :

- les supports de cours associés à ce tutoriel ainsi que les données utilisées à l'adresse suivante https://lrouviere.github.io/ml_lecture/ ;
- le tutoriel sans les correction à l'url https://lrouviere.github.io/TUTO_R/
- le tutoriel avec les corrigés (à certains moment) à l'url https://lrouviere.github.io/TUTO_R/correction/.

Il est recommandé d'utiliser **mozilla firefox** pour lire le tutoriel.

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...
- **Introduction au problème de l'estimation**, lois de probabilités, notions d'estimateurs, performance d'estimateurs, intervalles de confiance.

On pourra trouver des supports de cours ainsi que les données utilisées à l'adresse suivante 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.

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 spécifiées 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 visualier 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 prises 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
[5] 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  
[6] 0.5555556 0.6666667 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).

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 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)"
```

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

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

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,]   15   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 dataframe

```
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 |

2. Créer un vecteur qui contient la diagonale de **mat**.
3. Créer une matrice qui contient les 2 premières lignes de **mat**.
4. Créer une matrice qui contient les 2 dernières colonnes de **mat**.
5. Calculer le déterminant et l'inverse de **mat**.

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**.

2. Créer un sous jeu de données qui contient uniquement les iris de l'espèce `versicolor`. On appellera ce tableau `iris2`.
3. Ordonner les individus dans `iris2` par valeurs décroissantes de la variable `Sepal.Length` (on pourra utiliser la fonction `order`).
4. Calculer les valeurs moyennes de `Sepal.Length` pour chaque espèce.
5. Ajouter une variable (qu'on appellera `sum.Petal`) dans le dataframme `iris` qui contiendra la somme de `Petal.Length` et `Petal.Width`.

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")
```

2. Calculer les quartiles de chaque variables. On pourra faire un `apply` avec la fonction `quantile`.
3. Faire de même pour les déciles.

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

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
Length:20      Length:20      Min.   :-27.61
Class :character Class :character  1st Qu.:-27.55
```

```

Mode :character Mode :character Median :-27.49
                  Mean :-27.49
                  3rd Qu.:-27.45
                  Max. :-27.31

Longitude
Min. :152.9
1st Qu.:153.0
Median :153.0
Mean   :153.0
3rd Qu.:153.1
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
Length:20      Length:20      Min.   :-27.61
Class :character Class :character 1st Qu.:-27.55
Mode  :character Mode  :character Median :-27.49
                  Mean   :-27.49
                  3rd Qu.:-27.45
                  Max.   :-27.31

Longitude
Min.   :152.9
1st Qu.:153.0
Median :153.0
Mean   :153.0
3rd Qu.:153.1
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.
2. Importer **correctement** les données qui se trouvent dans le fichier *mydata.csv* (utiliser *sep*, *dec* et *row.names*).

3. Importer les données qui se trouvent dans le fichier *mydata2.csv*.
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.
5. Changer les **levels** de la variable **sex** en **woman** et **man** (on pourra utiliser la fonction **levels**).
 - 1ère façon :
 - 2ème façon avec **recode_factor**

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.
2. Assembler **df** avec **df3** en utilisant **full_join**.
3. Faire la même chose avec **inner_join**.
4. Expliquer les différences entre **full_join** et **inner_join**.

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 Acacia Ridge Lei~ 1391 Beaudesert Road~ -27.6      153.
2 Bellbowrie Pool  Sugarwood Street, Be~ -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 ~
2 Swimming pool Bellbowrie Pool is located at the address ~
3 Swimming pool Carole Park is located at the address Cnr ~
4 Swimming pool Centenary Pool (inner City) is located at ~
5 Swimming pool Chermside Pool is located at the address 3~
6 Swimming pool Colmslie Pool (Morningside) is located at ~
7 Swimming pool Spring Hill Baths (inner City) is located ~
```

```

8 Swimming pool Dunlop Park Pool (Corinda) is located at t~  

9 Swimming pool Fortitude Valley Pool is located at the ad~  

10 Swimming pool Hibiscus Sports Complex (upper MtGravatt) ~  

11 Swimming pool Ithaca Pool ( Paddington) is located at th~  

12 Swimming pool Jindalee Pool is located at the address 11~  

13 Swimming pool Manly Pool is located at the address 1 Fai~  

14 Swimming pool Mt Gravatt East Aquatic Centre is located ~  

15 Swimming pool Musgrave Park Pool (South Brisbane) is loc~  

16 Swimming pool Newmarket Pool is located at the address 7~  

17 Swimming pool Runcorn Pool is located at the address 37 ~  

18 Swimming pool Sandgate Pool is located at the address 23~  

19 Swimming pool Langlands Parks Pool (Stones Corner) is lo~  

20 Swimming pool Yeronga Park Pool is located at the addres~
```

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 R~ 1391 Beau~ -27.6      153. Swimming ~ 125.
2 Bellbowr~ Sugarwood~ -27.6      153. Swimming ~ 125.
3 Carole P~ Cnr Bound~ -27.6      153. Swimming ~ 125.
4 Centenar~ 400 Grego~ -27.5      153. Swimming ~ 126.
5 Chermsid~ 375 Hamil~ -27.4      153. Swimming ~ 126.
6 Colmslie~ 400 Lytto~ -27.5      153. Swimming ~ 126.
7 Spring H~ 14 Torrin~ -27.5      153. Swimming ~ 126.
8 Dunlop P~ 794 Oxley~ -27.5      153. Swimming ~ 125.
9 Fortitud~ 432 Wickh~ -27.5      153. Swimming ~ 126.
10 Hibiscus~ 90 Klumpp~ -27.6      153. Swimming ~ 126.
11 Ithaca P~ 131 Caxto~ -27.5      153. Swimming ~ 126.
12 Jindalee~ 11 Yallam~ -27.5      153. Swimming ~ 125.
13 Manly Po~ 1 Fairlea~ -27.5      153. Swimming ~ 126.
14 Mt Grava~ Cnr wecke~ -27.5      153. Swimming ~ 126.
15 Musgrave~ 100 Edmon~ -27.5      153. Swimming ~ 126.
16 Newmarke~ 71 Alders~ -27.4      153. Swimming ~ 126.
17 Runcorn ~ 37 Bonemi~ -27.6      153. Swimming ~ 125.
18 Sandgate~ 231 Flind~ -27.3      153. Swimming ~ 126.
19 Langland~ 5 Panitya~ -27.5      153. Swimming ~ 126.
20 Yeronga ~ 81 School~ -27.5      153. Swimming ~ 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, ~ -27.4    153.
2 Colmslie Pool (Mo~ 400 Lytton Road, Mo~ -27.5    153.
3 Spring Hill Baths~ 14 Torrington Street~ -27.5    153.
4 Dunlop Park Pool ~ 794 Oxley Road, Cor~ -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, ~ -27.6    153.
2 Carole Park   Cnr Boundary Road ~ -27.6    153.
3 Jindalee Pool 11 Yallambee Road, ~ -27.5    153.
4 Dunlop Park Pool ~ 794 Oxley Road, Co~ -27.5    153.
5 Newmarket Pool 71 Alderson Stret,~ -27.4    153.
6 Ithaca Pool ( Pad~ 131 Caxton Street,~ -27.5    153.
7 Musgrave Park Poo~ 100 Edmonstone Str~ -27.5    153.
8 Yeronga Park Pool 81 School Road, Ye~ -27.5    153.
9 Spring Hill Baths~ 14 Torrington Stre~ -27.5    153.
10 Centenary Pool (i~ 400 Gregory Terrac~ -27.5    153.
11 Acacia Ridge Leis~ 1391 Beaudesert Ro~ -27.6    153.
12 Chermside Pool   375 Hamilton Road, ~ -27.4    153.
13 Fortitude Valley ~ 432 Wickham Street~ -27.5    153.
14 Langlands Parks P~ 5 Panitya Street, ~ -27.5    153.
15 Sandgate Pool    231 Flinders Parad~ -27.3    153.
16 Hibiscus Sports C~ 90 Klumpp Road, Up~ -27.6    153.
17 Runcorn Pool     37 Bonemill Road, ~ -27.6    153.
18 Colmslie Pool (Mo~ 400 Lytton Road, M~ -27.5    153.
19 Mt Gravatt East A~ Cnr wecker Road an~ -27.5    153.
20 Manly Pool       1 Fairlead Crescen~ -27.5    153.

```

ou

```

arrange(piscines, desc(Longitude))
# A tibble: 20 x 4
  Name          Address      Latitude Longitude
  <chr>        <chr>       <dbl>     <dbl>
1 Manly Pool     1 Fairlead Crescen~ -27.5    153.
2 Mt Gravatt East A~ Cnr wecker Road an~ -27.5    153.
3 Colmslie Pool (Mo~ 400 Lytton Road, M~ -27.5    153.
4 Runcorn Pool     37 Bonemill Road, ~ -27.6    153.
5 Hibiscus Sports C~ 90 Klumpp Road, Up~ -27.6    153.

```

| | | | | |
|----|-------------------|----------------------|-------|------|
| 6 | Sandgate Pool | 231 Flinders Parade | -27.3 | 153. |
| 7 | Langlands Parks P | 5 Panitya Street, ~ | -27.5 | 153. |
| 8 | Fortitude Valley | 432 Wickham Street~ | -27.5 | 153. |
| 9 | Chermside Pool | 375 Hamilton Road, ~ | -27.4 | 153. |
| 10 | Acacia Ridge Leis | 1391 Beaudesert Ro~ | -27.6 | 153. |
| 11 | Centenary Pool (i | 400 Gregory Terrac~ | -27.5 | 153. |
| 12 | Spring Hill Baths | 14 Torrington Stre~ | -27.5 | 153. |
| 13 | Yeronga Park Pool | 81 School Road, Ye~ | -27.5 | 153. |
| 14 | Musgrave Park Poo | 100 Edmonstone Str~ | -27.5 | 153. |
| 15 | Ithaca Pool (Pad | 131 Caxton Street, ~ | -27.5 | 153. |
| 16 | Newmarket Pool | 71 Alderson Stret, ~ | -27.4 | 153. |
| 17 | Dunlop Park Pool | 794 Oxley Road, Co~ | -27.5 | 153. |
| 18 | Jindalee Pool | 11 Yallambee Road, ~ | -27.5 | 153. |
| 19 | Carole Park | Cnr Boundary Road ~ | -27.6 | 153. |
| 20 | Bellbowrie Pool | Sugarwood Street, ~ | -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 `summarize` (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**.
2. Construire une table qui contient uniquement les iris d'espèce **versicolor** ou **virginica** (on pourra utiliser le symbole **|** pour la condition **ou**).
3. Calculer le nombre d'iris de l'espèce **setosa** en utilisant **summarise**.
4. Calculer la moyenne de la variable **Petal.Width** pour les iris de l'espèce **versicolor**.
5. Ajouter dans le jeu de données la variable **Sum_Petal** qui correspond à la somme de **Petal.Width** et **Sepal.Width**.
6. Calculer la moyenne et la variance de la variable **Sepal.Length** pour chaque espèce (on pourra utiliser **group_by**).

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.
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()`.
3. Ajouter une variable `ActualGroundTime` qui correspond à `ActualElapsedTime` moins `AirTime`.
4. Ajouter la variable `AverageSpeed` ($=\text{Distance}/\text{AirTime}$) et ordonner la table selon les valeurs décroissantes de cette variable.
5. Sélectionner les vols à destination de `JFK`.
6. Calculer le nombre de vols à destination de `JFK`.
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.
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...).
9. Calculer pour chaque compagnie :
 - le nombre total de vols ;
 - La valeur moyenne de `AirTime`.
10. Ordonner les compagnies en fonction des retards moyens au départ.

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.
2. Afficher le nom des adversaires de `Roger Federer`.
3. Afficher le nom des demi-finalistes (ceux qui ont atteint le 6ème tour).
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`).
5. Combien y a t-il eu d'aces par match en moyenne ?
6. Combien y a t-il eu d'aces par match en moyenne à chaque tour ?
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) ?
8. Importer les données pour le tournoi masculin de Wimbledon 2013.
9. Concaténer les tables en ajoutant une variable permettant d'identifier le tournoi. On pourra utiliser `bind_rows` avec l'option `.id`.
10. Afficher les matchs de Federer pour chaque tournoi.

ou

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

ou pour une présentation plus synthétique

3.2.5 Compléments : Tidy data avec **tidy**

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

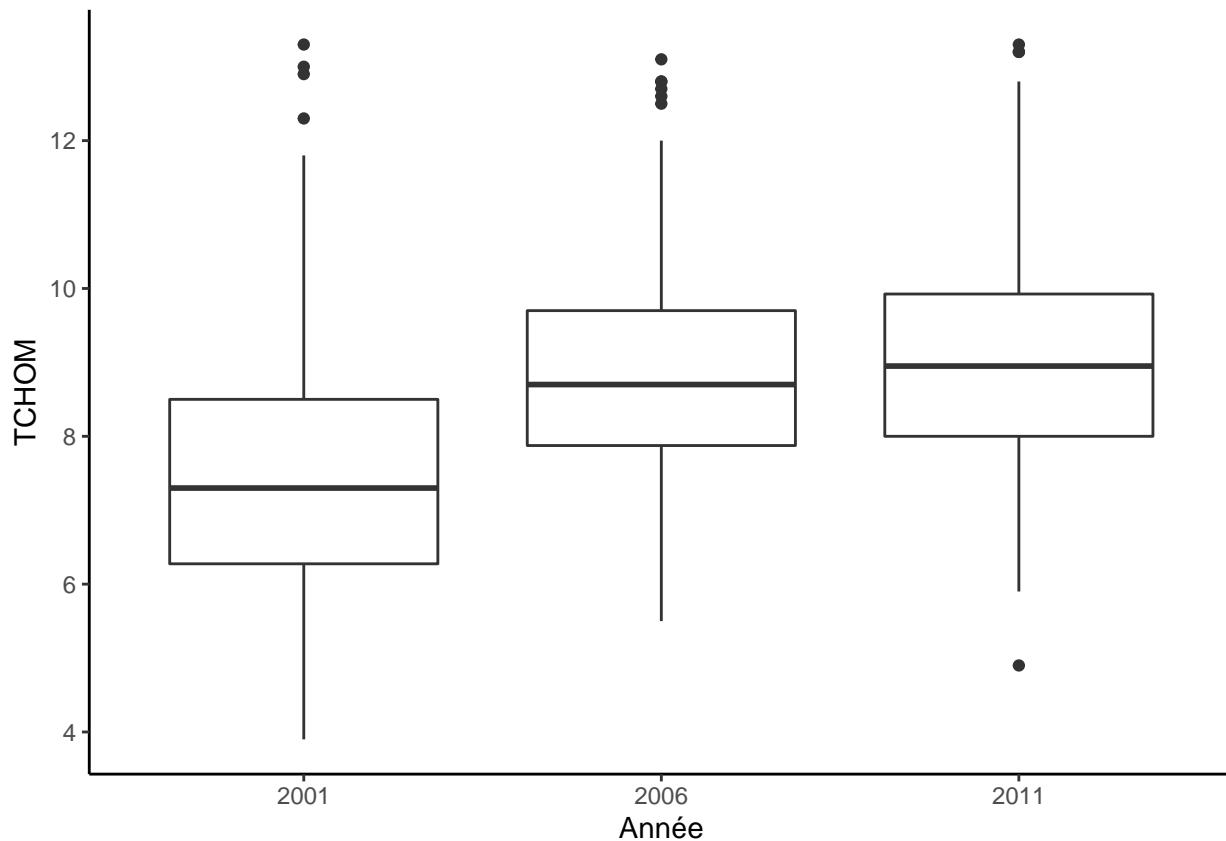
- le département
- l'année
- le taux de chômage

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 aisément 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 **tidyR** 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

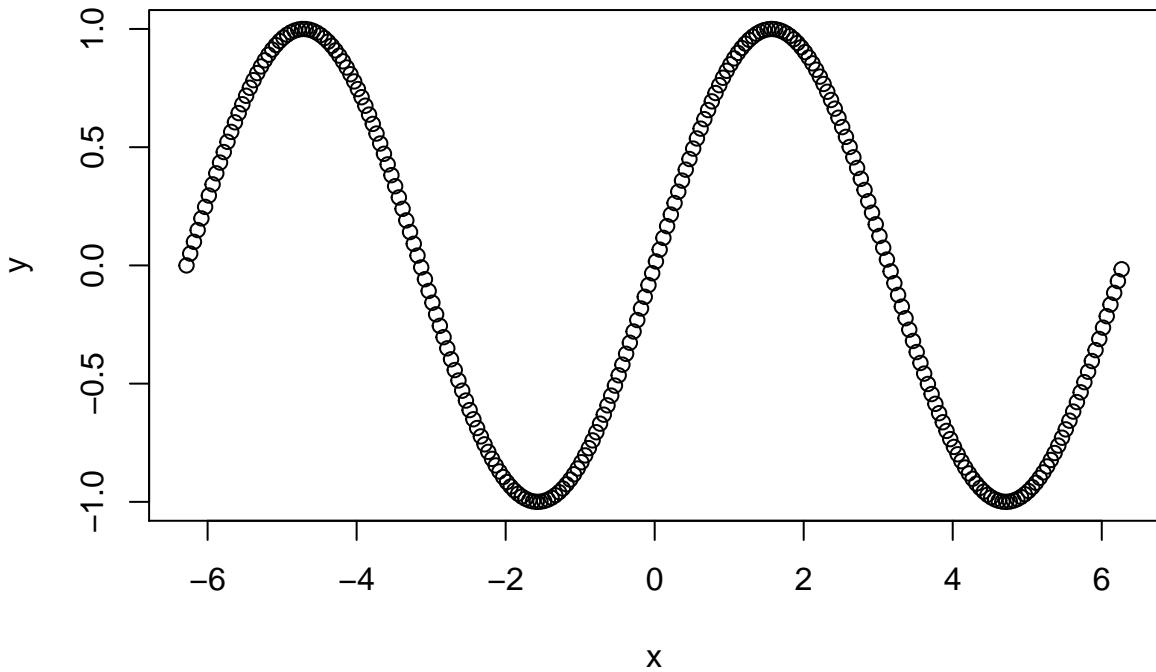
Pour commencer il est intéressant d'examiner quelques exemples de représentations graphiques construits avec **R**. On peut les obtenir à l'aide de la fonction **demo**.

```
demo(graphics)
```

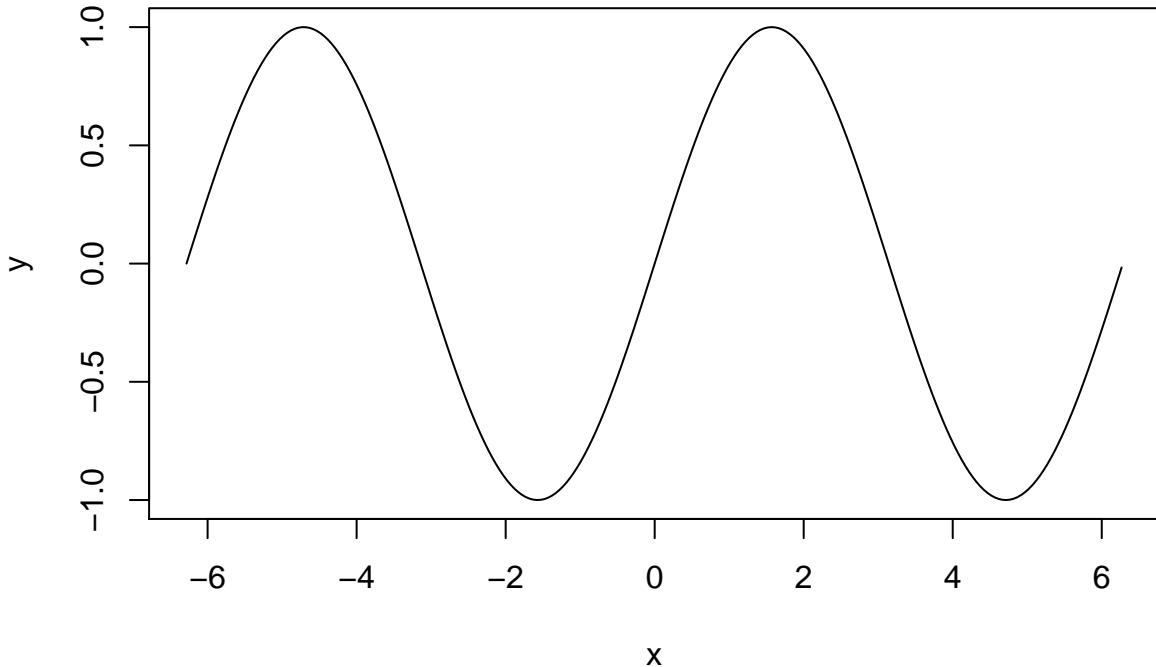
4.1.1 La fonction plot

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
```

```

Min. : 42.00  Min. :11.30  Min. :14.00
1st Qu.: 70.75 1st Qu.:16.20 1st Qu.:18.60
Median : 81.50 Median :17.80 Median :20.55
Mean   : 90.30 Mean  :18.36 Mean  :21.53
3rd Qu.:106.00 3rd Qu.:19.93 3rd Qu.:23.55
Max.   :166.00 Max.  :27.00 Max.  :33.50

      T15          Ne9          Ne12
Min. :14.90      Min. :0.000      Min. :0.000
1st Qu.:19.27    1st Qu.:3.000    1st Qu.:4.000
Median :22.05    Median :6.000    Median :5.000
Mean   :22.63    Mean  :4.929    Mean  :5.018
3rd Qu.:25.40    3rd Qu.:7.000    3rd Qu.:7.000
Max.   :35.50    Max.  :8.000    Max.  :8.000

      Ne15          Vx9          Vx12
Min. :0.00      Min. :-7.8785   Min. :-7.878
1st Qu.:3.00    1st Qu.:-3.2765  1st Qu.:-3.565
Median :5.00    Median :-0.8660   Median :-1.879
Mean   :4.83    Mean  :-1.2143   Mean  :-1.611
3rd Qu.:7.00    3rd Qu.: 0.6946   3rd Qu.: 0.000
Max.   :8.00    Max.  : 5.1962   Max.  : 6.578

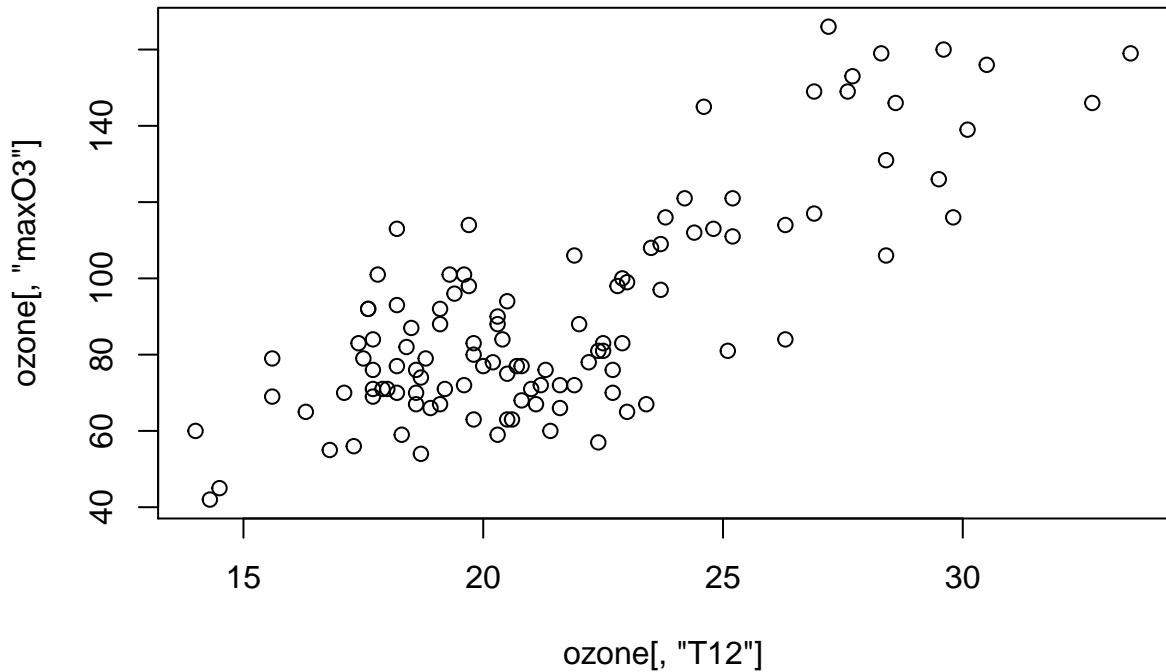
      Vx15          max03v        vent
Min. : -9.000   Min. : 42.00   Length:112
1st Qu.: -3.939 1st Qu.: 71.00   Class :character
Median : -1.550 Median : 82.50   Mode  :character
Mean   : -1.691 Mean  : 90.57
3rd Qu.:  0.000 3rd Qu.:106.00
Max.   :  5.000  Max.  :166.00

      pluie
Length:112
Class :character
Mode  :character

