

Tutorial : Dynamic data visualization with R

Laurent Rouvière

2020-10-04

Table des matières

Presentation	1
1 Data visualization with ggplot2	2
1.1 Conventional Graphical Functions	2
1.2 Ggplot2 grammar	11
1.3 Complements	30
1.4 Some exercises	38
2 Mapping with R	43
2.1 ggmap package	43
2.2 Maps with contours, shapefile format	47
2.3 Dynamic maps with leaflet	54
3 Dynamic visualization	63
3.1 Classical charts with rAmCharts and plotly	63
3.2 Graphs to visualize networks with visNetwork	69
3.3 Dashboard	73
4 Shiny web applications	74
4.1 First step	74
4.2 Input - output	74
4.3 Organizing the application	75
4.4 Adding interactive charts	76
4.5 Reactive, isolation, observe, html,	76
4.6 Some additional exercises	77

Presentation

This tutorial provides some **R** tools for **data visualization**. You can find

- materials (slides) associated with this tutotial as well as datasets at <https://lrouviere.github.io/DATAVIZ/>;
- tutorial without correction at https://lrouviere.github.io/TUTO_DATAVIZ/;
- tutorial with corrections at https://lrouviere.github.io/TUTO_DATAVIZ/correction/.

I strongly encourage to use **mozilla firefox** to read this tutorial.

Basics on **R** and computer programming are necessary. The tutorial is divided into tree parts :

- **Visualization with ggplot2** : presentation of the **ggplot2** grammar ;
- **Mapping with R** : how to build map with **ggmap**, **sf** and **leaflet** ;

- Dynamic visualization : some tools to make interactive visualization (rAmCharts, plotly), dashboard and web applications (shiny).

1 Data visualization with ggplot2

Graphs are often the starting point for statistical analysis. One of the main advantages of **R** is how easy it is for the user to create many different kinds of graphs. We begin this chapter by studying conventional graphs, followed by an examination of some more complex representations. This final (and main) part uses the **ggplot2** package.

1.1 Conventional Graphical Functions

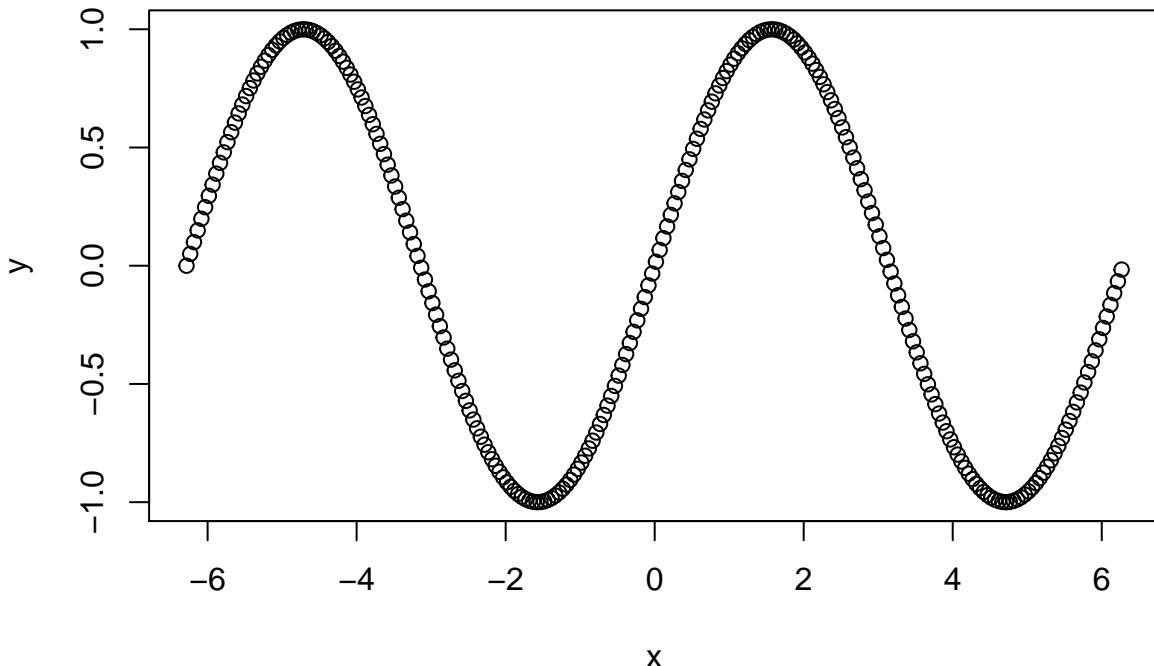
To begin with, it may be interesting to examine few examples of graphical representations which can be constructed with **R**. We use the **demo** function :

```
demo(graphics)
```

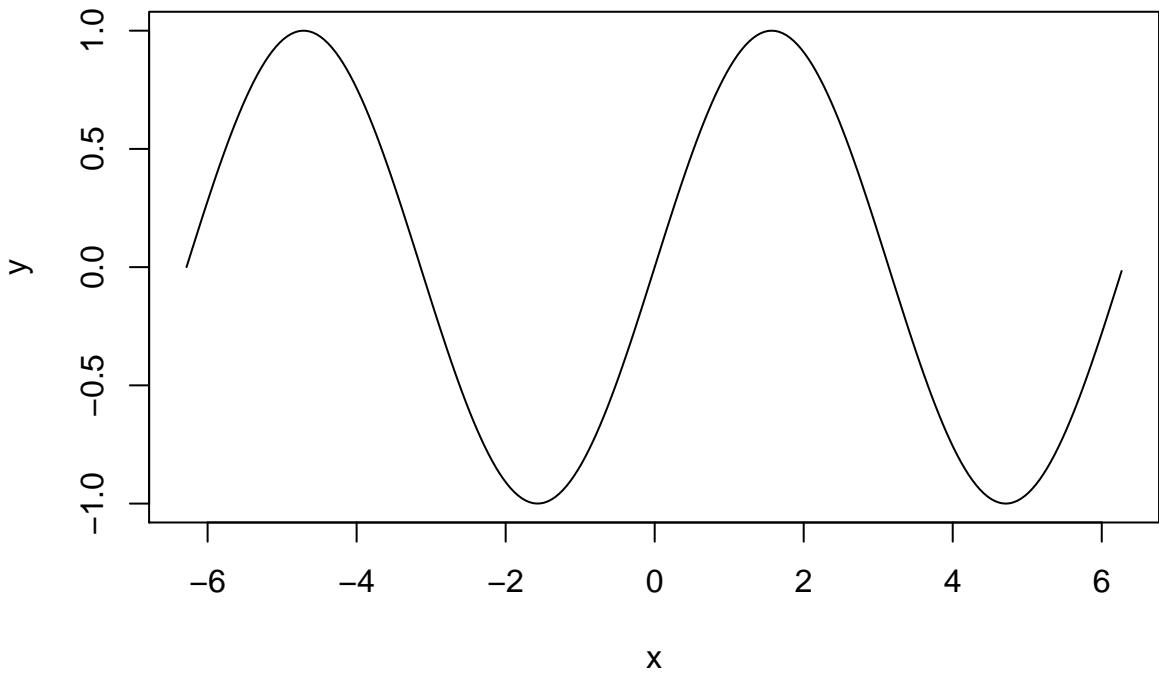
1.1.1 The plot function

The **plot** function is a **generic function** used to represent all kinds of data. We can use this function for representing a scatterplot for a variable y according to another variable x . For example, to represent the graph of the function $x \mapsto \sin(2\pi x)$ on $[0, 1]$, at regular steps we use the following commands :

```
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
```