```

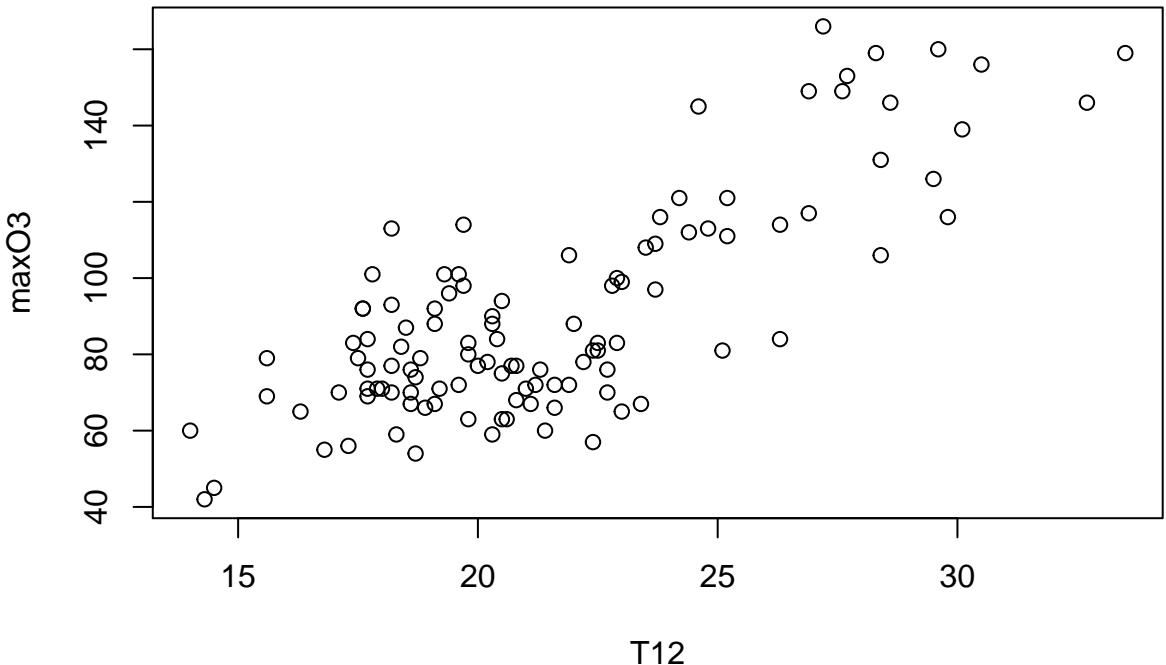
On visualise tout d'abord 2 variables quantitatives à l'aide d'un nuage de points : la concentration en ozone maximale **maxO3** en fonction de la température à 12h **T12**.

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



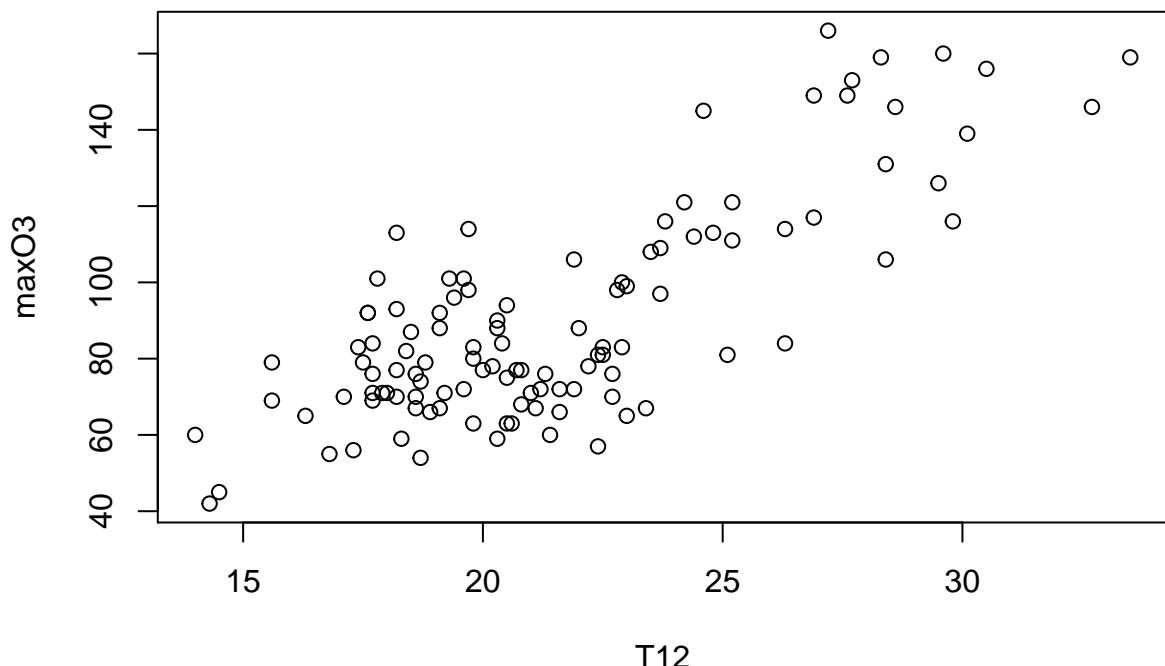
Comme les deux variables appartiennent au même jeu de données, on peut obtenir la même représentation à l'aide d'une syntaxe plus claire qui ajoutent automatiquement les noms des variables sur les axes :

```
plot(maxO3~T12, data=ozone)
```



Une autre façon de faire (moins naturelle) :

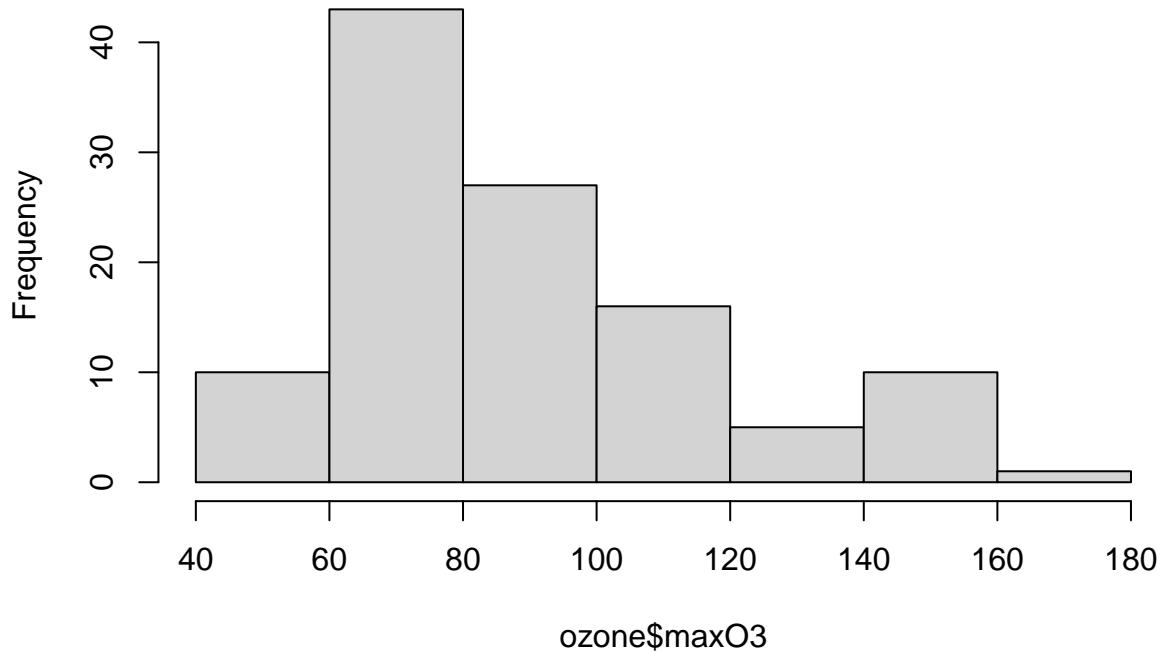
```
plot(ozone[, "T12"], ozone[, "maxO3"], xlab="T12", ylab="maxO3")
```



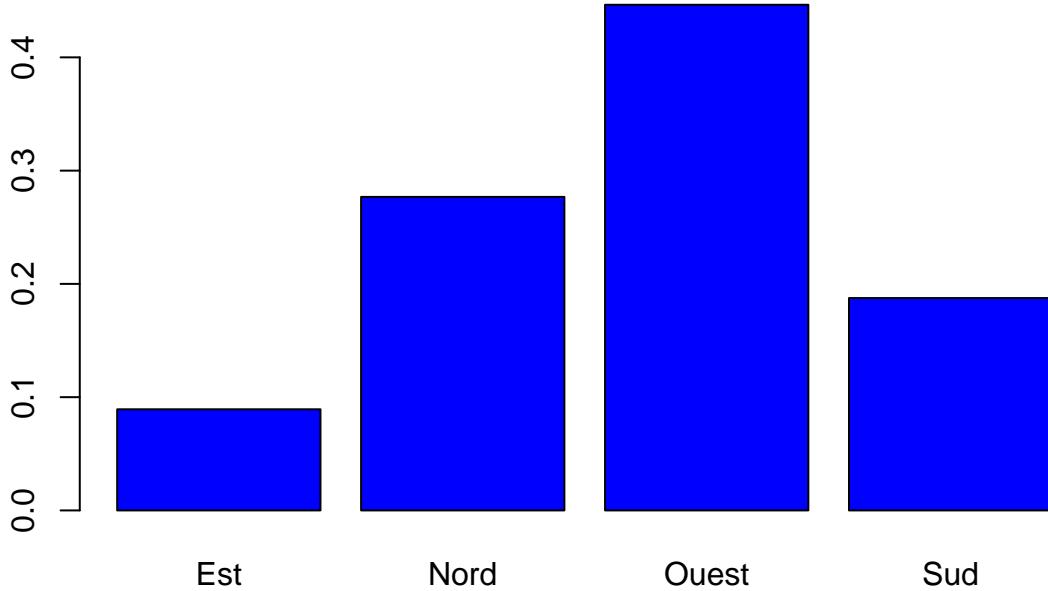
Il existe des fonctions spécifiques pour chaque type de graphs, par exemple **histogram**, **barplot** et **boxplot** :

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

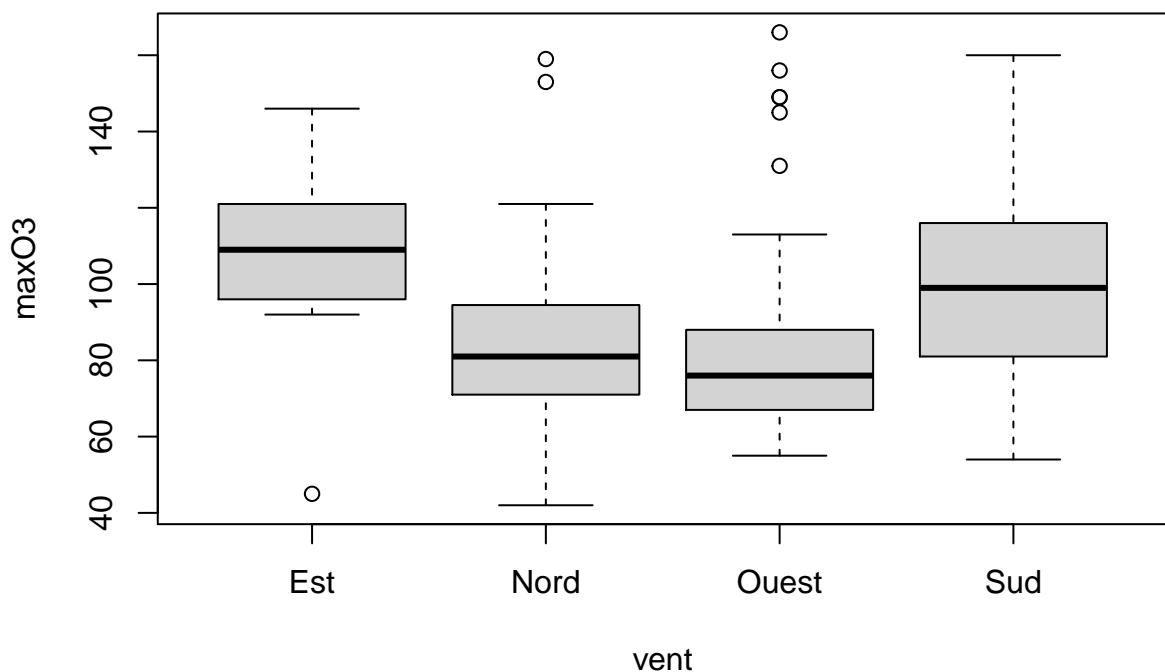
Histogram



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



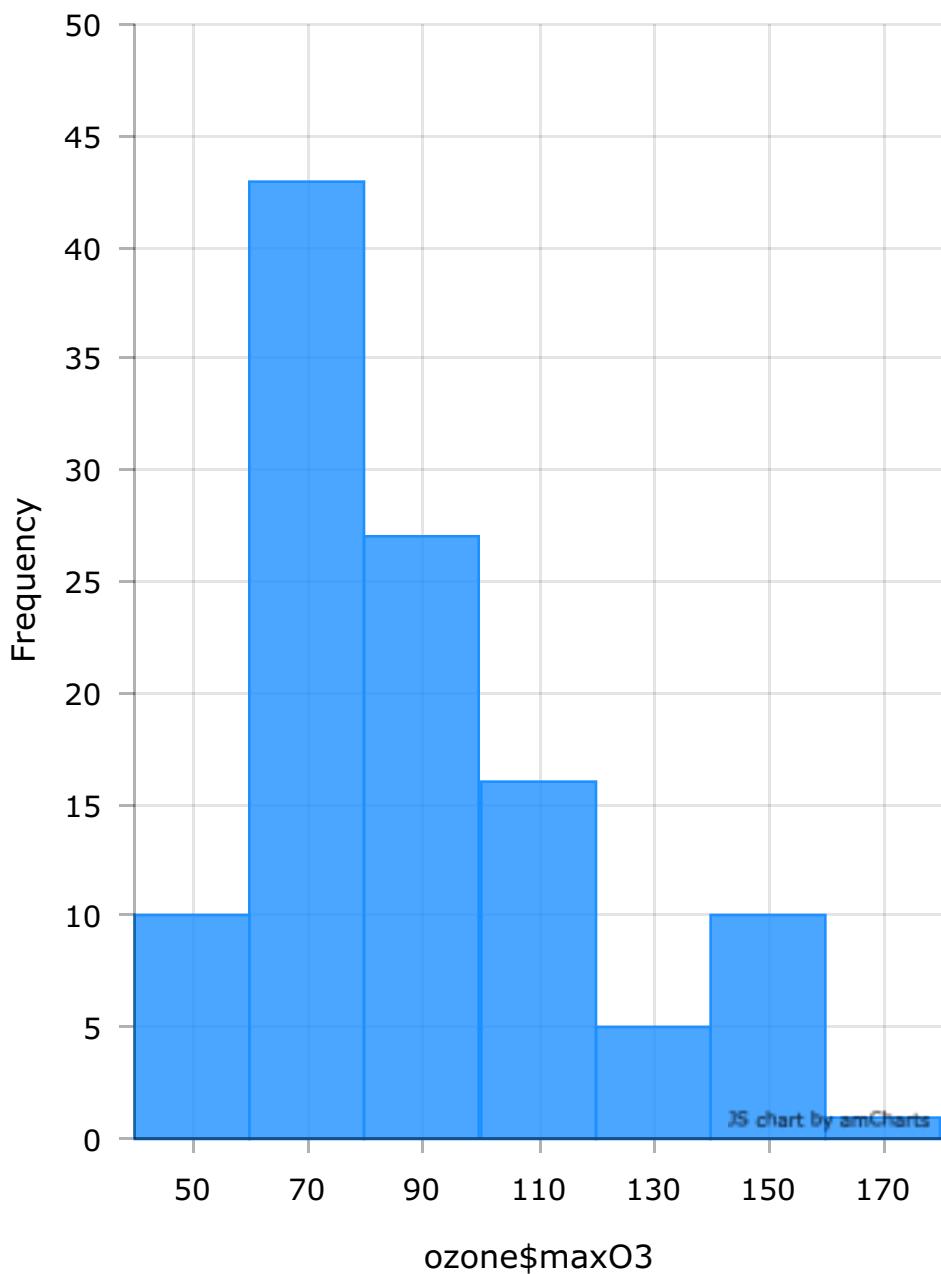
```
boxplot(max03~vent, data=ozone)
```



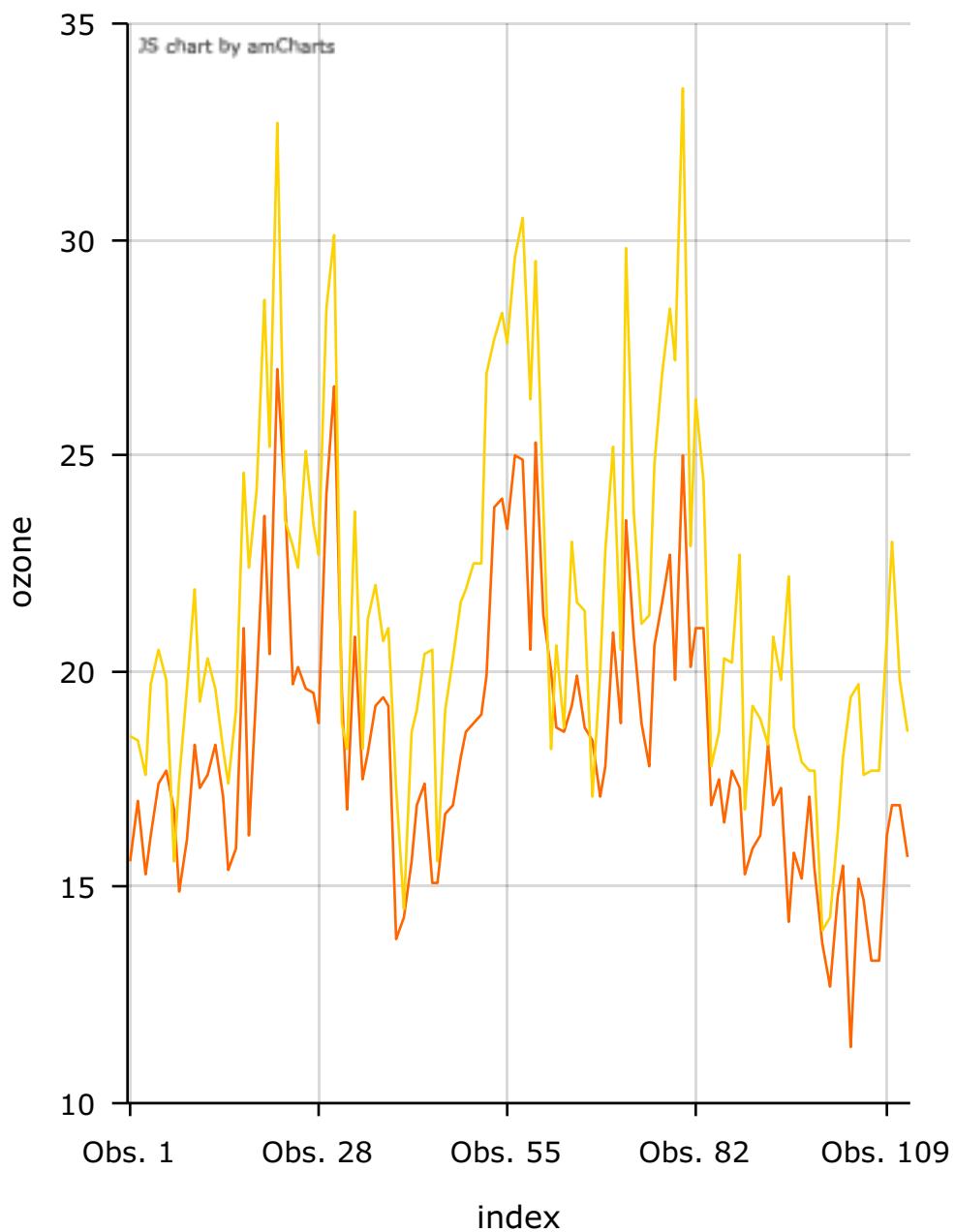
4.1.2 Graphes interactifs avec rAmCharts

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 :

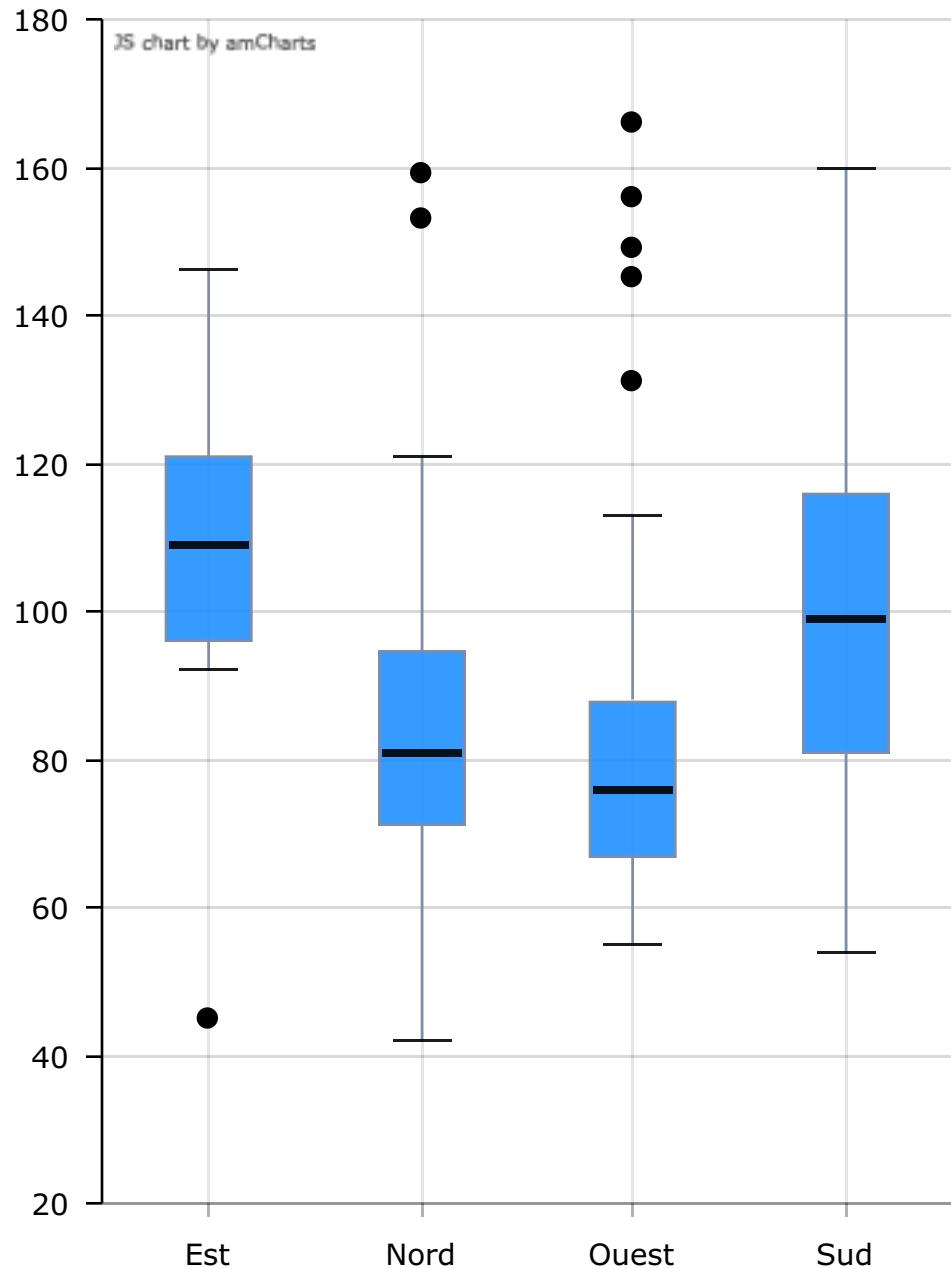
```
library(rAmCharts)
amHist(ozone$max03)
```



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



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



Exercice 4.1 (Premier graphe).

1. Tracer la fonction **sinus** entre 0 et 2π .
2. A l'aide de la fonction **title** ajouter le titre **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**).

Exercice 4.3 (Tâches solaires).

1. Importer la série **taches_solaires.csv** qui donne, date par date, un nombre de taches solaires observées.
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.
3. Utiliser les levels suivants pour le facteur **periode**.

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

4. Expliquer la sortie de la fonction
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 taches en utilisant une couleur différente pour chaque période.

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**)
2. 2 graphes sur la deuxième colonne (histogramme de **T12** et boxplot de **maxO3**).

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
Min.   :0.2000   Fair     :158    D: 640    SI1    :1189
1st Qu.:0.4000   Good    :455    E: 916    VS2    :1157
Median :0.7000   Very Good:1094   F: 900    SI2    : 876
Mean   :0.7969   Premium :1280    G:1018    VS1    : 738
3rd Qu.:1.0400   Ideal   :2013    H: 775    VVS2   : 470
Max.   :4.1300                    I: 481    VVS1   : 326
                           J: 270    (Other) : 244

      depth          table        price
Min.   :43.00   Min.   :49.00   Min.   : 365
1st Qu.:61.10   1st Qu.:56.00   1st Qu.: 945
Median :61.80   Median :57.00   Median :2376
Mean   :61.76   Mean   :57.43   Mean   :3917
3rd Qu.:62.50   3rd Qu.:59.00   3rd Qu.:5294
Max.   :71.60   Max.   :95.00   Max.   :18757

      x              y              z
Min.   : 0.000   Min.   :3.720   Min.   :0.000
1st Qu.: 4.720   1st Qu.:4.720   1st Qu.:2.920
Median : 5.690   Median :5.700   Median :3.520
Mean   : 5.728   Mean   :5.731   Mean   :3.538
```

```
3rd Qu.: 6.530    3rd Qu.: 6.520    3rd Qu.: 4.030  
Max.     :10.000    Max.     :9.850    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 :

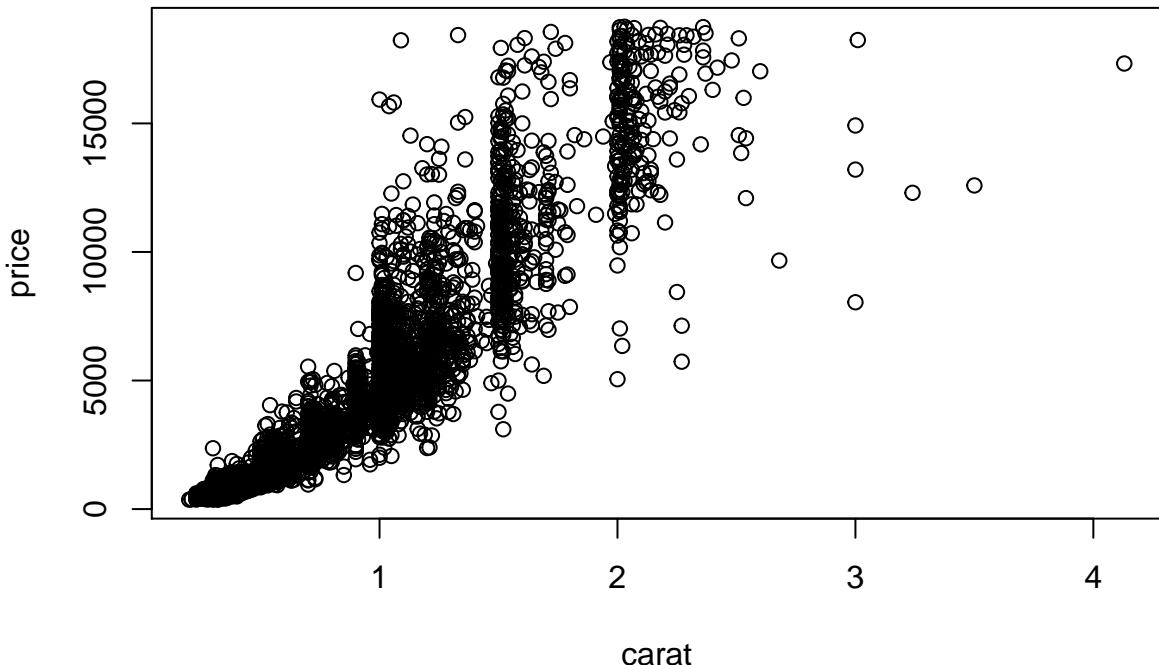
- les données
- les variables que l'on souhaite représenter
- le type de représentation (nuage de points, boxplot...).

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)
```

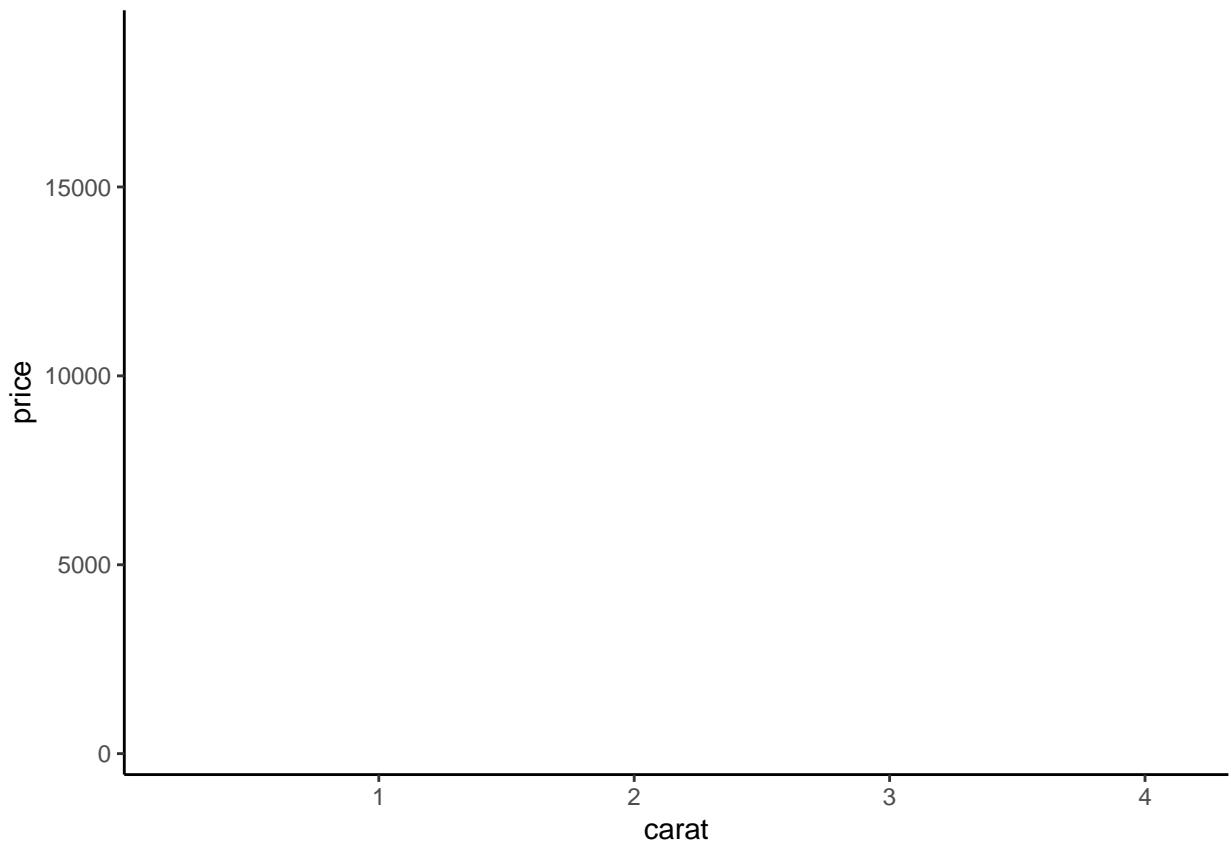


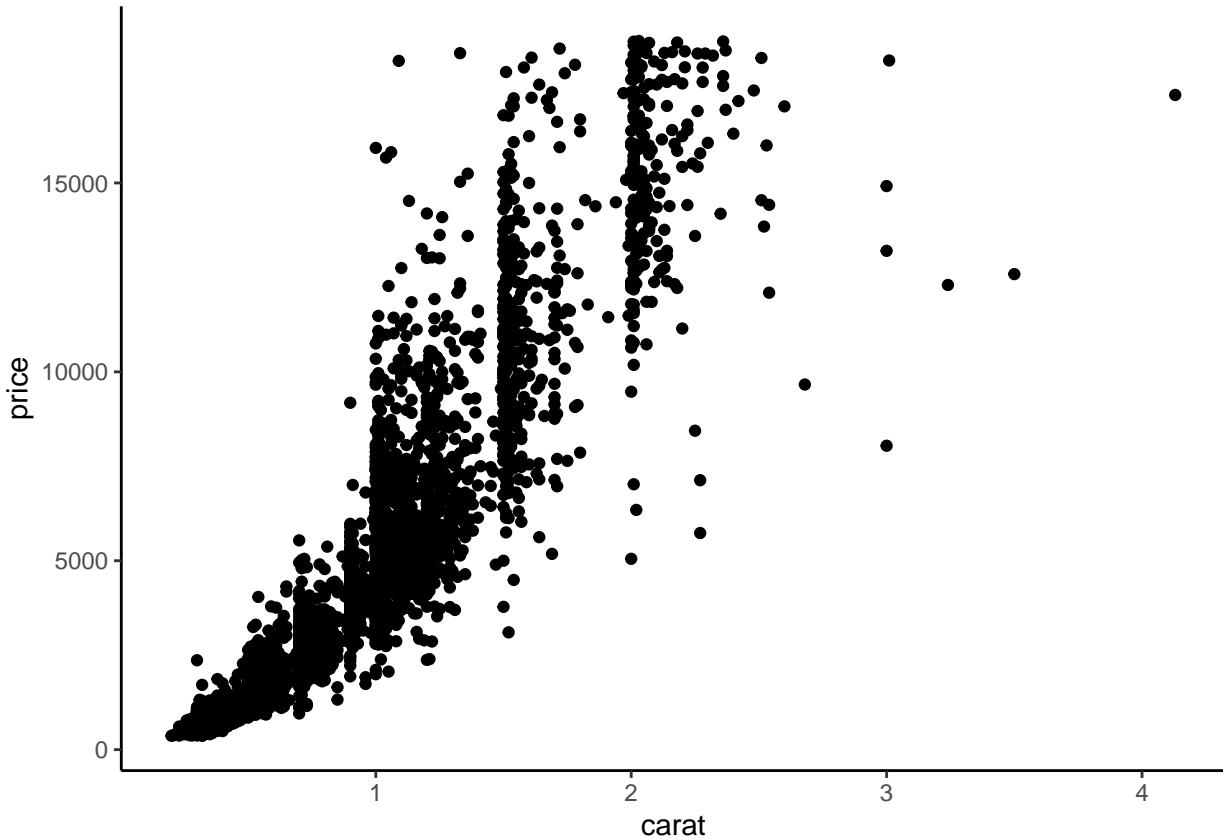
Avec **ggplot**, on va faire

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



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





Exercice 4.5 (Permiers 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`).

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)
```

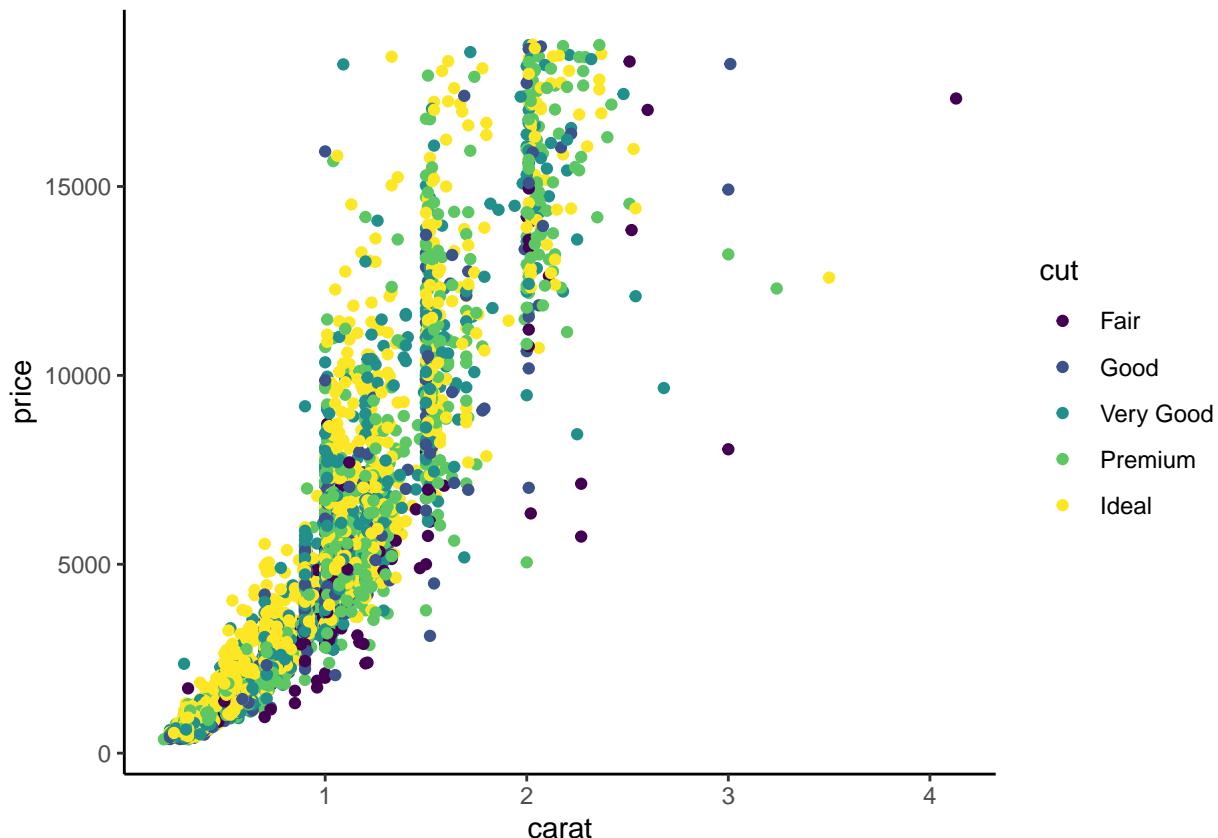
Les variables peuvent également être utilisées pour colorier des points ou des barres, définir des tailles...
Dans ce cas on pourra renseigner les arguments **color**, **size**, **fill** dans la fonction **aes**. Par exemple

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

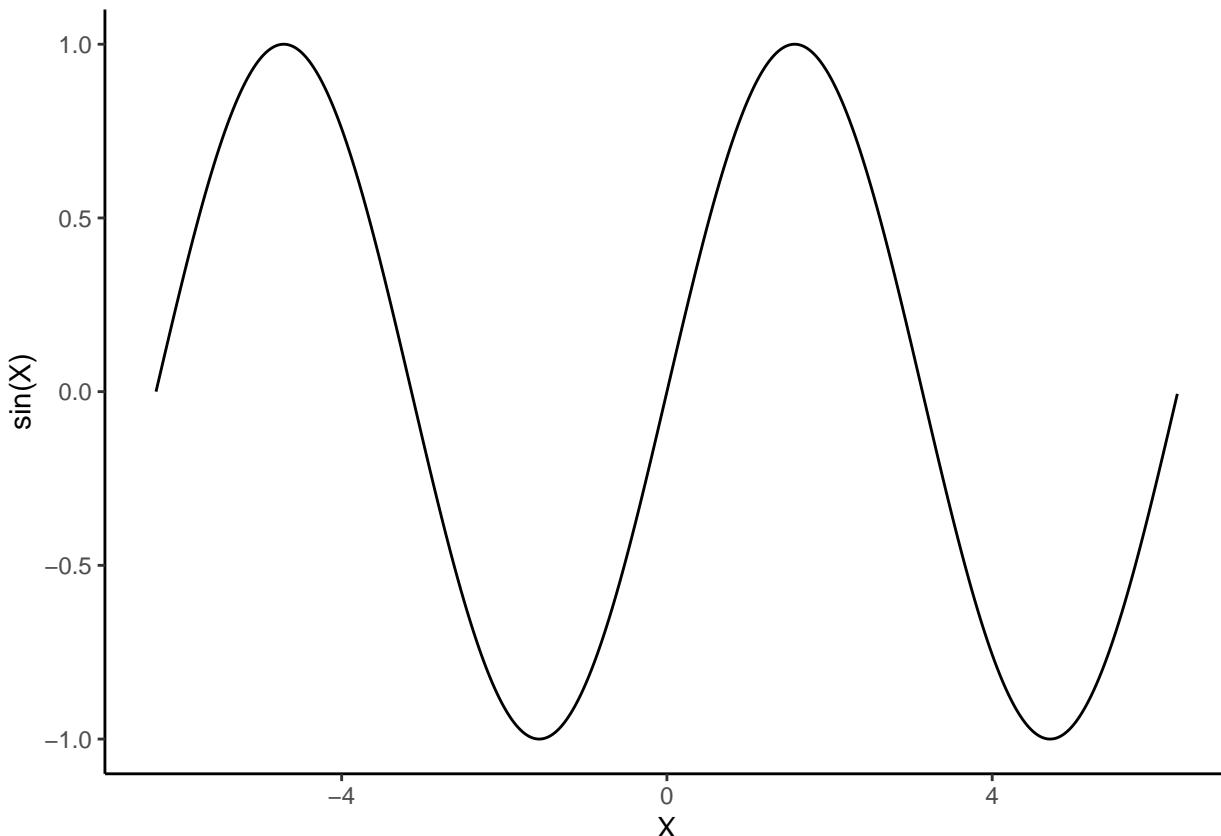
Exercice 4.6 (Diagrammes en barres).

1. Tracer le diagramme en barres de la variable **cut** avec des barres bleues.
2. 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.
3. Tracer le diagramme en barres de la variable **cut** avec une couleur pour chaque modalité que vous choisirez (et sans légende).