We provide examples of representations for quantitative and qualitative variables. We use the data file `ozone.txt` :

```

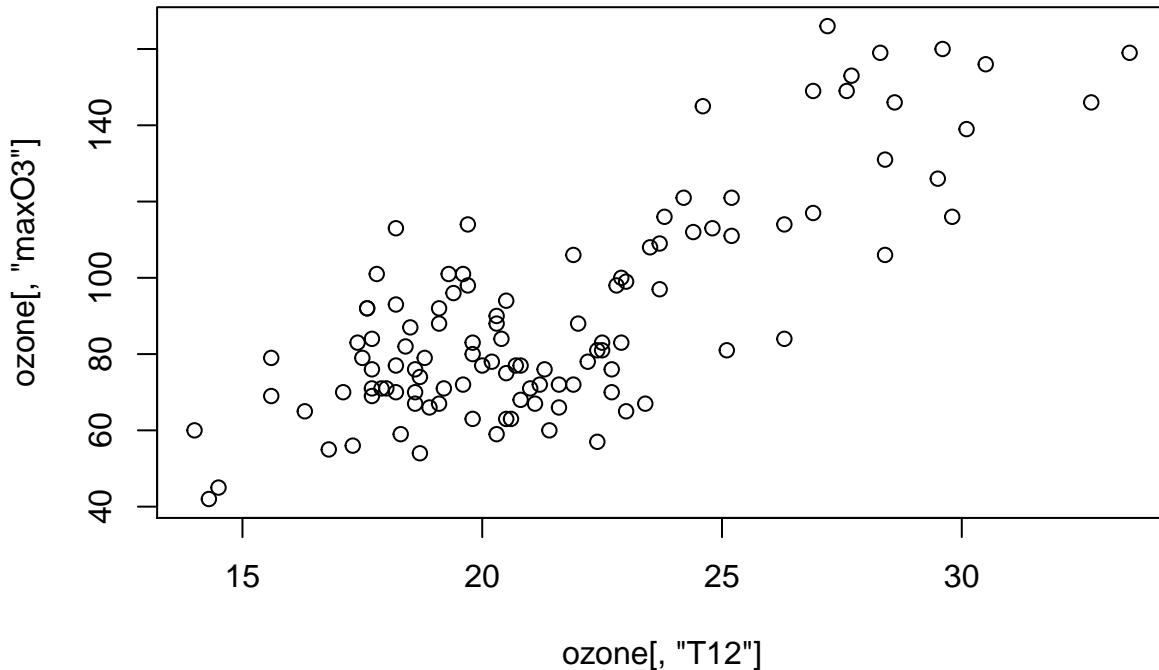
ozone <- read.table("data/ozone.txt")
summary(ozone)
      max03          T9          T12
Min.   : 42.00  Min.   :11.30  Min.   :14.00
1st Qu.: 70.75  1st Qu.:16.20  1st Qu.:18.60
Median : 81.50  Median :17.80  Median :20.55
Mean   : 90.30  Mean   :18.36  Mean   :21.53
3rd Qu.:106.00  3rd Qu.:19.93  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
```

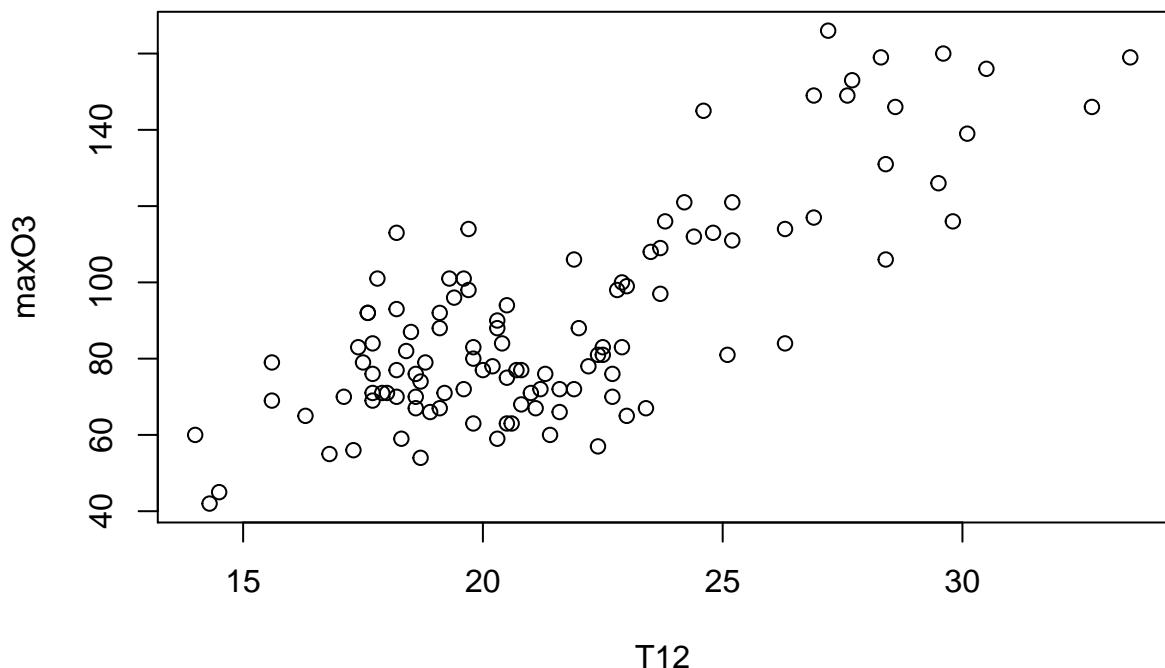
Let us start by representing two quantitative variables : maximum ozone concentrations **maxO3** in term of temperatures **T12** :