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()
```

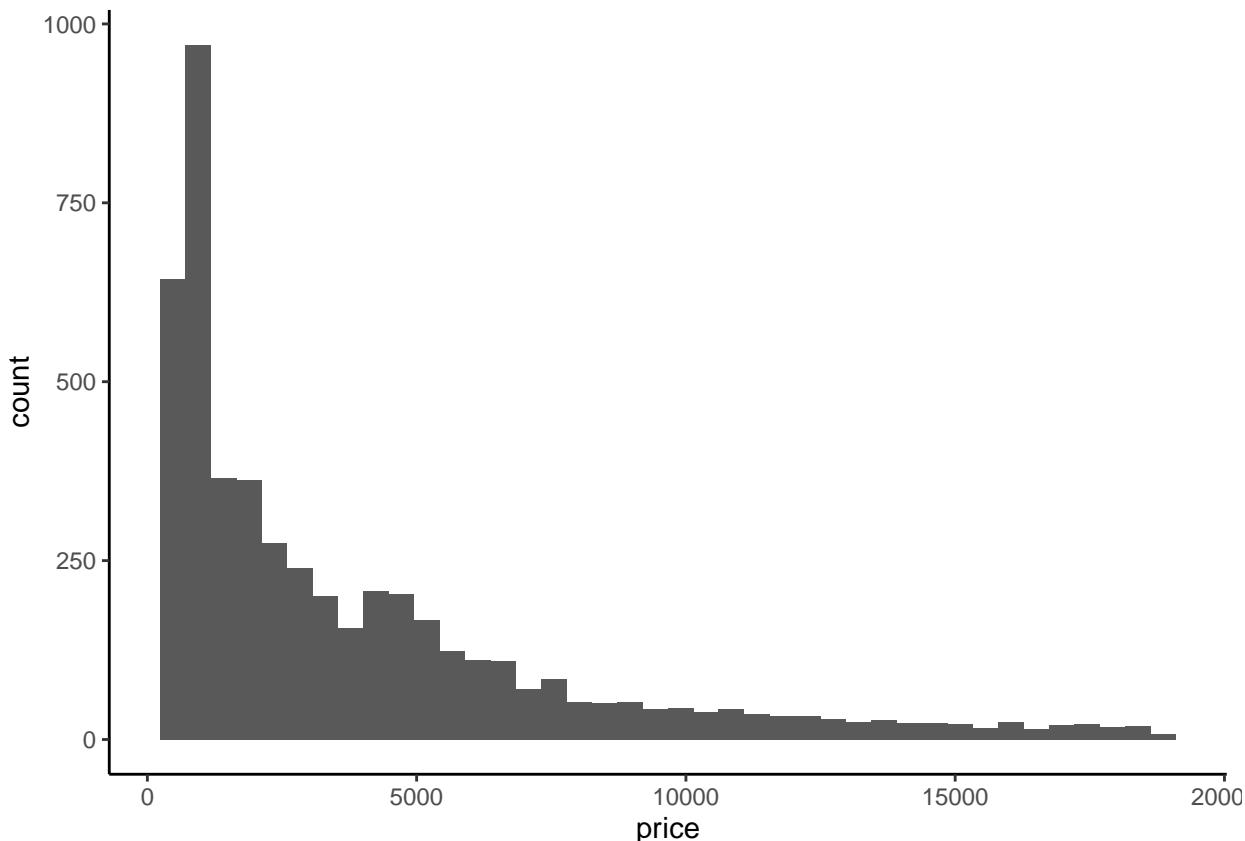


La transformation est spécifiée dans la fonction **aes**. Pour des transformations plus complexes, nous devons utiliser des **statistics**. Une fonction **stat** permet de définir des nouvelles variables à partir du jeu de données initial, il est ensuite possible de représenter ces nouvelles variables. 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.

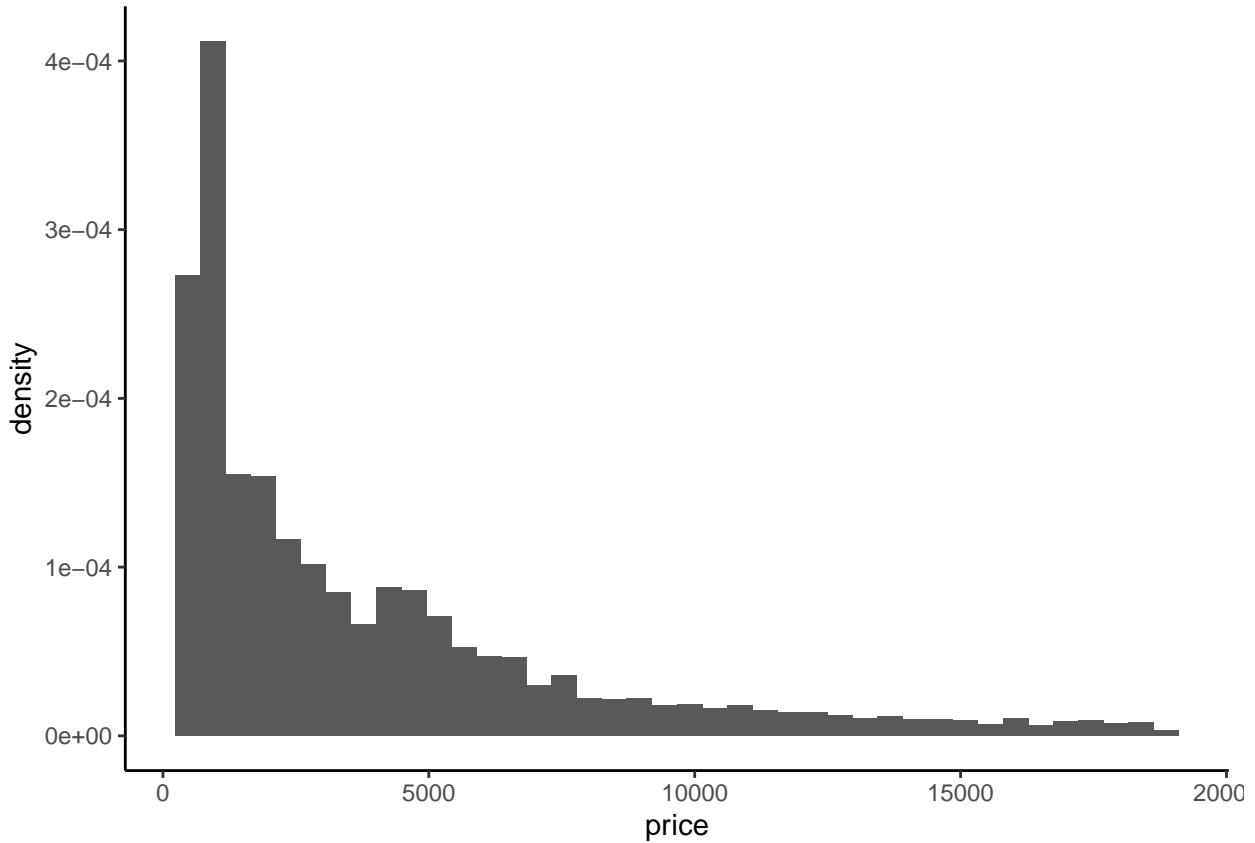
Par défaut *geom_histogram* fait appel à cette fonction **stat_bin** et représente sur l'axe *y* le nombre d'observations dans chaque classe (la variable **count**).

```
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.

Exercice 4.8 (Lissage).

1. Représenter le lissage non linéaire de la variable `price` contre la variable `carat` à l'aide de `geom_smooth` puis de `stat_smooth`.
2. Même question mais avec une ligne en pointillés à la place d'un trait plein.

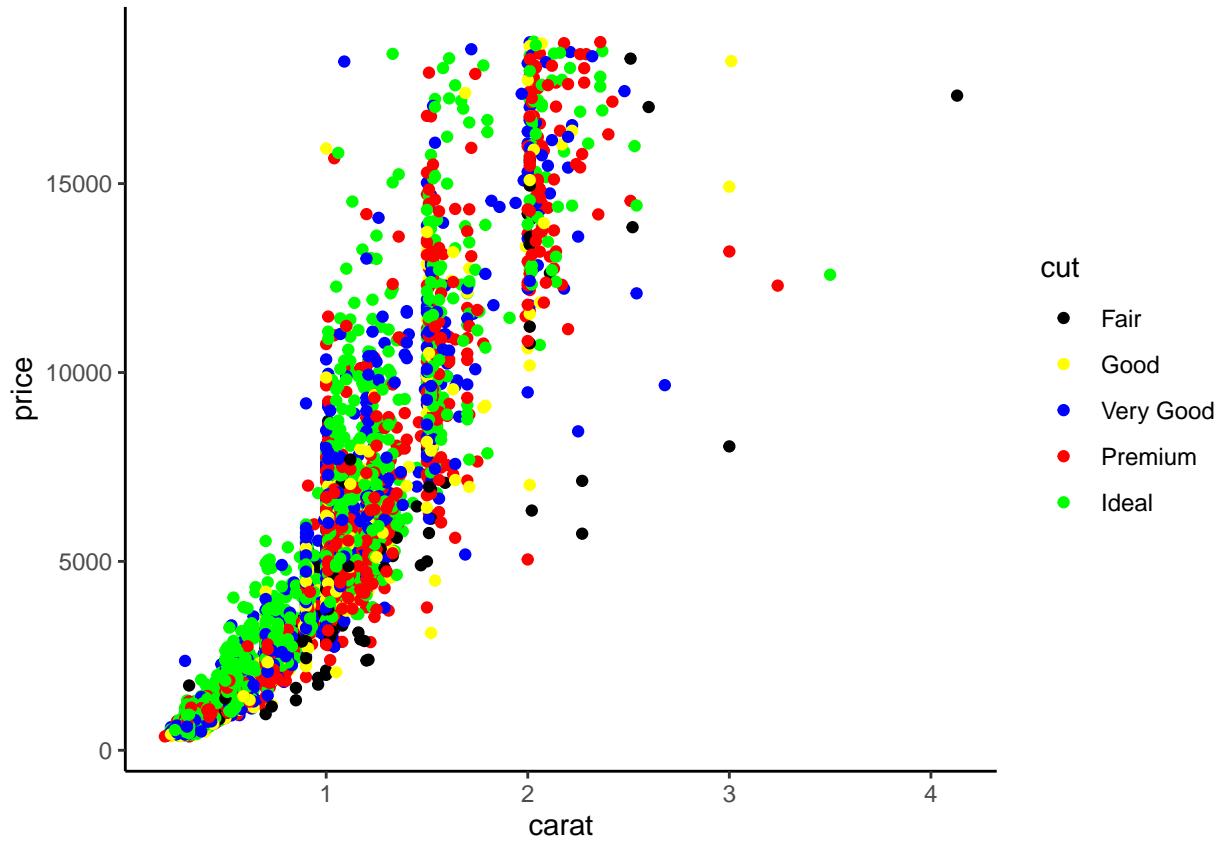
4.2.5 Scales

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

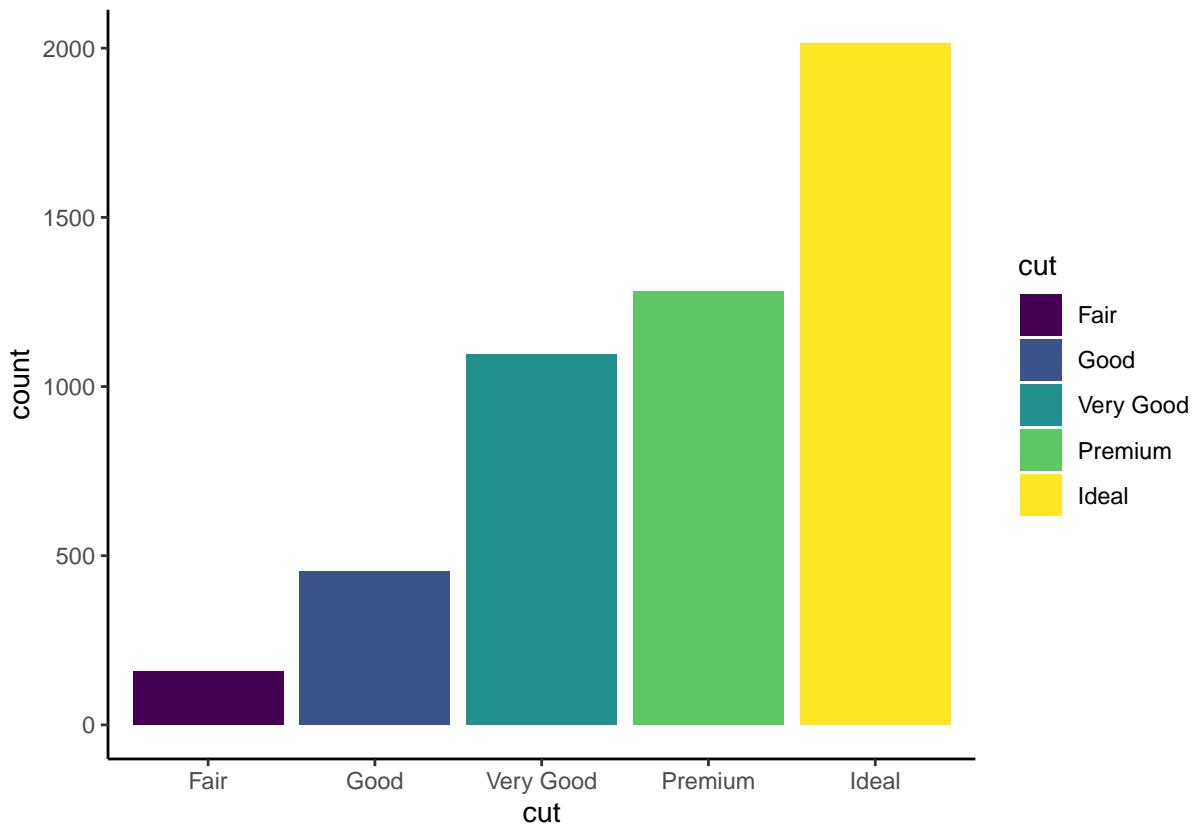
| aes | Discret | Continu |
|-------------------------|---------|-----------|
| Couleur (color et fill) | brewer | gradient |
| - | grey | gradient2 |

| aes | Discret | Continu |
|-------------------|----------|------------|
| - | 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 :

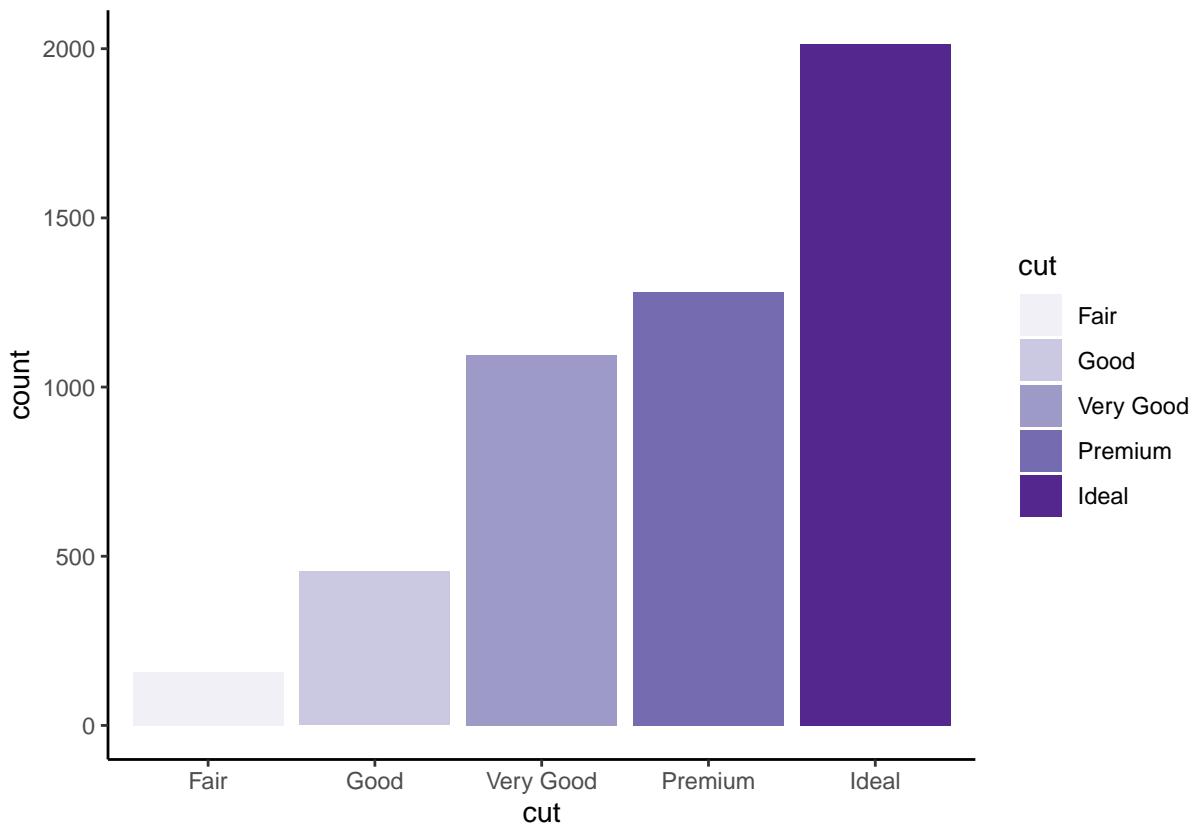
- Couleur dans un diagramme en barres

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



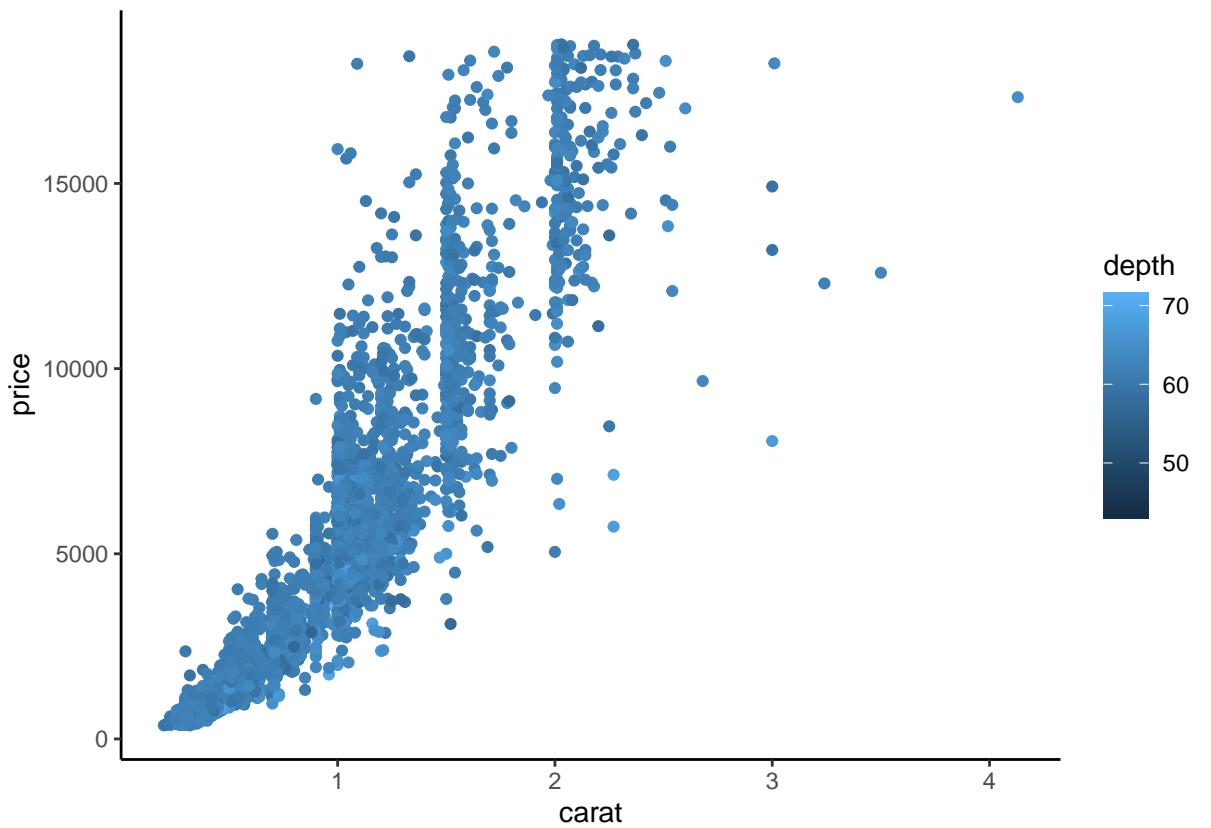
On change la couleur en utilisant la palette **Purples** :

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



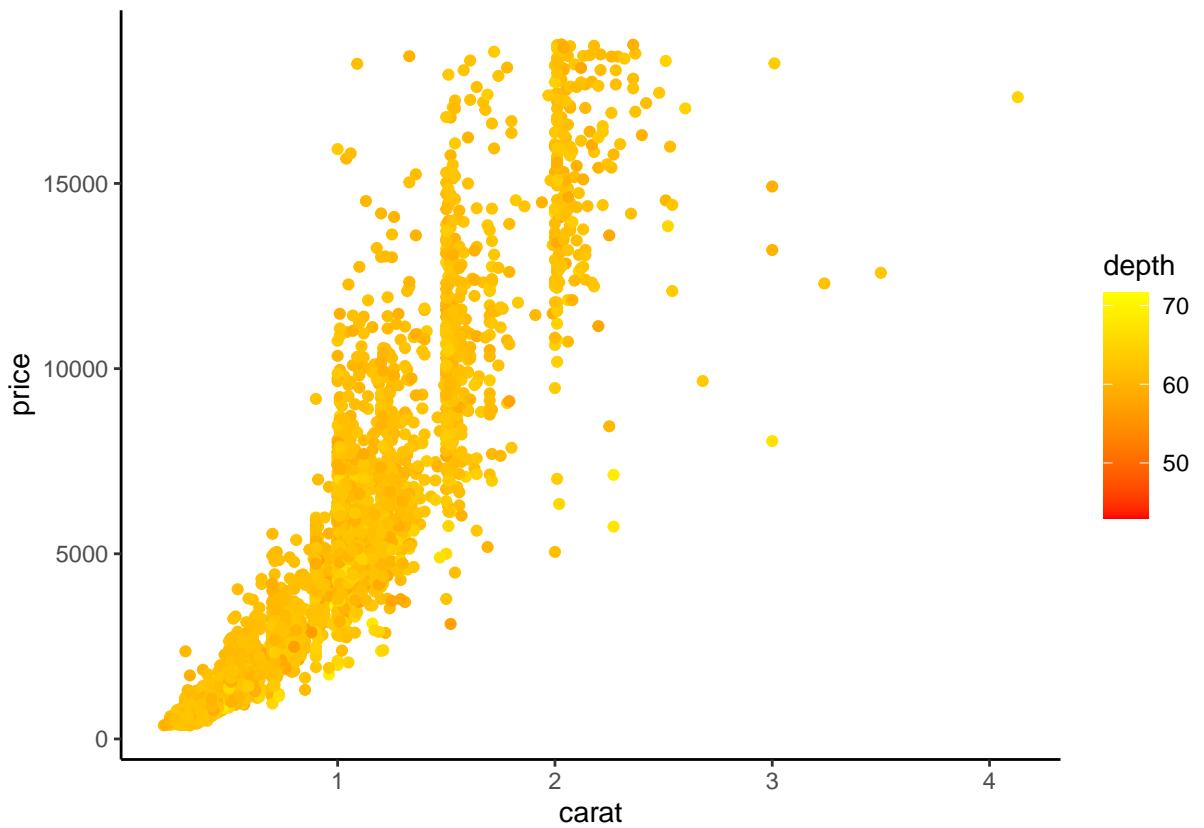
— Gradient de couleurs pour un nuage de points :

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



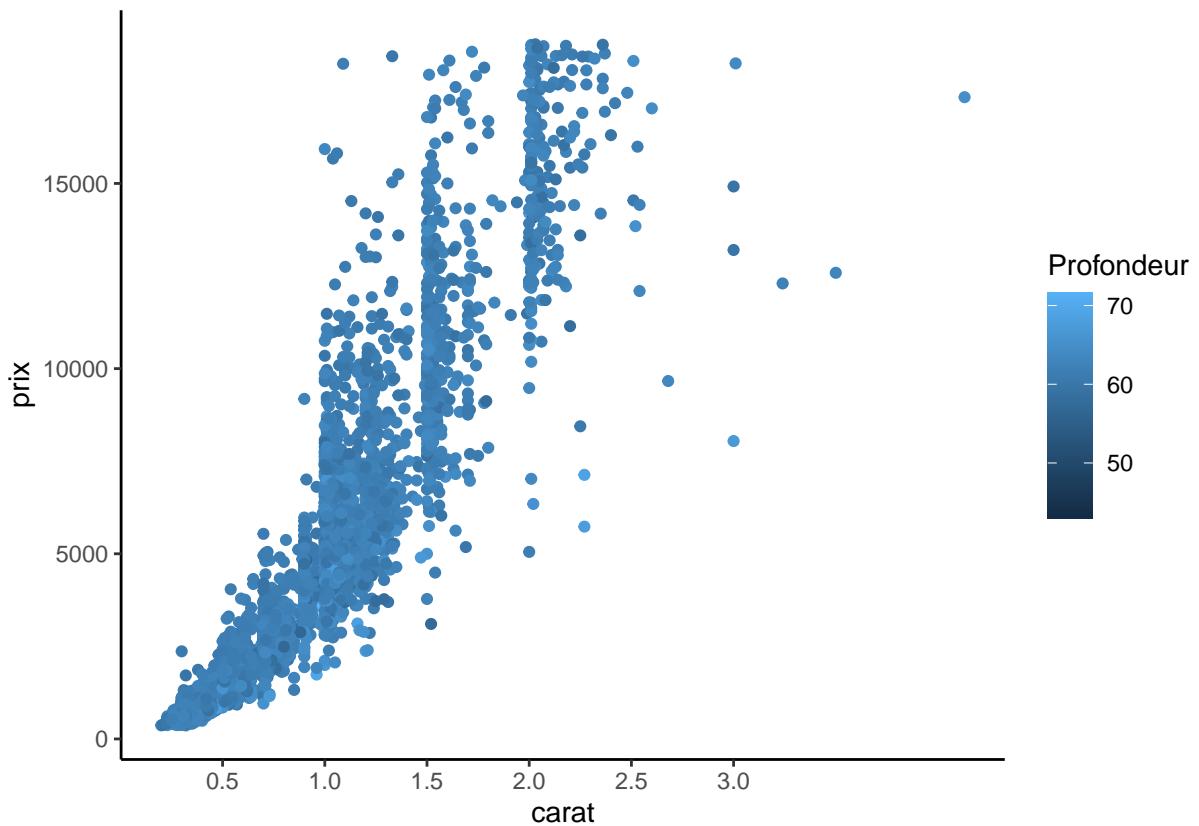
On change le gradient de couleur

```
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")
```



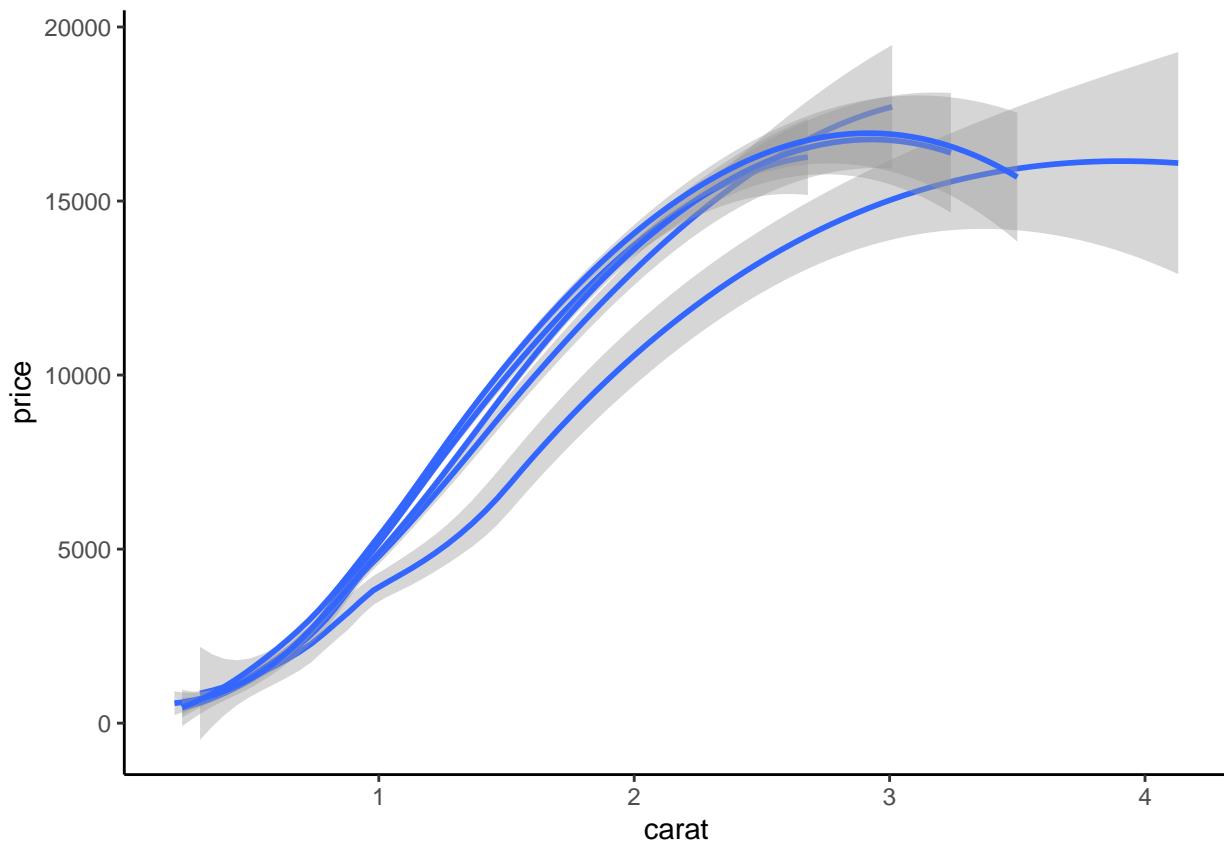
4.2.6 Group et facets

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**.

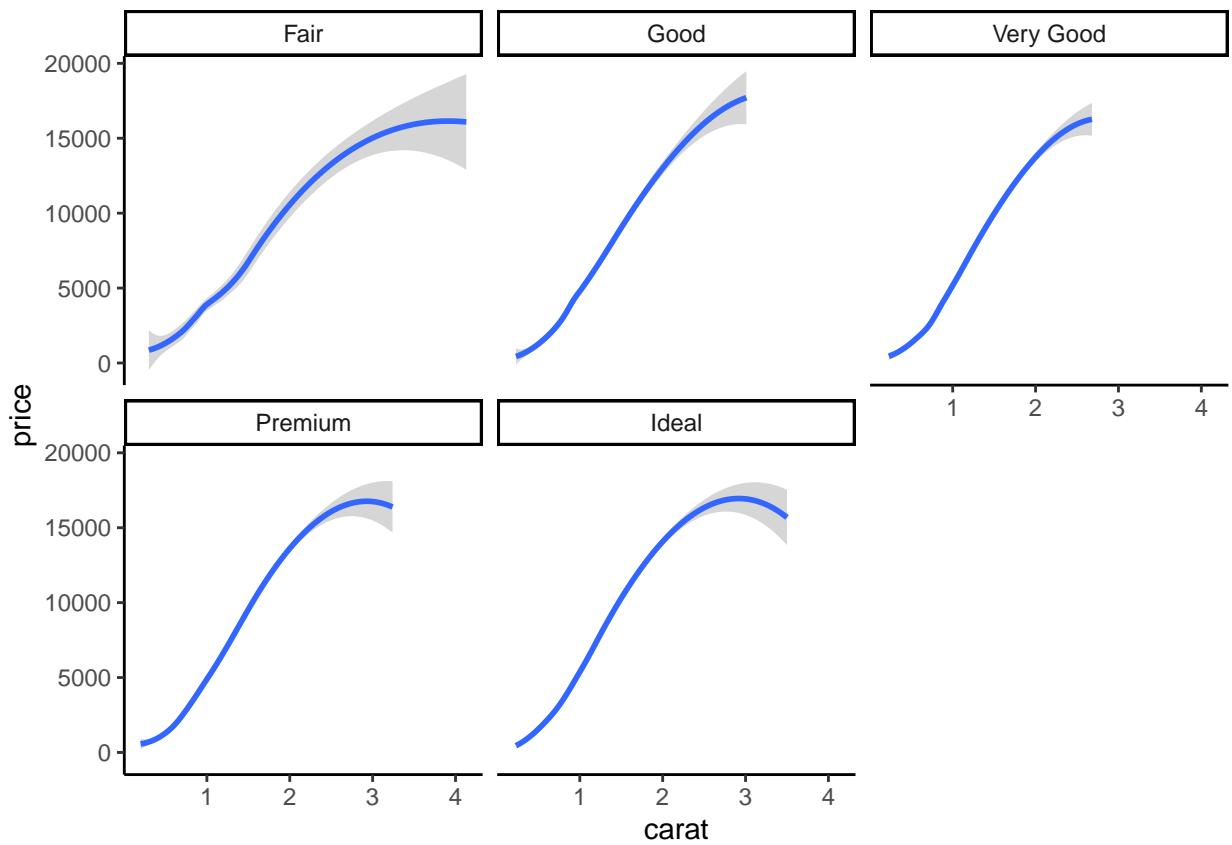
Représentons ici (sur le même graphe) le lissoir **price vs carat** pour chaque modalité de *cut*

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

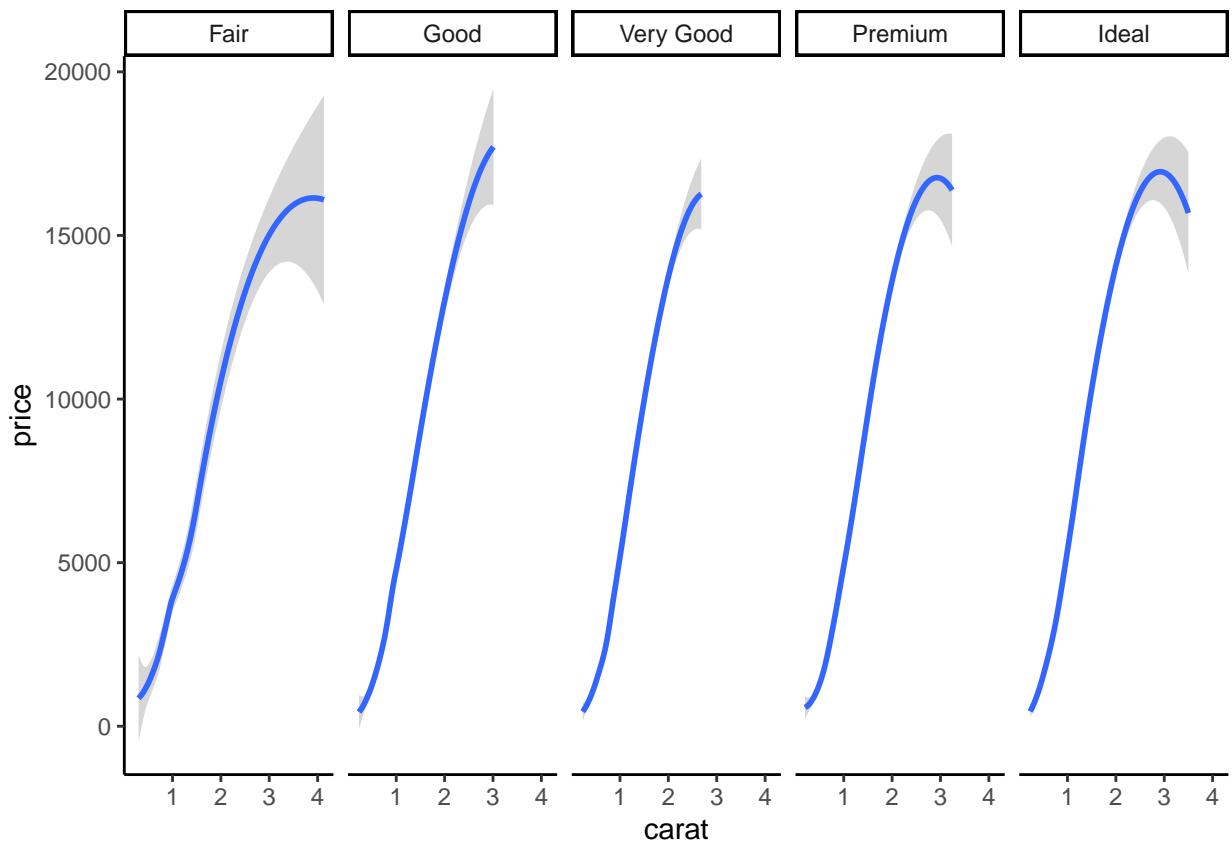


Pour obtenir cette représentation sur plusieurs fenêtres, on utilise

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

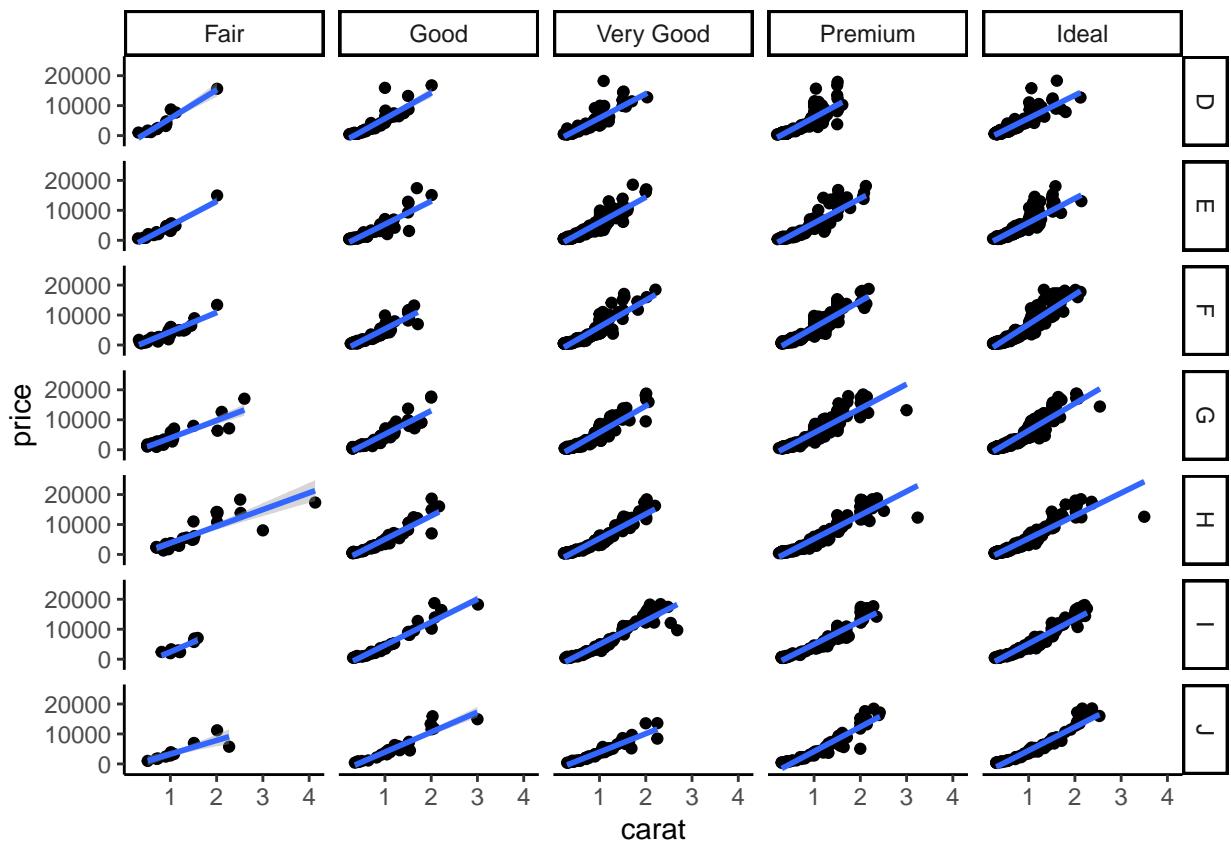


```
ggplot(diamonds2)+aes(x=carat,y=price)+  
  geom_smooth(method="loess")+facet_wrap(~cut,nrow=1)
```

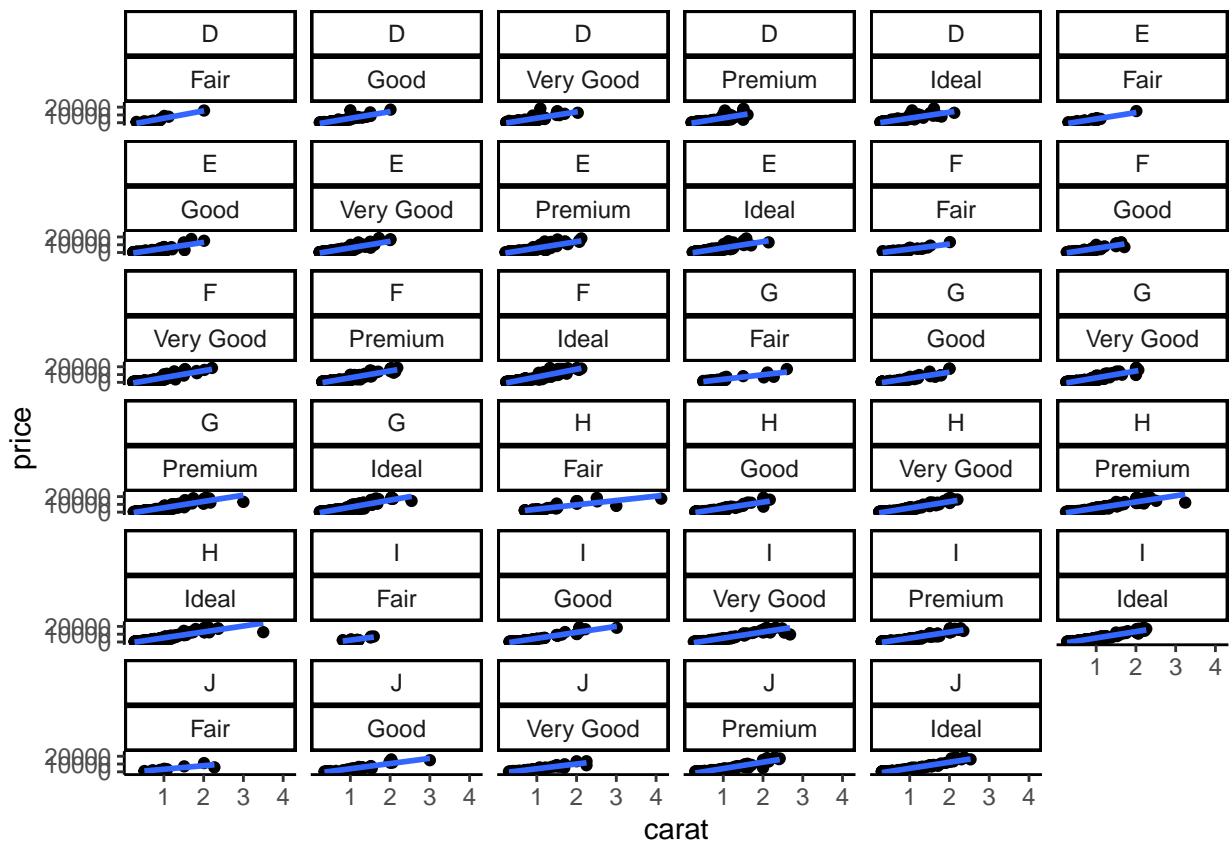


facet_grid et *facet_wrap* font des choses proches mais divisent la fenêtre de façon différente :

```
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)
```



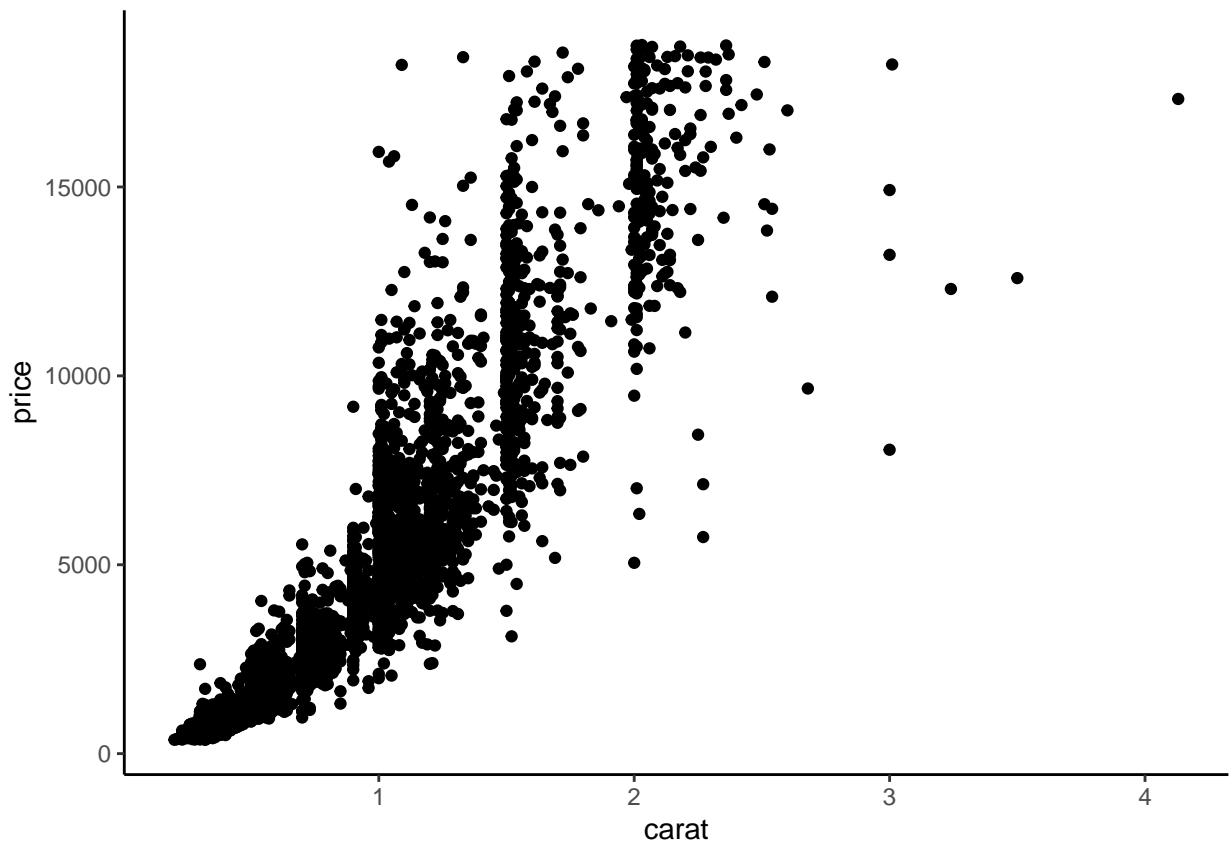
4.3 Compléments

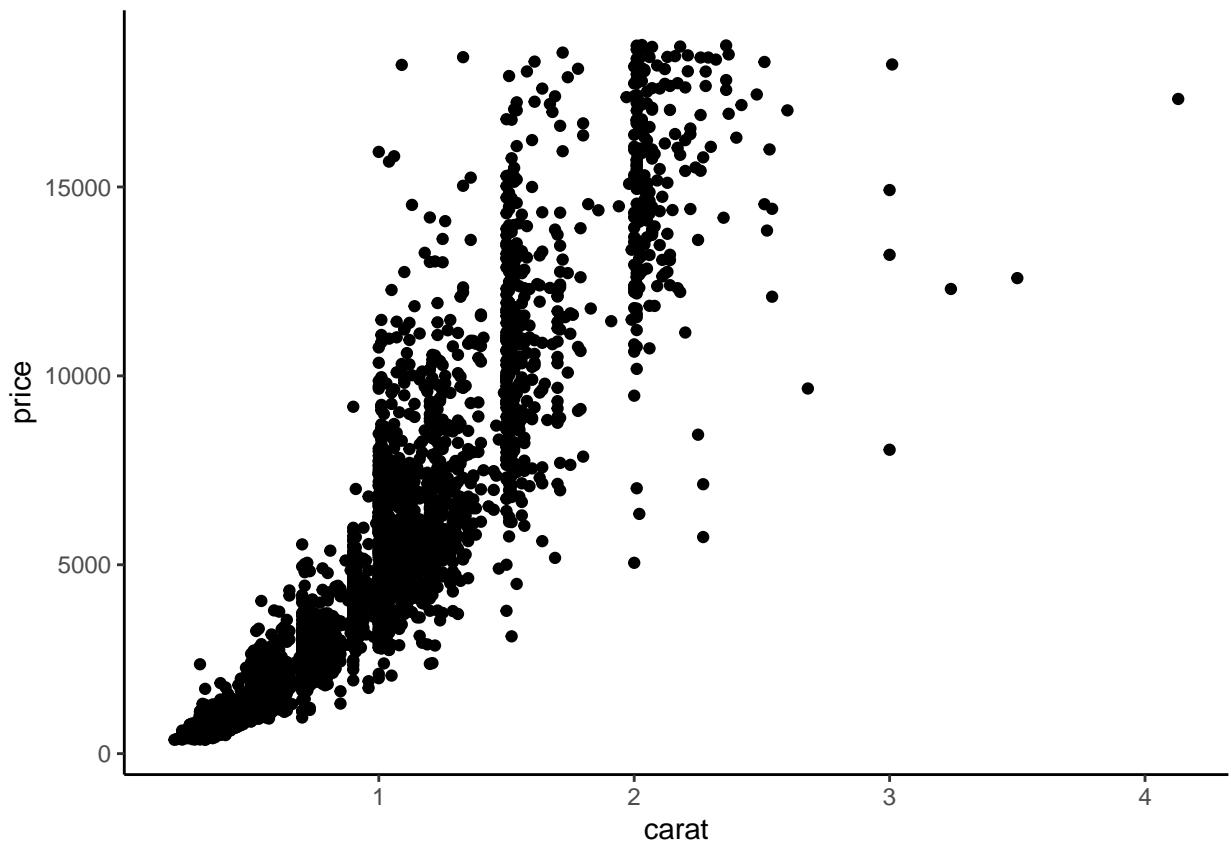
La syntaxe `ggplot` est définie selon le schéma :