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



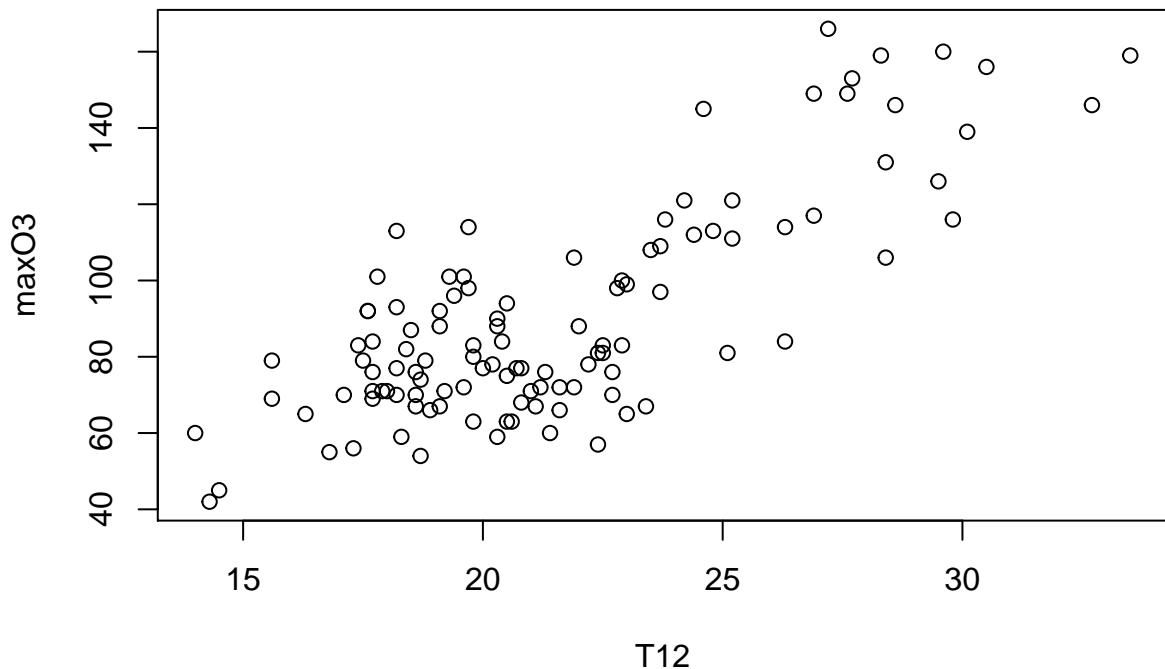
As the two variables are contained and named within the same table, a simpler syntax can be used, which automatically inserts the variables as labels for the axes :

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



We can also use (more complicated and less interesting)

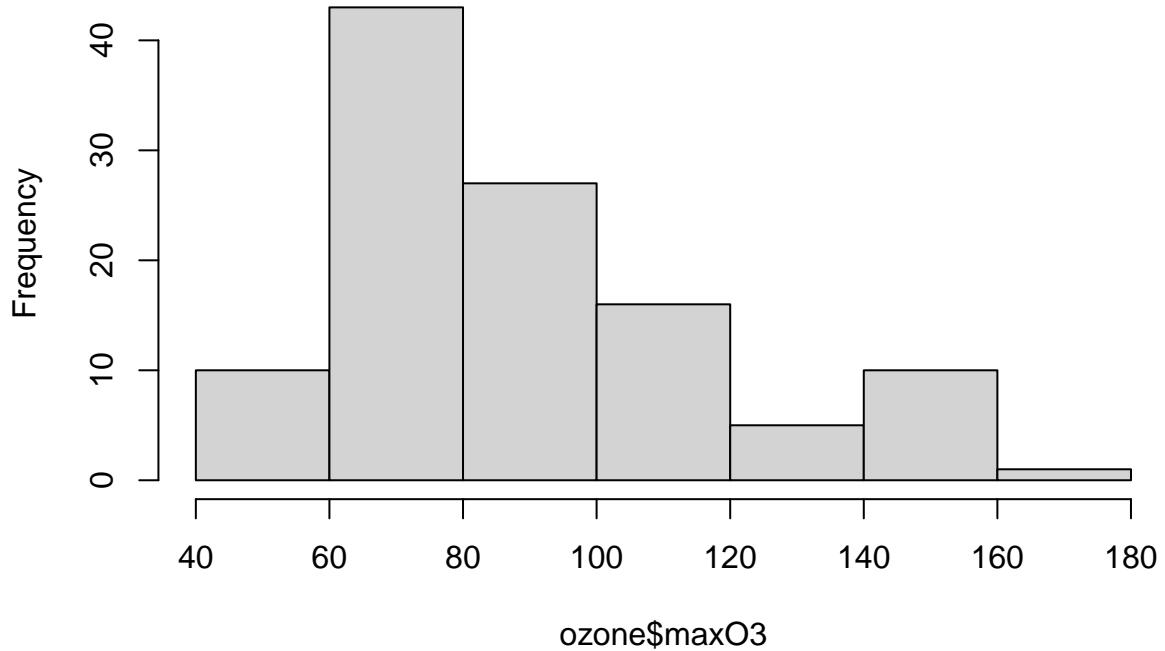
```
plot(ozone[, "T12"], ozone[, "maxO3"], xlab="T12", ylab="maxO3")
```



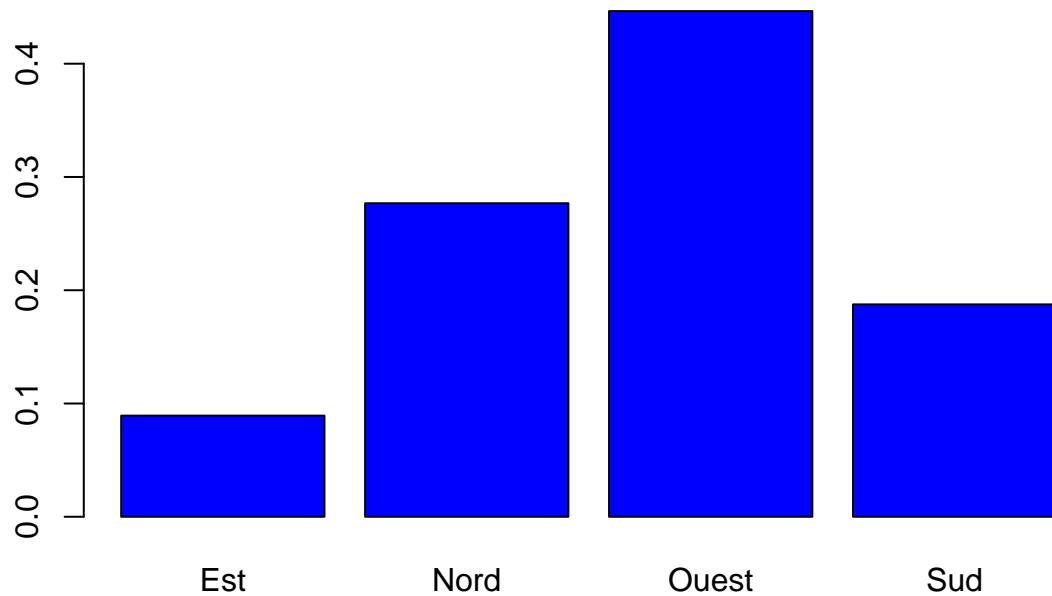
We have specific functions for each kind of charts, for instance