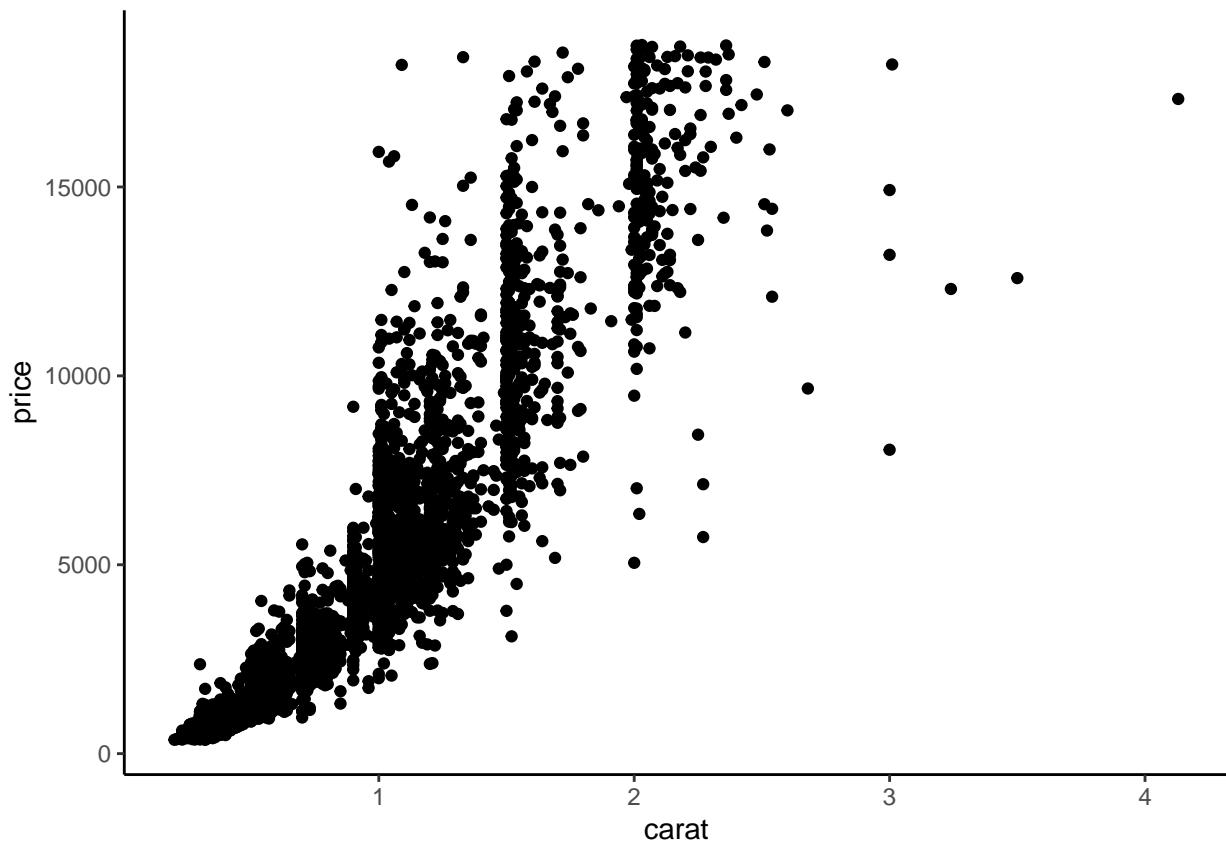
```
ggplot() + aes() + geom_() + scale_()
```

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()
```



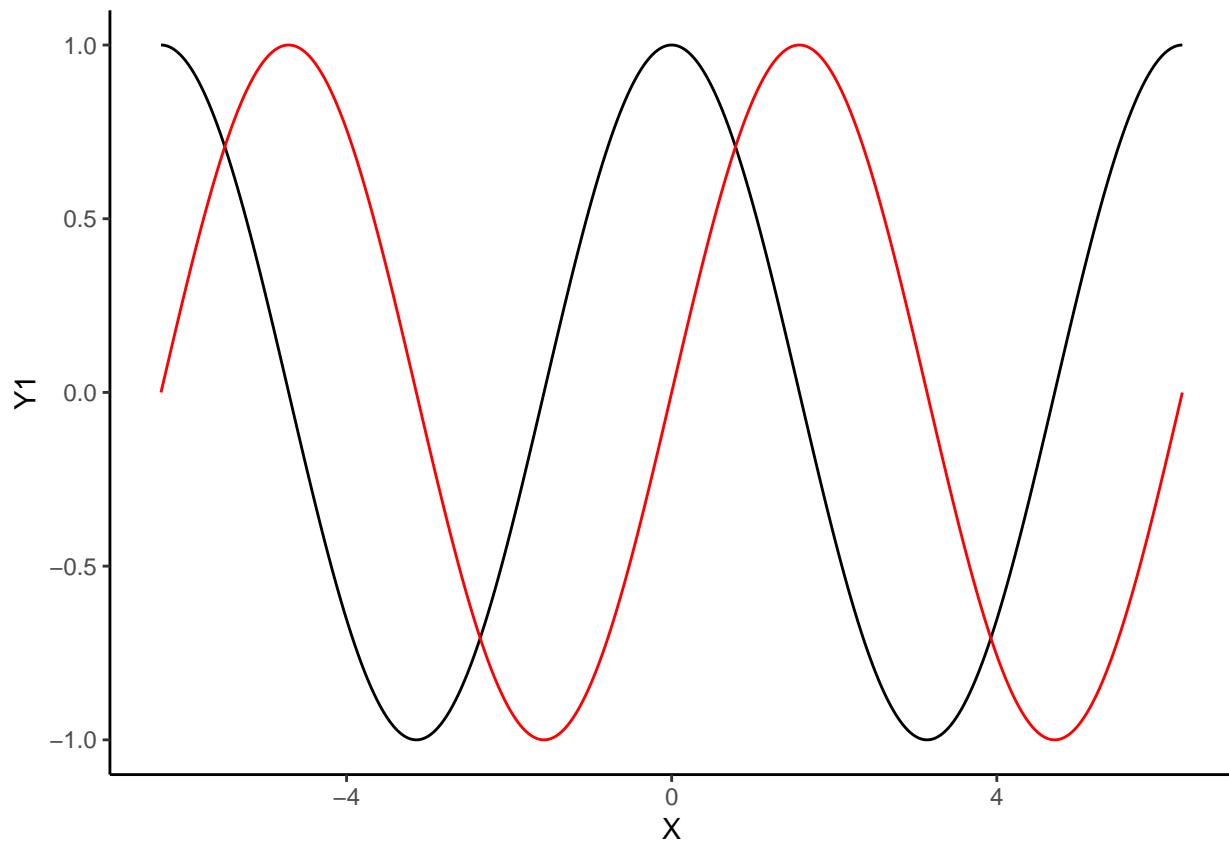




Ceci peut se révéler très utile lorsqu'on utilise des **aes** différents dans les **geom_**.

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

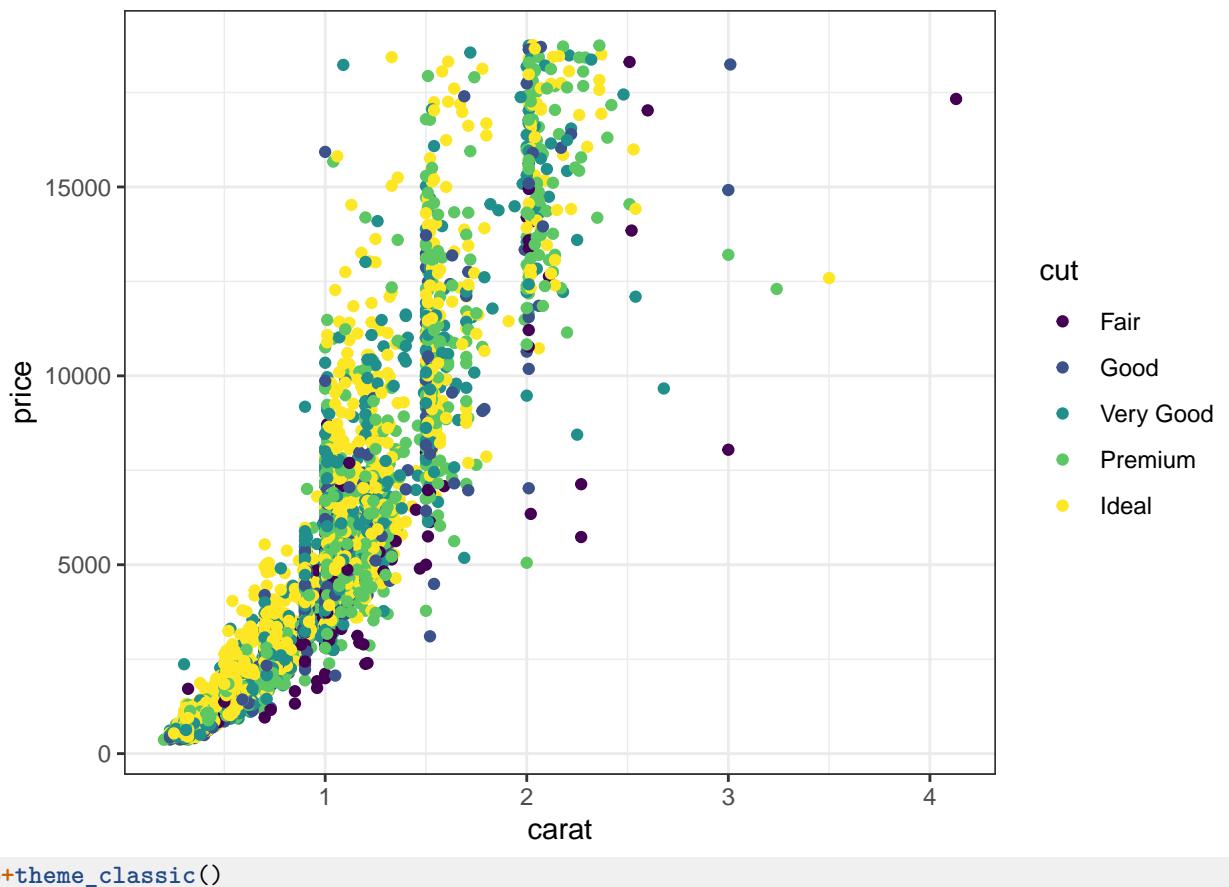
```
X <- seq(-2*pi,2*pi,by=0.001)
Y1 <- cos(X)
Y2 <- sin(X)
donnees1 <- data.frame(X,Y1)
donnees2 <- data.frame(X,Y2)
ggplot(donnees1)+geom_line(aes(x=X,y=Y1))+
  geom_line(data=donnees2,aes(x=X,y=Y2),color="red")
```

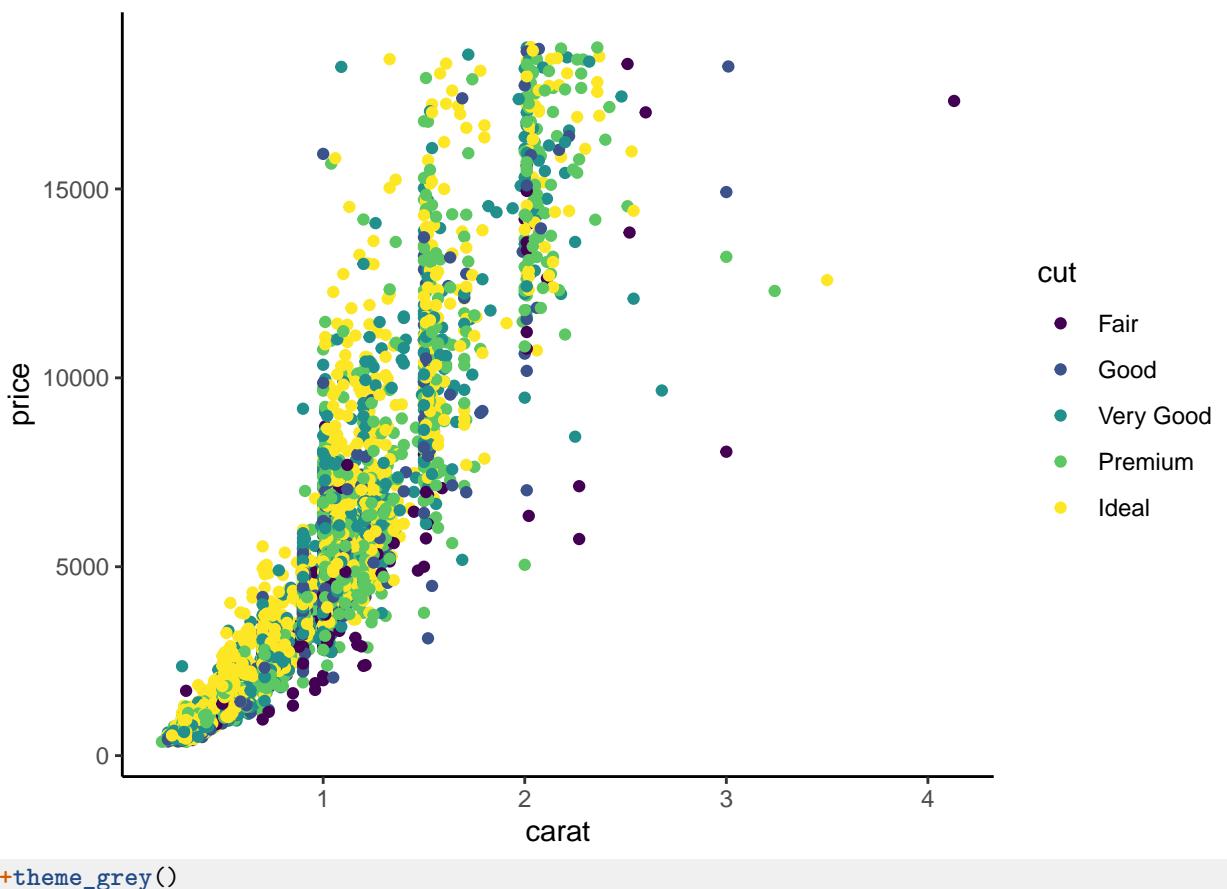


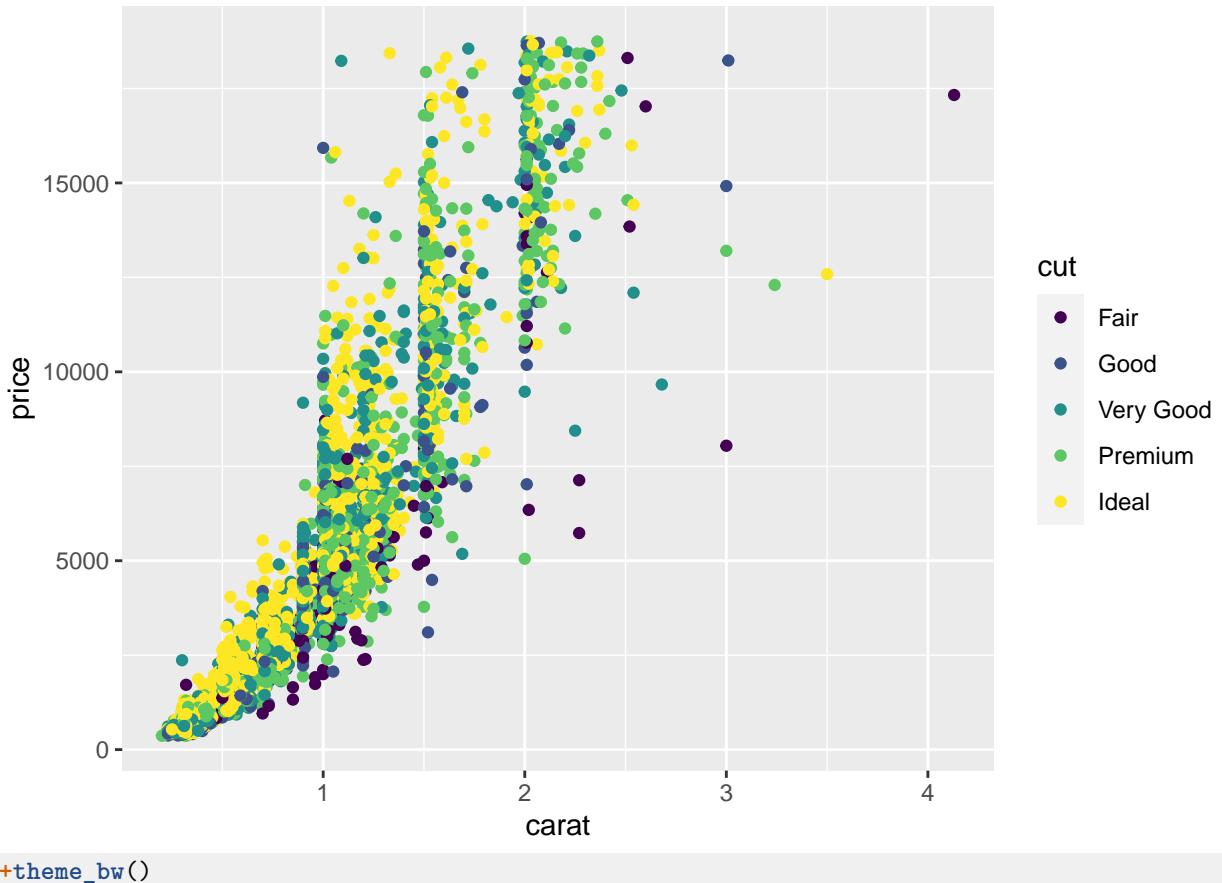
Il existe d'autres fonctions **ggplot** :

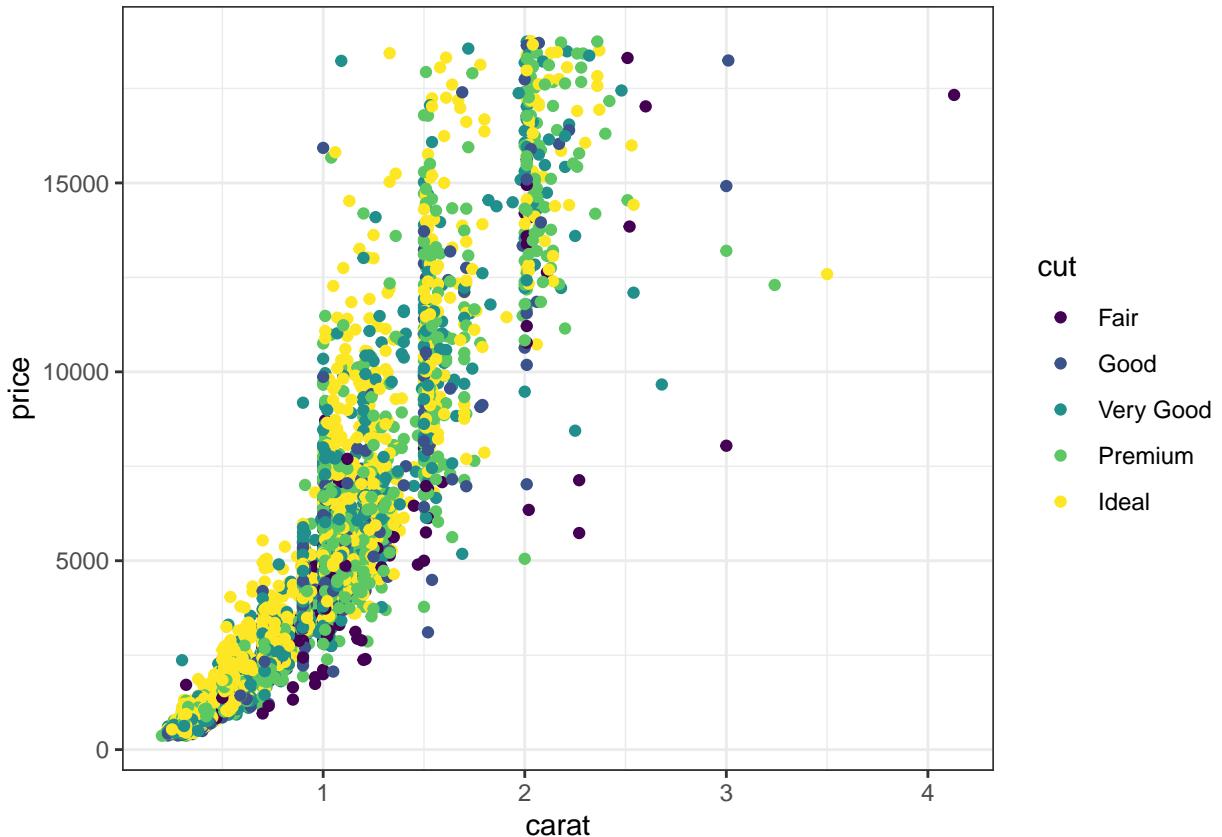
- **ggtitle** pour ajouter un titre.
- **ggsave** pour sauver un graphe.
- **theme_** pour changer le theme du graphe.

```
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).

1. Tracer les fonctions sinus et cosinus. On utilisera tout d'abord deux jeux de données : un pour le sinus, l'autre pour le cosinus.
2. Faire la même chose avec un jeu de données et deux appels à la fonction `geom_line`. On pourra ajouter une légende.
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`.
4. Tracer les deux fonctions sur deux fenêtres graphiques (utiliser `facet_wrap`).
5. Faire la même chose avec la fonction `grid.arrange` du package `gridExtra`.

Exercice 4.10 (Différents graphes).

On considère les données `mtcars`

```
data(mtcars)
summary(mtcars)

  mpg          cyl          disp  
Min.   :10.40  Min.   :4.000  Min.   : 71.1 
1st Qu.:15.43  1st Qu.:4.000  1st Qu.:120.8 
Median :19.20  Median :6.000  Median :196.3 
Mean    :20.09  Mean    :6.188  Mean    :230.7
```

```

3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0
Max.    :33.90   Max.    :8.000   Max.    :472.0
      hp          drat          wt
Min.    : 52.0   Min.    :2.760   Min.    :1.513
1st Qu.: 96.5   1st Qu.:3.080   1st Qu.:2.581
Median  :123.0   Median  :3.695   Median  :3.325
Mean    :146.7   Mean    :3.597   Mean    :3.217
3rd Qu.:180.0   3rd Qu.:3.920   3rd Qu.:3.610
Max.    :335.0   Max.    :4.930   Max.    :5.424
      qsec         vs           am
Min.    :14.50   Min.    :0.0000  Min.    :0.0000
1st Qu.:16.89   1st Qu.:0.0000  1st Qu.:0.0000
Median  :17.71   Median  :0.0000  Median  :0.0000
Mean    :17.85   Mean    :0.4375  Mean    :0.4062
3rd Qu.:18.90   3rd Qu.:1.0000  3rd Qu.:1.0000
Max.    :22.90   Max.    :1.0000  Max.    :1.0000
      gear          carb
Min.    :3.000   Min.    :1.000
1st Qu.:3.000   1st Qu.:2.000
Median  :4.000   Median  :2.000
Mean    :3.688   Mean    :2.812
3rd Qu.:4.000   3rd Qu.:4.000
Max.    :5.000   Max.    :8.000

```

1. Tracer l'histogramme de **mpg** (on fera varier le nombre de classes).
2. Tracer l'histogramme de la densité.
3. Tracer le diagramme en barres de **cyl**.
4. Tracer le nuage de points **disp vs mpg** en utilisant une couleur différente pour chaque valeur de **cyl**.
5. Ajouter le lisseur linéaire sur le graphe.

Exercice 4.11 (Résidus pour régression simple).

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

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

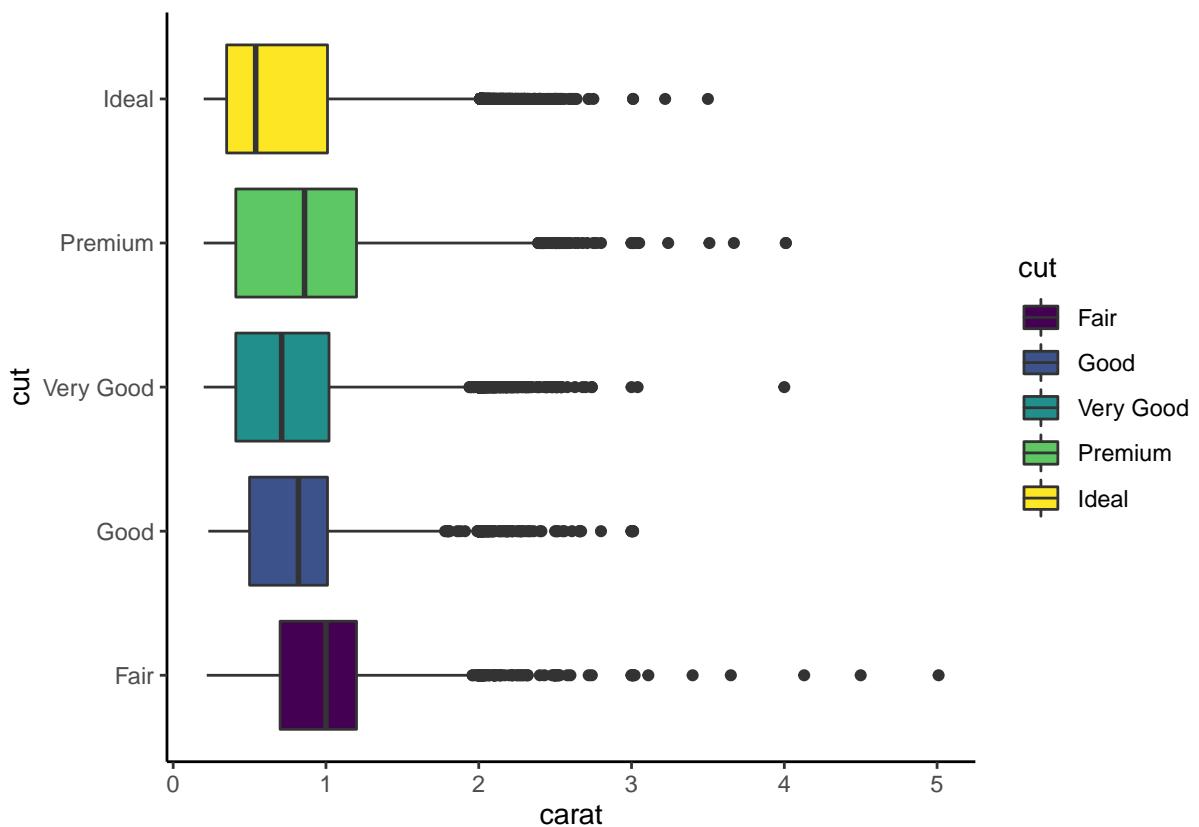
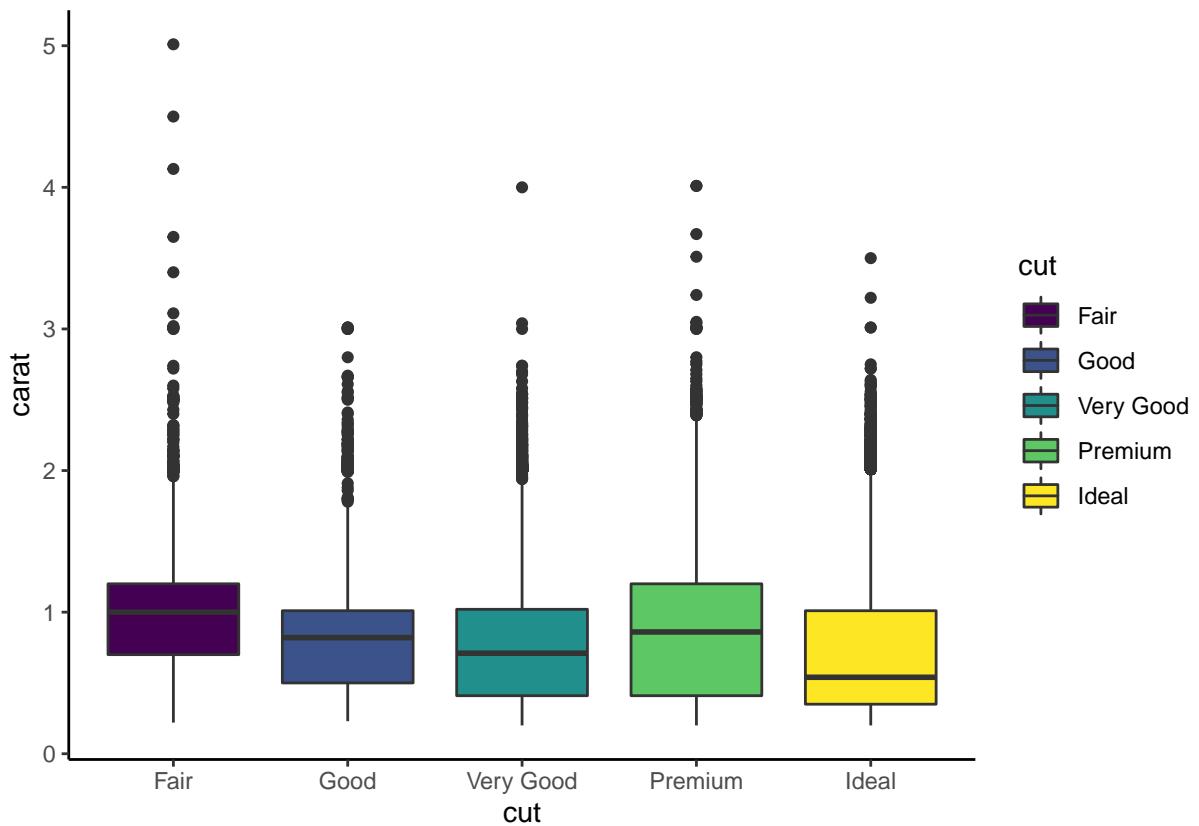
où X_i sont i.i.d. de loi uniforme sur $[0, 1]$ et ε_i sont i.i.d. de loi gaussienne $N(0, 0.2^2)$ (utiliser **rnorm** et **runif**).

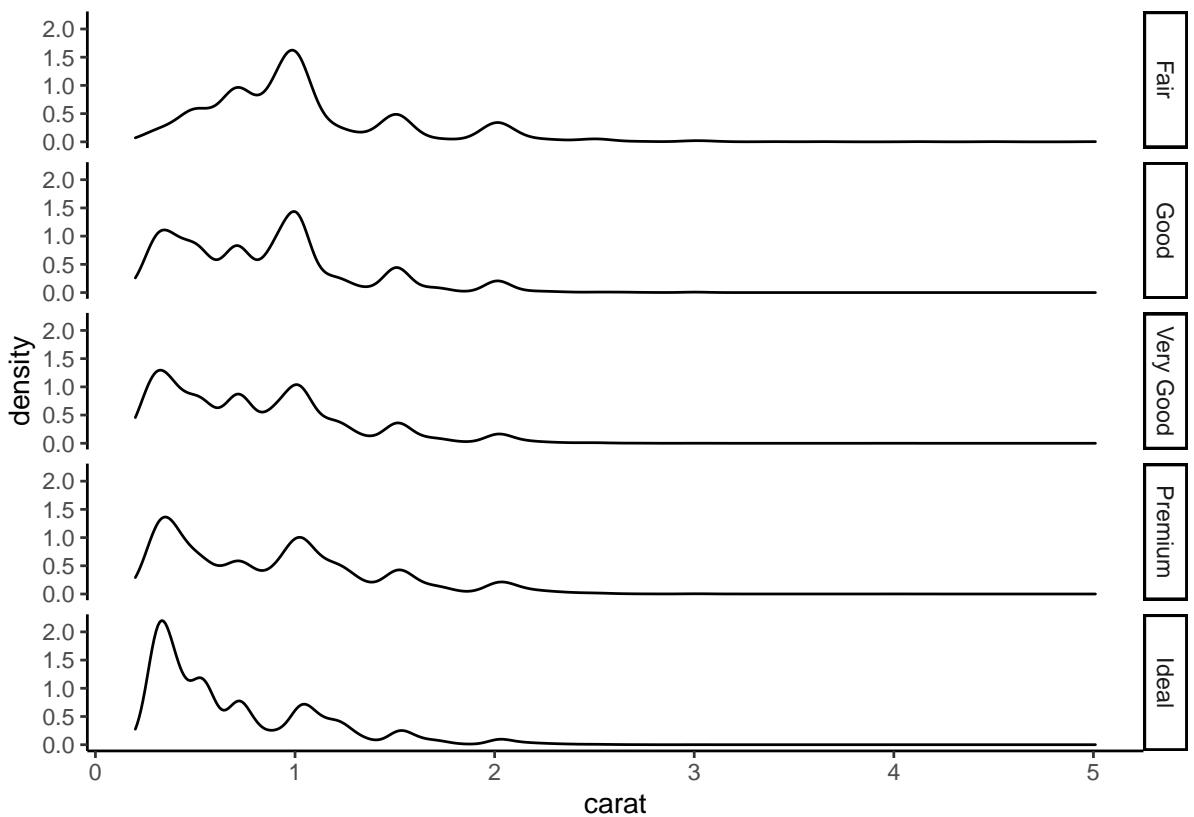
2. Tracer le nuage de points **Y vs X** et ajouter le lisseur linéaire.
3. Représenter les résidus : on ajoutera une ligne verticale entre chaque point et la droite de lissage (utiliser **geom_segment**).

Exercice 4.12 (Challenge).

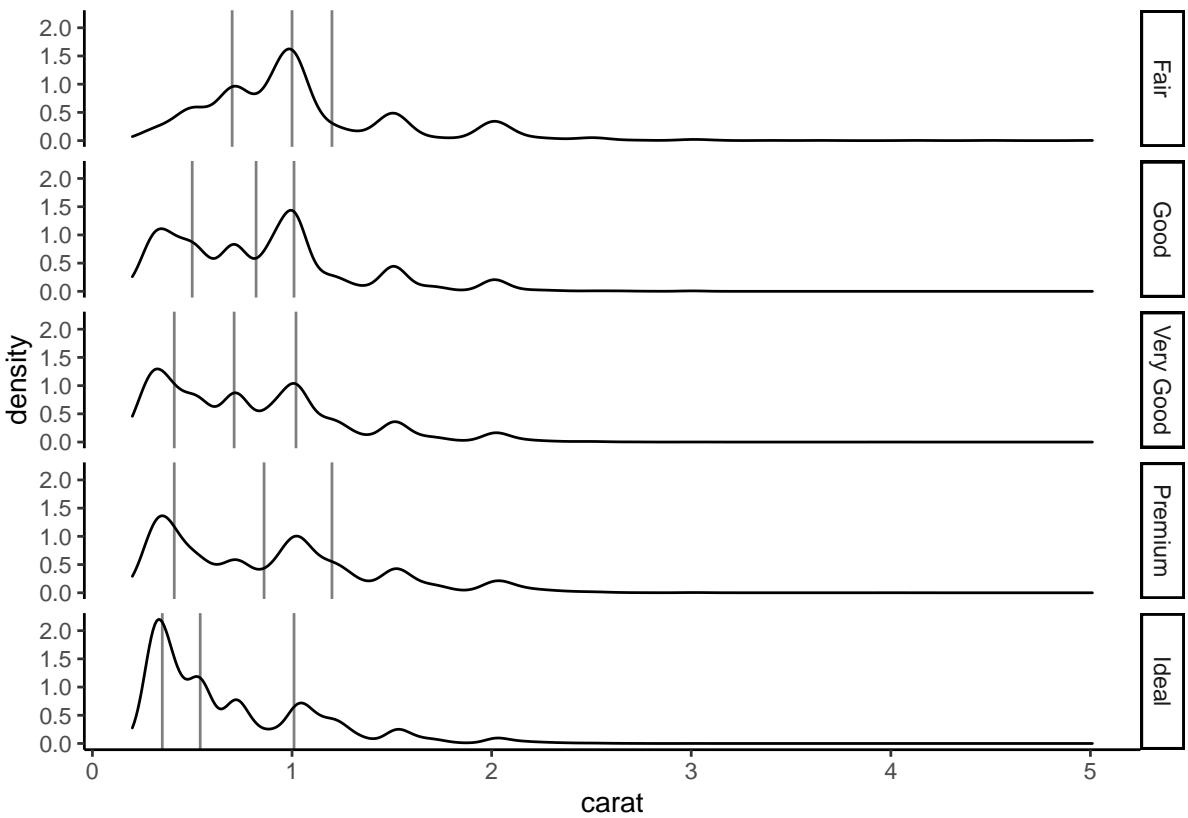
On considère les données **diamonds**.

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

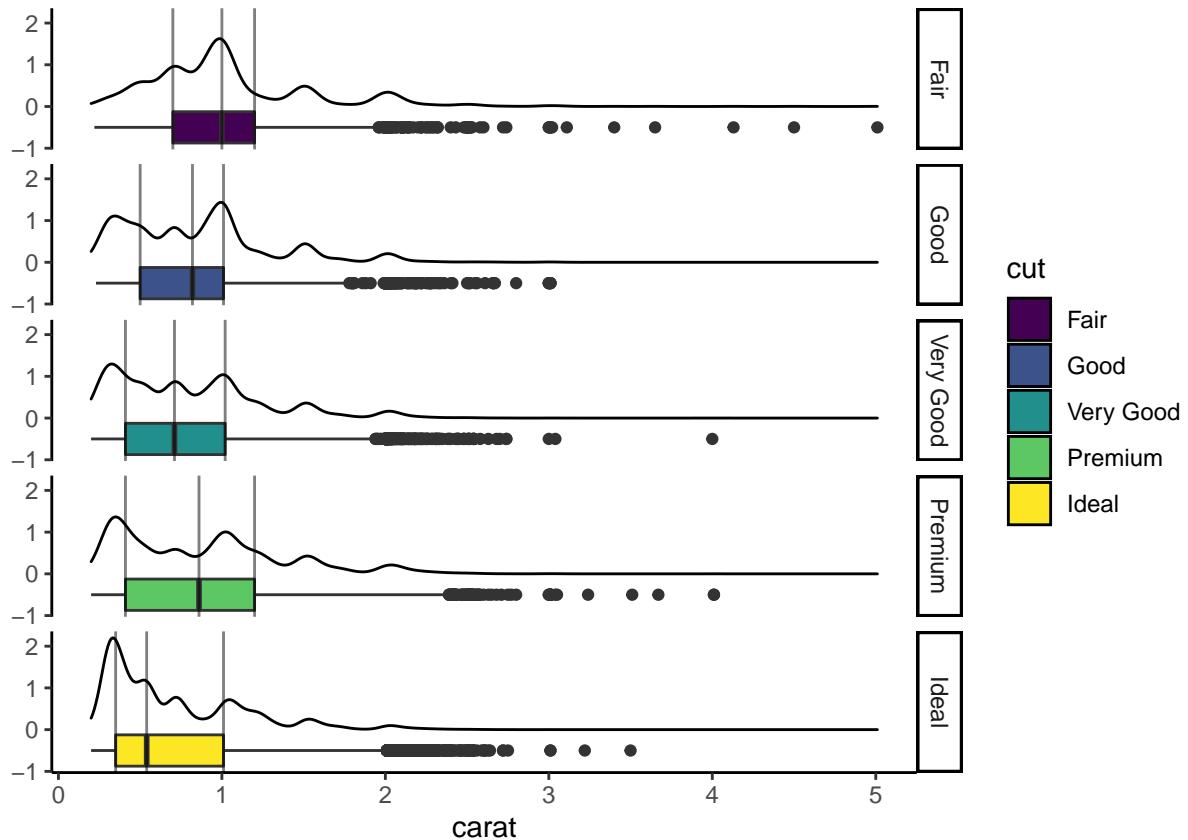




2. Ajouter sur le troisième graphe les quartiles de la variable **carat** pour chaque valeur de **cut**. On utilisera une ligne verticale.



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

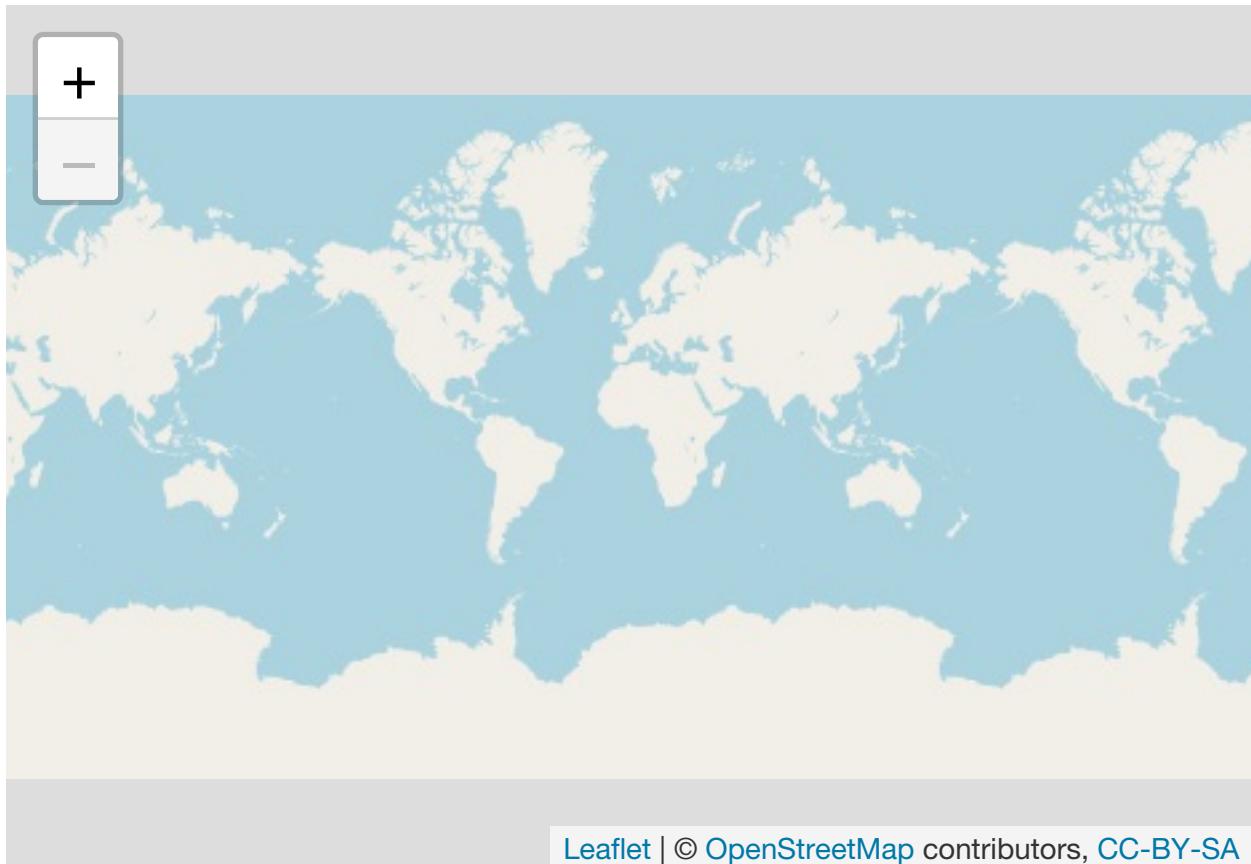


5 Faire des cartes interactives avec leaflet

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 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")
```

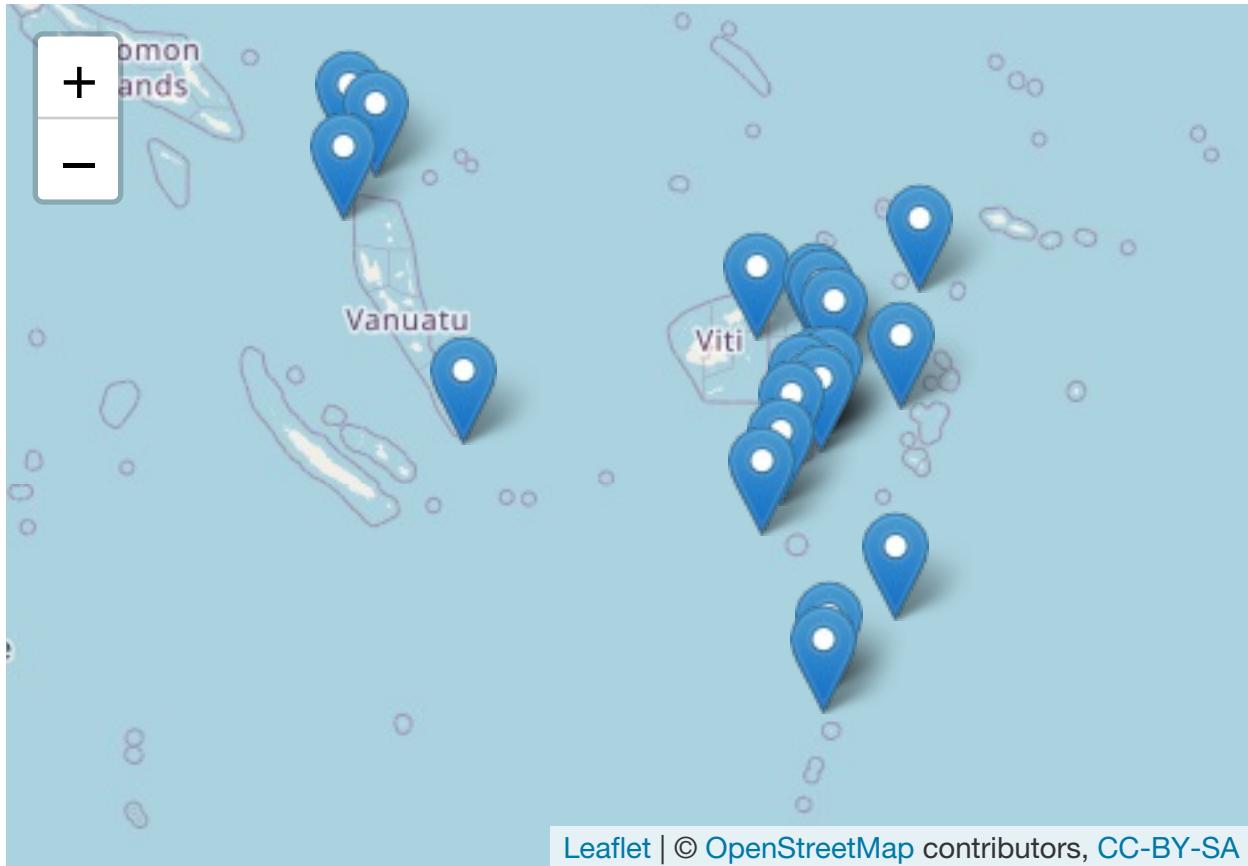