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

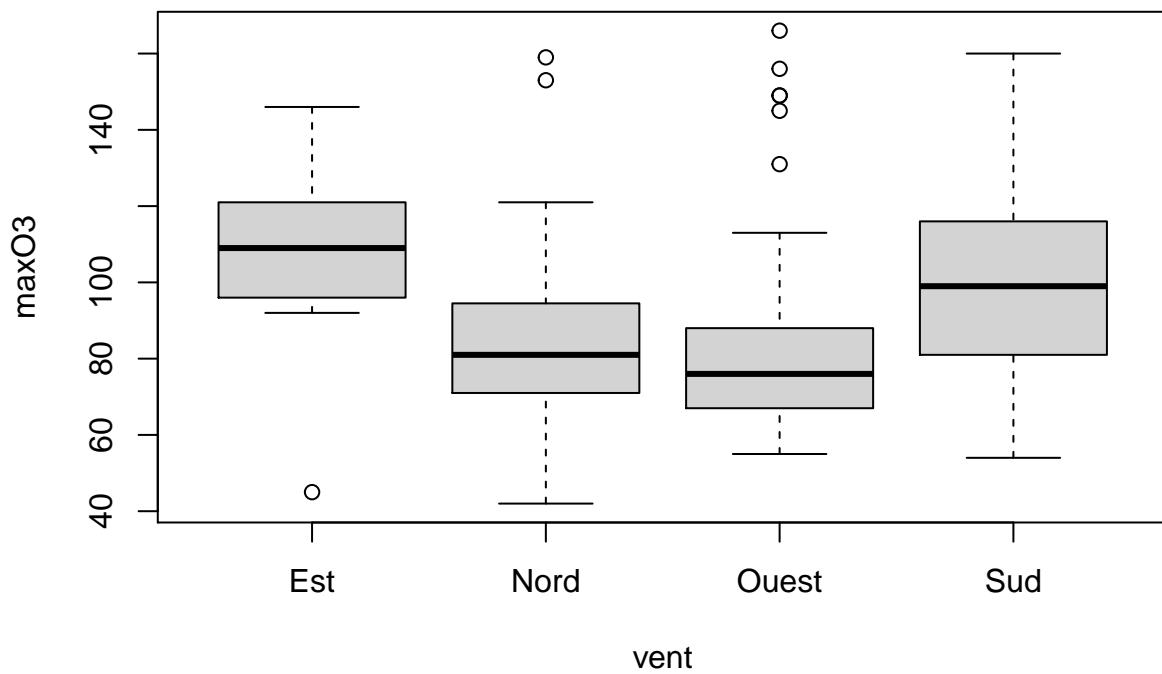
Histogram



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



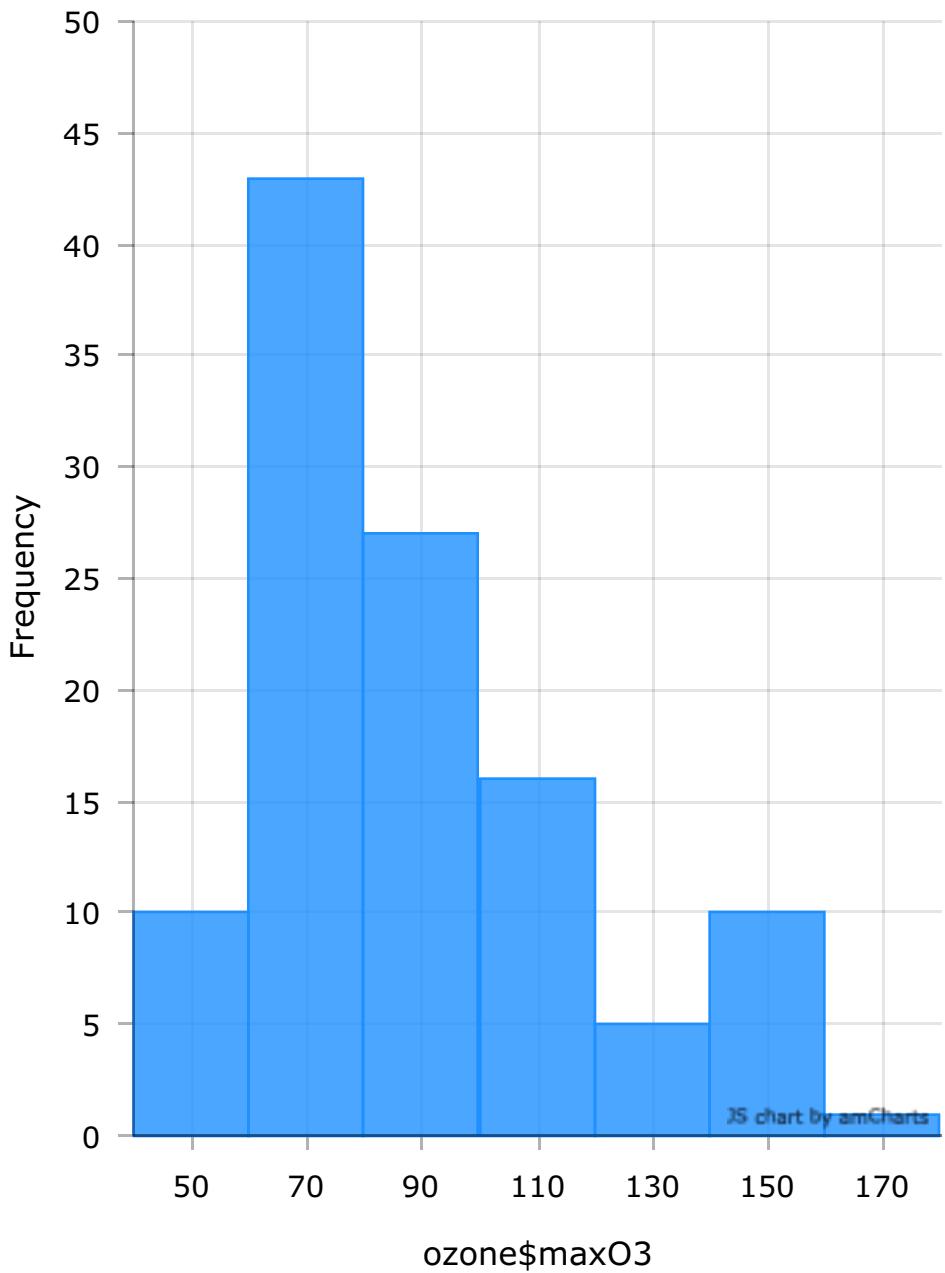
```
boxplot(max03~vent, data=ozone)
```



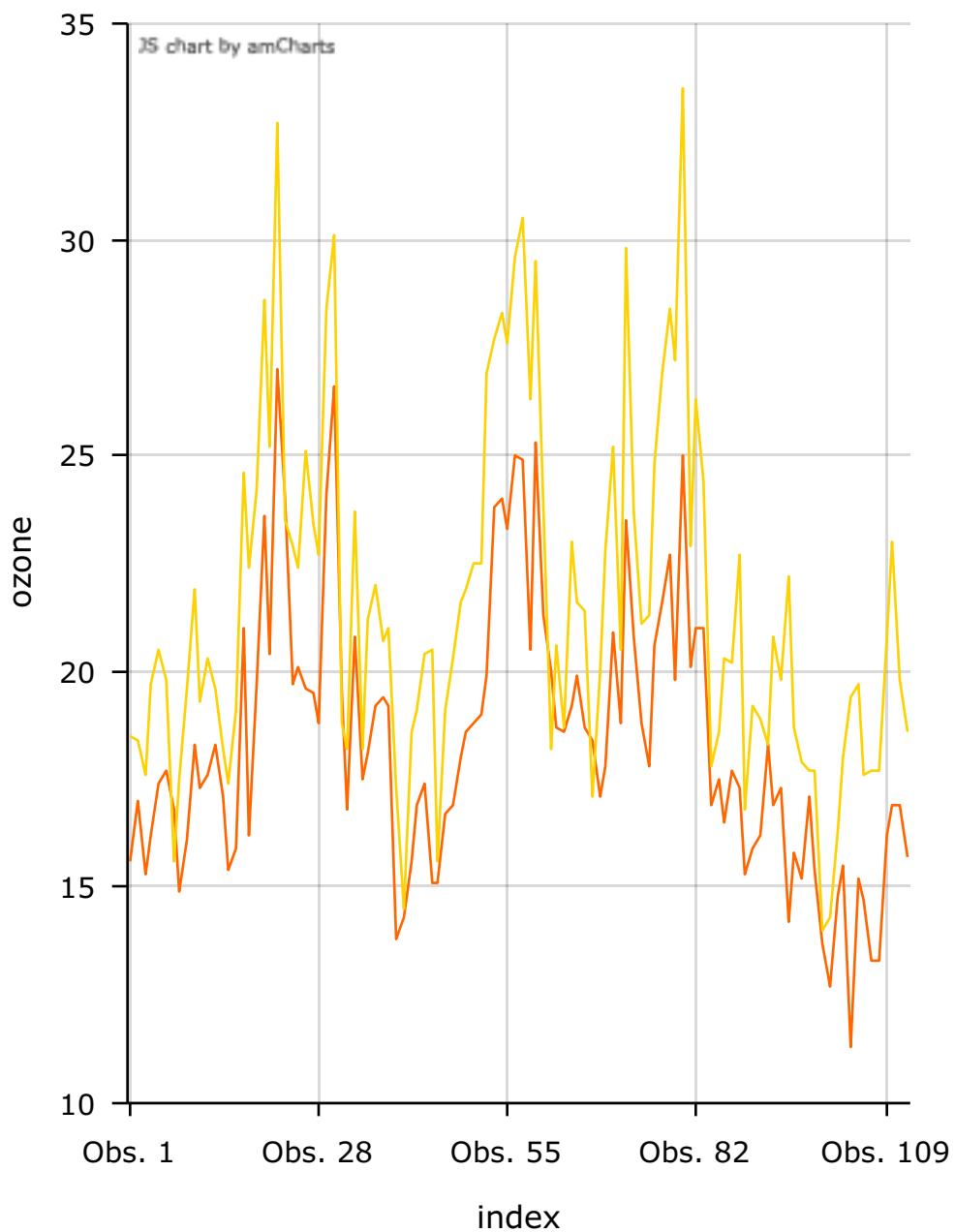
1.1.2 Interactive graphs with rAmCharts

We can use this package to obtain dynamic graphs. It is easy, we just have to use the prefix **am** before the name of the function :

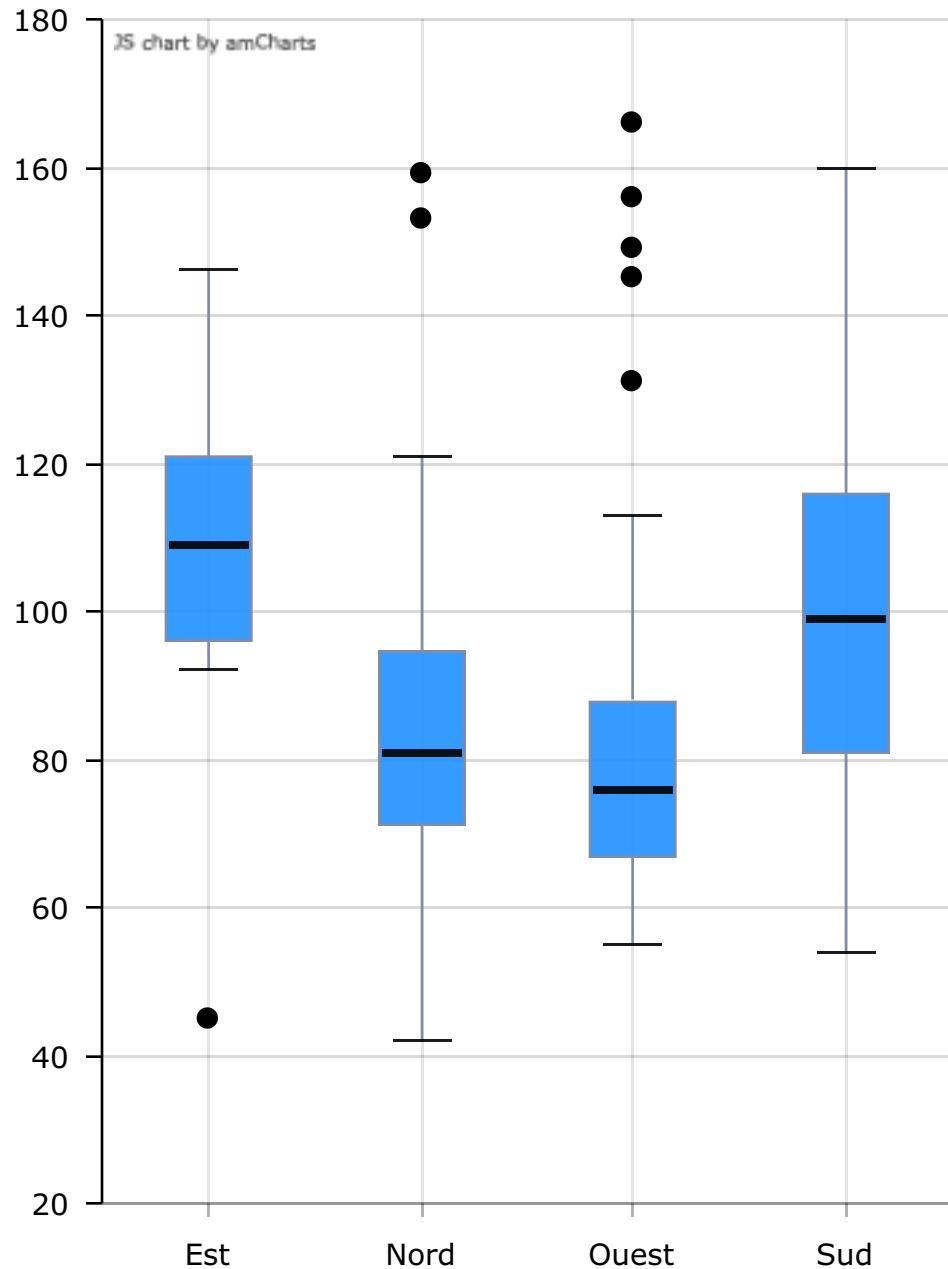
```
library(rAmCharts)
amHist(ozone$maxO3)
```



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



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



1.1.3 Some exercises

Exercise 1.1 (First chart).