```
m2 %>%
  addProviderTiles("Stamen.Watercolor") %>%
  addProviderTiles("Stamen.TonerHybrid")
```



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`...

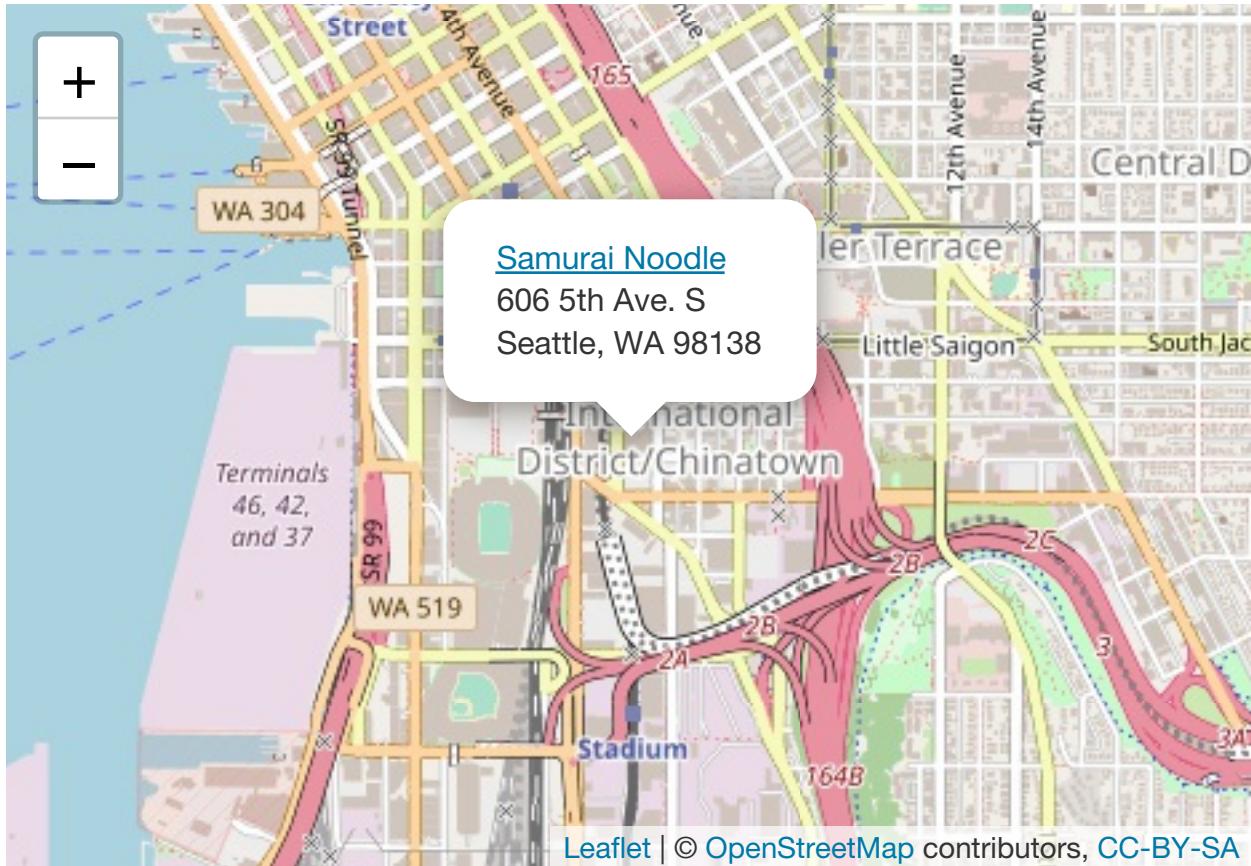
```
data(quakes)
leaflet(data = quakes[1:20,]) %>% addTiles() %>%
  addMarkers(~long, ~lat, popup = ~as.character(mag))
```



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
```

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 requêter (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.paris.fr/explore/dataset/velib-disponibilite-en-temps-reel/information/>. On pourra utiliser la fonction `read_delim` avec l'option `delim=";"`.
2. Décrire les variables du jeu de données.
3. Créer une variable **latitude** et une variable **longitude** à partir de la variable **geo**.
4. Visualiser les positions des stations sur une carte leaflet.
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**).
6. Ajouter le nom de la station dans le popup.
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))
```

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

La fonction devra par exemple renvoyer

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

```
\begin{center}\includegraphics{TUTO_R_files/figure-latex/unnamed-chunk-160-1} \end{center}
```

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

```
\begin{center}\includegraphics{TUTO_R_files/figure-latex/unnamed-chunk-160-2} \end{center}
```

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 |
|---------------------|--------------------------------|--------------|--------------|
| lm | modèle linéaire | | Reg |
| glm | modèle logistique | | Class |
| lda | analyse discriminante linéaire | MASS | Class |
| svm | Support Vector Machine | e1071 | Class |
| knn.reg | plus proches voisins | FNN | Reg |
| knn | plus proches voisins | class | Class |
| rpart | arbres | rpart | Reg et Class |
| glmnet | ridge et lasso | glmnet | Reg et Class |
| gbm | boosting | gbm | Reg et Class |
| randomForest | forêts aléatoires | randomForest | Reg et Class |

Remarque : pour **glmnet**, on ne peut pas utiliser de formule de la forme $Y \sim ..$. 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 ε 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)
```

1. Construire un modèle linéaire permettant d'expliquer Y par X_1, \dots, X_5 (utiliser la fonction **lm**) et afficher les estimateurs de β_0, \dots, β_5 (on pourra utiliser les fonctions **coef** et **summary**).
2. 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*).

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

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

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 $\beta_0, \dots, \beta_{105}$.
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**.

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

1. 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)
```

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

5. Visualiser l'arbre construit à l'aide des fonctions **rpart.plot** et **visTree** des packages **rpart.plot** et **visNetwork**
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.
7. Ajouter au data-frame précédent une colonne où on mettra les valeurs observées de la variable à expliquer.
8. A l'aide de **summarize_at** calculer les erreurs de classification des 3 modèles.
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**.

7 Estimation et intervalles de confiance

Dans cette partie nous allons construire des échantillons par simulation et nous intéresser à l'étude de la moyenne de ces échantillons.

7.1 Générer des observations selon des lois de probabilités

R étant un logiciel de statistique, il est bien entendu possible de

- visualiser
- calculer des quantiles
- générer des observations

pour toutes les lois classiques de probabilités. Chaque loi va être identifiée par une chaîne de caractères :

| Loi | Chaine |
|---------------|--------|
| Binomiale | binom |
| Poisson | pois |
| Uniforme | unif |
| Exponentielle | exp |
| Normale | norm |

Un préfixe permettra de spécifier l'action que l'on souhaite effectuer sur la loi :

- **d** : calculer la densité pour une loi continue ou la fonction de masse pour une loi discrète
- **q** : calculer les quantiles
- **r** : générer des observations.

On pourra par exemple :

- Calculer la densité de la loi $\mathcal{N}(0, 1)$ en -1,0,1 avec

```
dnorm(c(-1,0,1),mean=0,sd=1)
[1] 0.2419707 0.3989423 0.2419707
```

- Calculer les quantiles d'ordre 0.05, 0.5 et 0.95 de la loi $\mathcal{N}(0, 1)$ en -1,0,1 avec

```
qnorm(c(0.05,0.5,0.95),mean=0,sd=1)
[1] -1.644854  0.000000  1.644854
```

- Générer 10 observations selon une loi $\mathcal{N}(0,1)$ avec

```
rnorm(10,mean=0,sd=1)
[1]  0.20222019 -1.19621049  0.04375938 -0.05545961
[5]  1.08570925  1.76993522  1.98121980  0.67664960
[9] -1.00061208 -0.31680937
```

Exercice 7.1 (Loi binomiale).

1. Soit X un variable de loi binomiale $B(20,0.6)$. Calculer la probabilité que X soit égale à 1,5,10,15.
2. Représenter le diagramme en barre associé à la loi $B(20,0.6)$. On pourra utiliser l'argument **stat="identity"** dans la fonction **geom_bar**.
3. Générer un échantillon de taille 5000 selon la loi $B(20,0.6)$. Tracer le diagramme en barres associé à cet échantillon et comparer le à celui de la question précédente.

Exercice 7.2 (Loi normale).

1. Tracer la densité de la loi $\mathcal{N}(0,1)$.
2. Générer un échantillon de taille 5000 selon la loi $\mathcal{N}(0,1)$. Tracer l'histogramme associé à cet échantillon et comparer le à la densité tracée à la question précédente (on pourra superposer les 2 représentations).

7.2 Une étude numérique de la moyenne empirique.

On s'intéresse dans cette partie à l'étude de la moyenne empirique

$$\bar{x}_n = \frac{1}{n} \sum_{i=1}^n x_i$$

d'un échantillon i.i.d x_1, \dots, x_n . Remarquons déjà que la moyenne empirique dépend des observations x_1, \dots, x_n : la moyenne va donc changer lorsque les observations changent.

7.2.1 Exemple.

On considère deux échantillons de taille 20 générées selon une loi uniforme entre 0 et 1 :

```
ech1 <- runif(20)
ech2 <- runif(20)
df <- data.frame(ech1,ech2)
```

Les moyennes empiriques pour ces deux échantillons sont différentes :

```
df %>% summarise_all(mean)
      ech1      ech2
1 0.6333745 0.4300223
```

Ainsi la moyenne empirique peut-être considérée comme une variable aléatoire : elle va donc posséder une loi de probabilité, une espérance... Si on considère l'exemple précédent, on sent bien que la distribution de la moyenne empirique doit

- se répartir autours de 0.5 (qui est la valeur à estimer).
- être de plus en plus concentrée autours de 0.5 lorsque le nombre d'observations n augmente.

On peut visualiser ce fait en

- générant nombre B (grand) d'échantillons de taille $n = 20$ selon une loi uniforme entre 0 et 1.

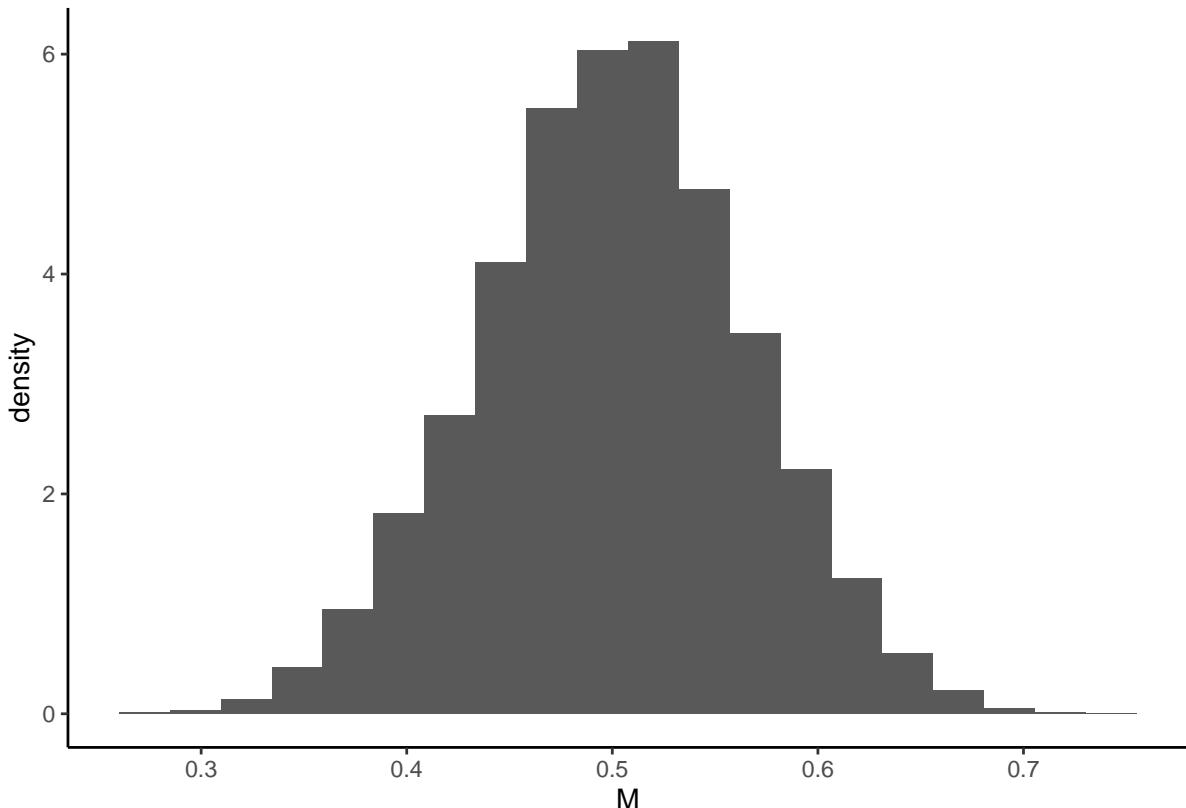
```
set.seed(1234)
df <- matrix(runif(20*5000), nrow=20) %>% as.data.frame()
```

- calculant les moyennes obtenues pour chaque échantillon

```
moy <- df %>% summarize_all(mean)
head(t(moy))
[,1]
V1 0.4719301
V2 0.4449401
V3 0.4833523
V4 0.3740339
V5 0.4132300
V6 0.3734092
```

- visualisant la distribution de la moyenne de chaque échantillon (en traçant l'histogramme de ces valeurs par exemple).

```
moy <- data.frame(M=t(moy))
ggplot(moy)+aes(x=M,y=..density..)+geom_histogram(bins=20)+theme_classic()
```



Le théorème central limite nous dit que cette moyenne \bar{X}_n vérifie

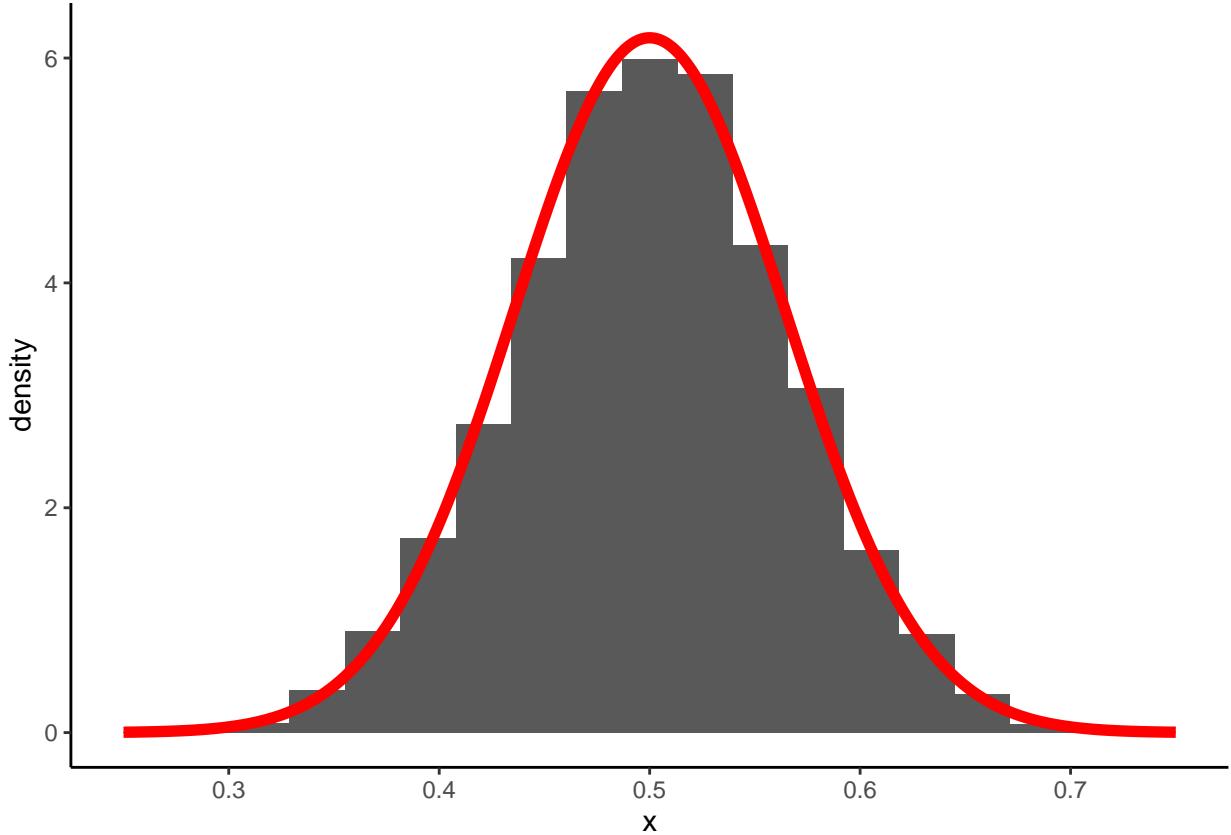
$$\sqrt{n} \frac{\bar{X}_n - \mu}{\sigma} \rightarrow \mathcal{N}(0, 1)$$

avec $\mu = 0.5$ et $\sigma = 1/\sqrt{12}$ ici. On a donc

$$\sqrt{n} \frac{\bar{X}_n - 0.5}{1/\sqrt{12}} \rightarrow \mathcal{N}(0, 1)$$

Ce qui signifie qu'on peut approcher la loi de \bar{X}_n par la loi $\mathcal{N}(0.5, 1/(12n))$ avec $n = 20$.

```
x <- seq(0.25, 0.75, by=0.001)
df <- data.frame(x=x, y=dnorm(x, 0.5, 1/(sqrt(12*20))))
ggplot(moy)+aes(x=M, y=..density..)+geom_histogram(bins=20)+
  geom_line(data=df, aes(x=x, y=y), color="red", size=2)+theme_classic() +xlab("x")
```



Exercice 7.3 (Théorème central limite).

Faire le même travail pour des tailles d'échantillon de 50, 100 et 500. Interpréter.

Exercice 7.4 (Théorème central limite (toujours)).

Faire le même exercice pour une loi gaussienne $\mathcal{N}(1, 2)$ et une loi de Bernoulli $\mathcal{B}(0.6)$.

7.3 Intervalles de confiance

On cherche ici à illustrer numériquement le niveau d'un intervalle de confiance. On rappelle que $[A, B]$ est un IC de niveau $1 - \alpha$ pour un paramètre θ si

$$P(\theta \in [A, B]) = 1 - \alpha.$$

Exercice 7.5 (IC pour l'espérance d'une gaussienne).

On fixe ici le niveau à 0.95 ($\alpha = 0.05$). On considère n observations x_1, \dots, x_n i.i.d de loi $\mathcal{N}(\mu, 1)$ et on cherche un intervalle de confiance pour μ .

1. Générer $n = 100$ observations i.i.d. selon la loi $\mathcal{N}(\mu, 1)$ avec $\mu = 1$.
2. Calculer un intervalle de confiance pour μ de niveau 0.95.
3. Selon-vous, peut-on dire que la probabilité que μ appartienne à l'intervalle trouvé est de 0.95 ? Si non, comment peut-on interpréter cette formule.
4. Générer 5000 échantillons i.i.d. de loi $\mathcal{N}(1, 1)$ de tailles 100. On pourra mettre le tout dans une matrice de taille 5000*100.
5. Calculer un intervalle de confiance de niveau 0.95 pour chacun des 5000 échantillons. On pourra utiliser une boucle **for** ou la fonction **apply**.
6. Sur les 5000 intervalles obtenus, calculer le nombre de fois où la vraie valeur de μ (en l'occurrence ici 1) se trouve à l'intérieur de l'intervalle.
7. Refaire les questions 5-6-7 avec des IC de niveau 0.90.

Exercice 7.6 (IC pour les iris de Fisher).

On considère les données sur les iris de Fisher. Construire un intervalle de confiance de niveau 90% pour les paramètres suivants :

- La longueur de Pétales moyenne
- La largeur de Sépales moyenne de l'espèce Setosa
- La largeur de Sépales moyenne des espèces Versicolor et Virginica

Exercice 7.7 (IC pour une proportion).

On considère x_1, \dots, x_n un échantillon i.i.d issu d'une loi de Bernoulli de paramètre $p \in [0, 1]$ inconnu.

1. Proposer un estimateur \hat{p} pour p .
2. A l'aide du TCL, obtenir la loi asymptotique de \hat{p} .
3. En déduire un intervalle de confiance de niveau 1_α pour p .
4. Que pouvez-vous reprocher à l'intervalle proposé à la question précédente ?
5. Proposer une solution.

Exercice 7.8 (IC pour une proportion (suite)).

Afin de tenter de deviner qui va gagner une élection entre deux candidats A et B on effectue un sondage. On demande à 100 personnes pour qui elles vont voter. Les résultats sont reportés dans le vecteur suivant.

```
set.seed(12345)
res <- rbinom(100, 1, 0.52)
```

On désigne par p la proportion (inconnue) d'électeurs qui vont voter pour A .

1. Proposer et calculer un estimateur de p .
2. Que pouvez-vous conclure a priori.

Il semble que A va remporter l'élection.

3. En vous basant sur l'exercice précédent, calculer un intervalle de confiance de niveau 95% pour p .
4. Est-ce que l'intervalle obtenu conforte votre conclusion de la question 2 ?
5. Calculer un intervalle de confiance pour p à l'aide de la fonction **prop.test**.

Exercice 7.9 (Comparaison de moyennes).

Pour le jeu de données **decathlon** disponible ici

```
library(FactoMineR)
data(decathlon)
```

on souhaite comparer les performances au 100m en fonction de la compétition (Decastar vs JO).

1. Calculer un intervalle de confiance de niveau 95% pour la vitesse moyenne au 100m au Decastar.
2. Même question pour les jeux olympiques.
3. Pouvez-vous conclure sur la question posée ? Si non, que faire ?

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.