1. Draw the **sine** function between 0 and 2π .
2. Add the following title : **Plot of the sine function**, you can use the function **title**.

Exercise 1.2 (Gaussian density distribution).

1. Draw the pdf (probability distribution function) of the standard Gaussian distribution between -4 and 4 (use **dnorm**).
2. Add a vertical dashed line of equation $x = 0$ (use **abline**).

3. On the same graph, draw Student's t -distribution to 5 and 30 degrees of freedom (use **dt**). Use the **lines** function and a different colour for each density.
4. Add a legend at the top left to differentiate between each distribution (use **legend**).

Exercise 1.3 (Draw sunspots).

1. Read the file **taches_solaires.csv** series which details, data by data, the relative number of sunspots.
2. Using the **cut_interval** function from the *tidyverse* package, create a qualitative variable which separates the dates into 8 intervals. We call **period** this variable.
3. Use the following levels for **period**

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

4. Expliquer la sortie de la fonction
5. Visualize the sunspot series with a diffrect color for the 8 periods.

Exercise 1.4 (Layout).

We consider the **ozone** dataset. With the **layout** function, split the window into two lines with

1. the scatter plot **maxO3 vs T12** on the first line ;
- 2 graphs on the second lines : histogram of **T12** et boxplot of **maxO3**.

1.2 Ggplot2 grammar

ggplot2 is a plotting system for R based on the grammar of graphics (as **dplyr** to manipulate data). We can find documentation at <http://ggplot2.org> and <https://ggplot2-book.org>.

1.2.1 First charts with ggplot2

We consider a subsample of the **diamond** dataset from the package **ggplot2** (or **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)
```

A graph is always defined from many **layers**. We usually have to specify :

- the data
- the variables we want to plot
- the type of representation (scatterplot, boxplot...).

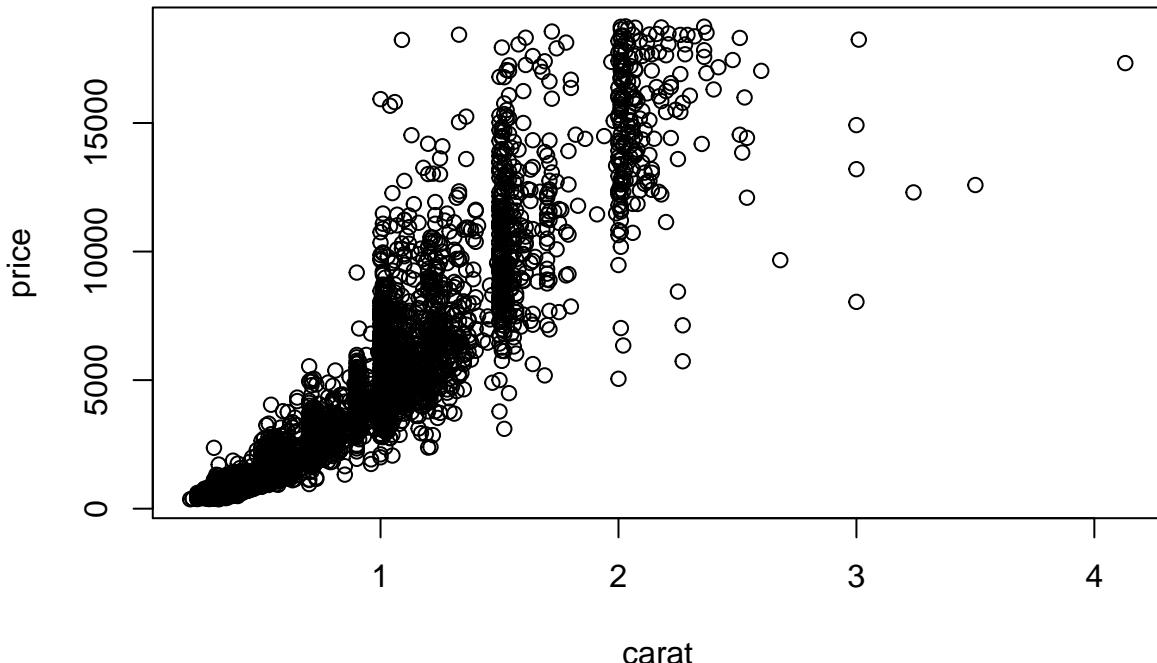
Gplot graphs are defined from these layers. We have to specify each layer with a ggplot function :

- the data with **ggplot**
- the variables with **aes** (aesthetics)
- the type of representation with **geom_**

These layers are gathered with the **+** operator.

The scatterplot **carat** vs **price** is obtained with the **plot** function with

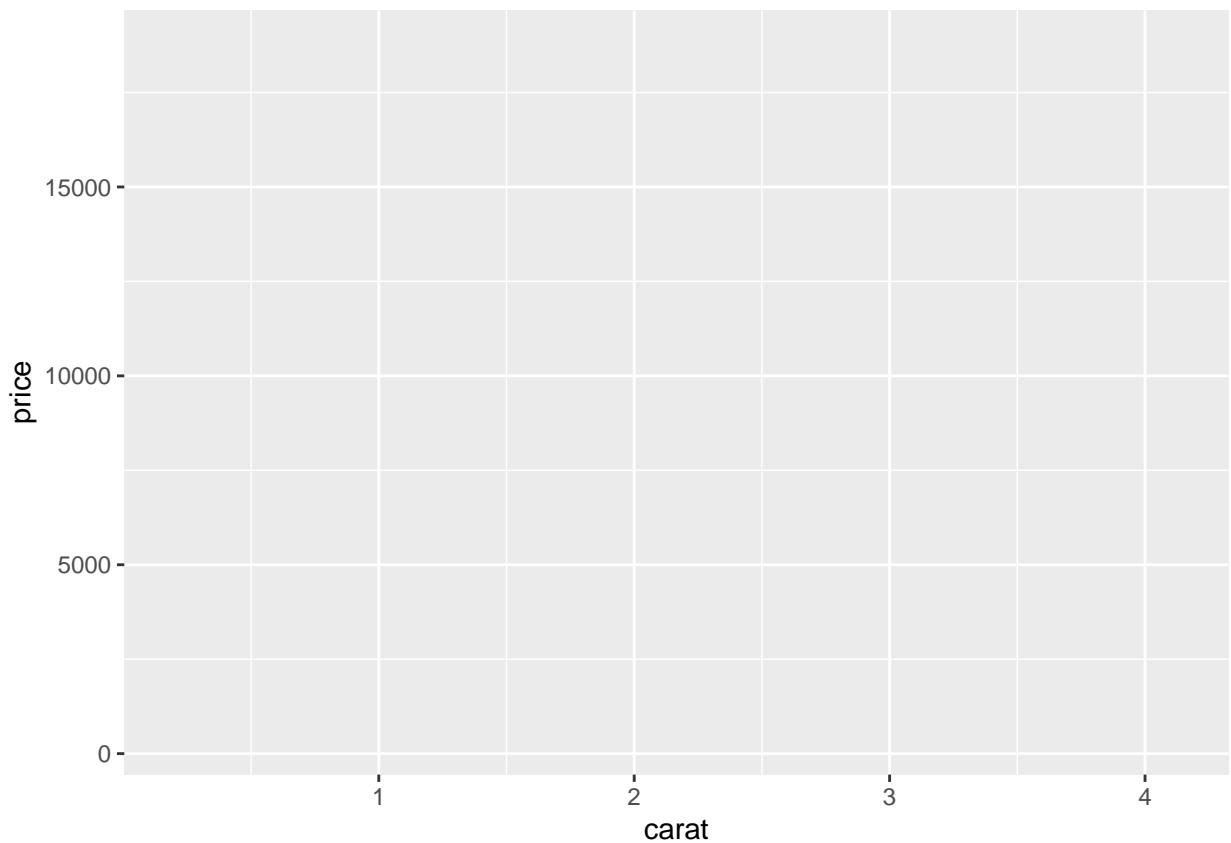
```
plot(price~carat,data=diamonds2)
```

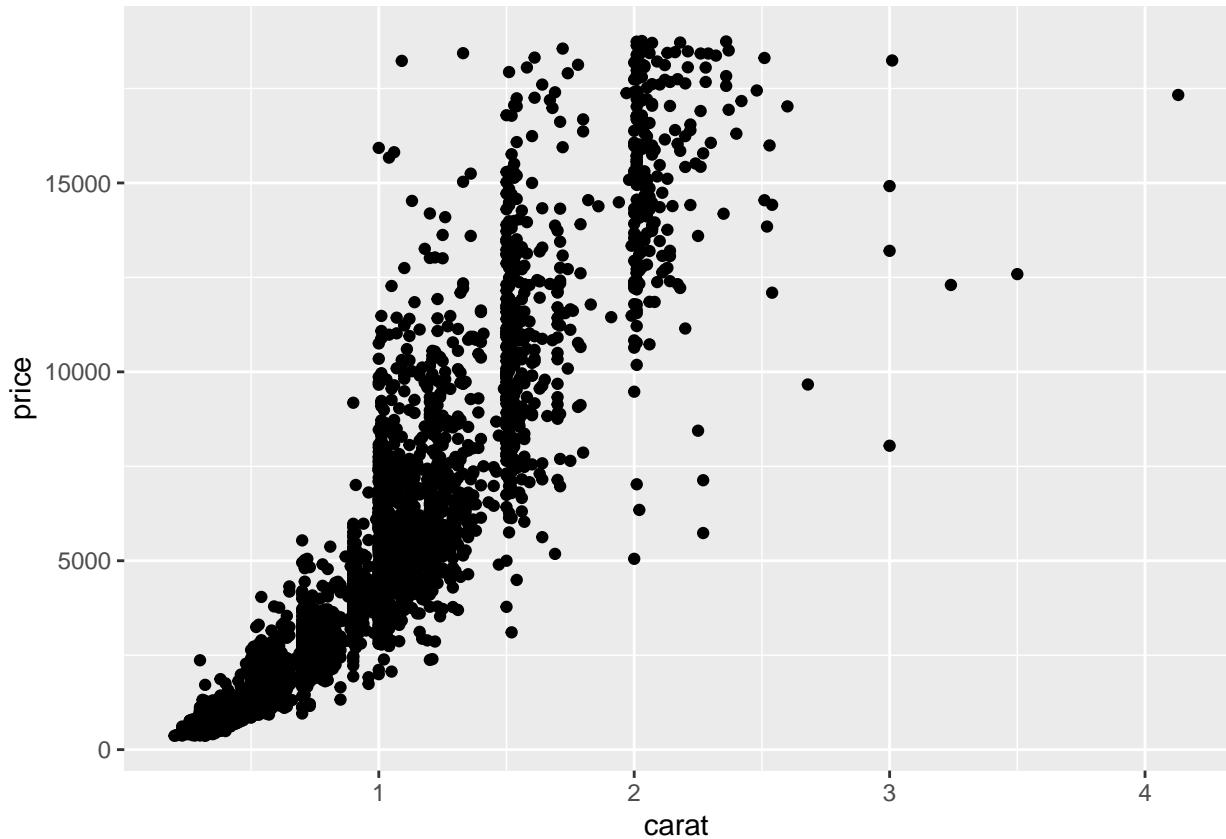


With **ggplot**, we use

```
ggplot(diamonds2) #nothing
```

```
ggplot(diamonds2)+aes(x=carat,y=price) #nothing
```





Exercise 1.5 (Histograms and barplot with ggplot2).

1. Draw the histogram of **carat** (use **geom_histogram**)
2. Draw the histogram of **carat** with 10 bins (**help(geom_histogram)**)
3. Draw the barplot for the variable **cut** (use **geom_bar**)

To summarize, ggplot syntax is based on independent elements. These elements define the **grammar** of **ggplot**. Main elements of this grammar include :

- **Data (ggplot)** : the dataset, it should be a dataframe or a **tibble**.
- **Aesthetics (aes)** : to describe the way that variables in the data are mapped. We have to specify **all** variables used to build the graph in **aes**.
- **Geometrics (geom_...)** : to control the type of plot (scatterplot, histogram...)
- **Statistics (stat_...)** : to describe transformations of the data needed for the chart.
- **Scales (scale_...)** : to control the mapping from data to aesthetic attributes (change of colors, options for the axis...)

All these elements are combined with a **+**.

1.2.2 Data et aesthetics

We have two use these elements for all graphs. **ggplot** verb is used to indicate the dataset. If the syntax is efficient, you will never have to use the name of the dataset to obtain the required graph. **aes** verb is used to specify the variables we want to visualize. For instance, for a scatterplot **price vs carat**, the syntax should start with

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

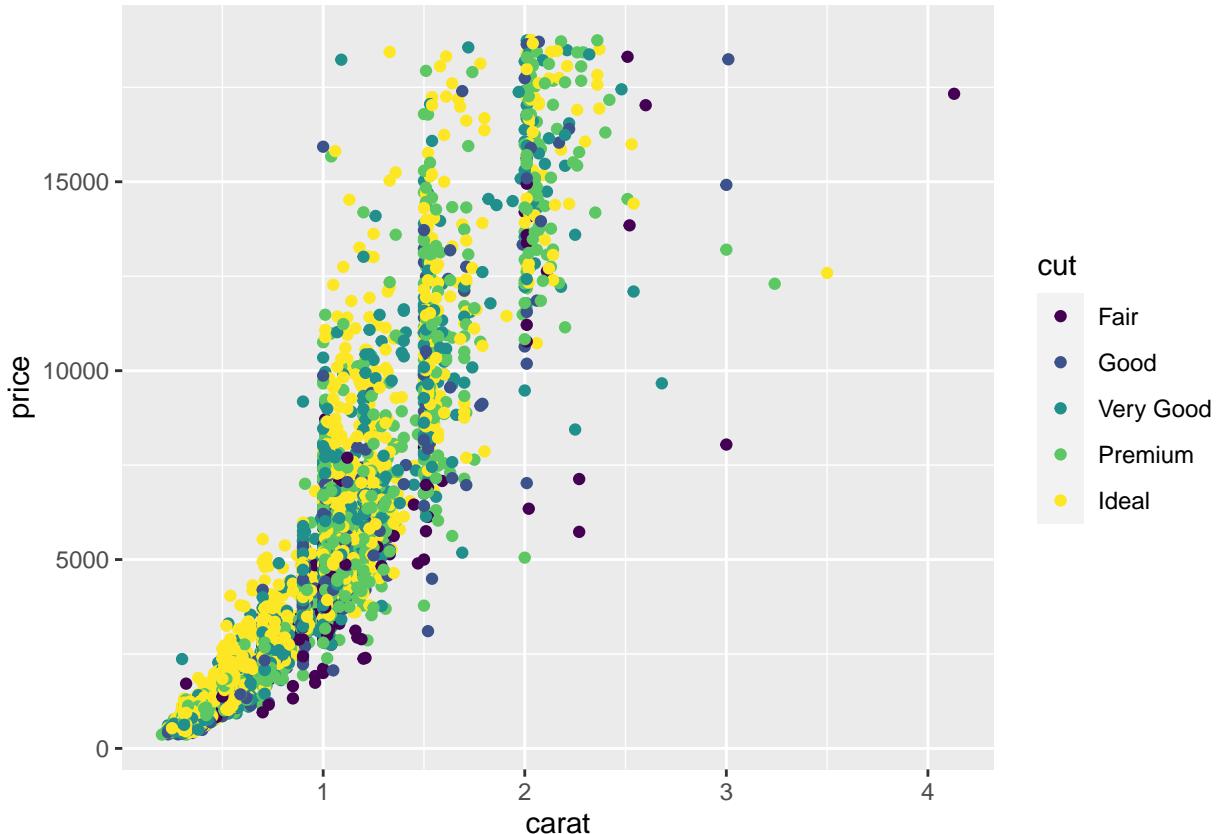
`aes` also possesses options such as `color`, `size`, `fill`. We use these options as soon as a color or a size is defined from a **variable of the dataset**. For instance

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

1.2.3 Geometrics

To draw a graph, we need to specify the type of representation. We have to use **geometrics** to do that. For a scatter plot, we use `geom_point` :

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



Observe that `ggplot` adds the legend automatically. Examples of **geometrics** are described here :

TABLE 1 : Main geometrics

Geom	Description	Aesthetics
geom_point()	Scatter plot	x, y, shape, fill
geom_line()	Line (ordered according to x)	x, y, linetype
geom_abline()	Line	slope, intercept
geom_path()	Line (ordered according to the index)	x, y, linetype
geom_text()	Text	x, y, label, hjust, vjust
geom_rect()	Rectangle	xmin, xmax, ymin, ymax, fill, linetype
geom_polygon()	Polygons	x, y, fill, linetype
geom_segment()	Segment	x, y, xend, yend, fill, linetype
geom_bar()	Barplot	x, fill, linetype, weight
geom_histogram()	Histogram	x, fill, linetype, weight
geom_boxplot()	Boxplots	x, y, fill, weight

Geom	Description	Aesthetics
geom_density()	Density	x, y, fill, linetype
geom_contour()	Contour lines	x, y, fill, linetype
geom_smooth()	Smoothers (linear or non linear)	x, y, fill, linetype
All		color, size, group

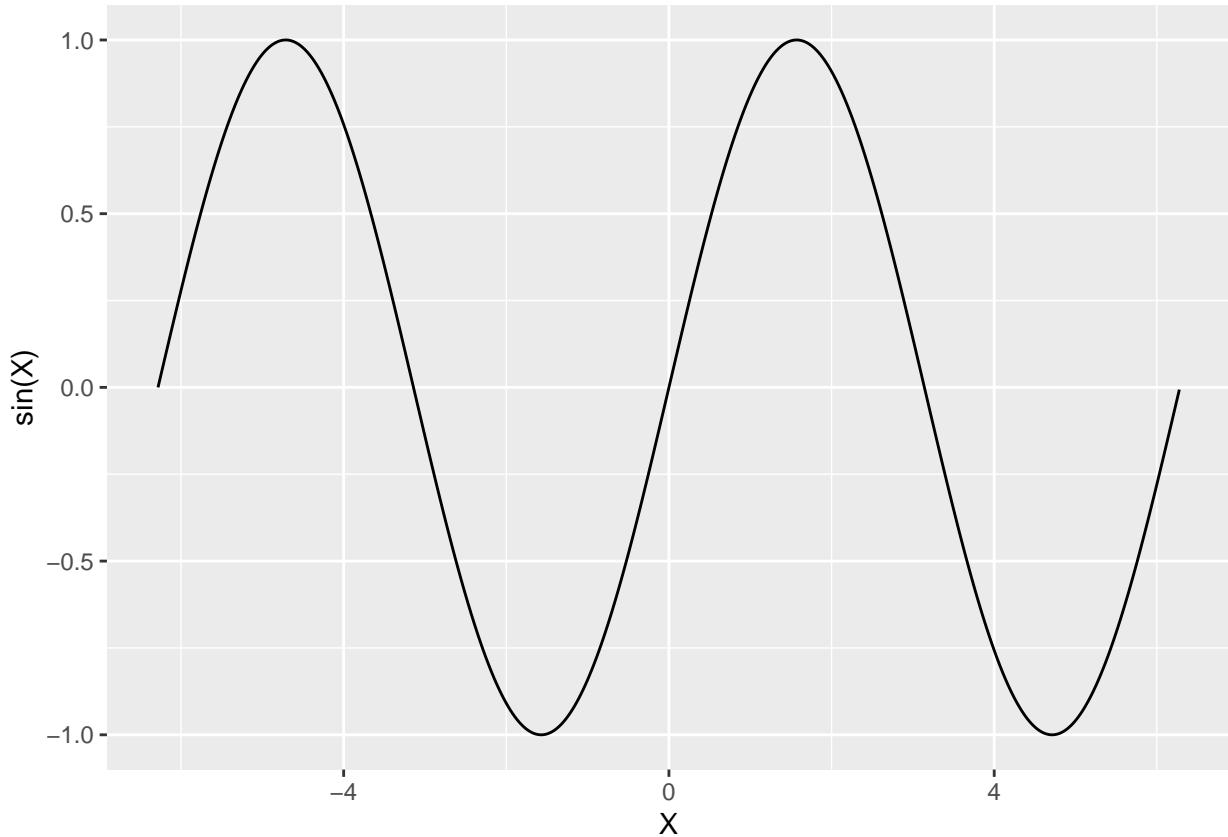
Exercise 1.6 (Barplots).

1. Draw the barplot of **cut** (with blue bars).
2. Draw the barplot of **cut** with one color for each modality of cut. Add a legend.
3. Draw the barplot of **cut** with one color **of your choice** for each modality of cut (and without lengend since it is not useful here).

1.2.4 Statistics

Many graphs need transformations the data to make the representation. It is for example the case for histogram or barplot : we have to calculate the heights for rectangles or bars. These heigths are not explicitely specified in the dataset. Simple transformations can be obtained quickly with **ggplot2**. For instance we can draw the sine function with

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

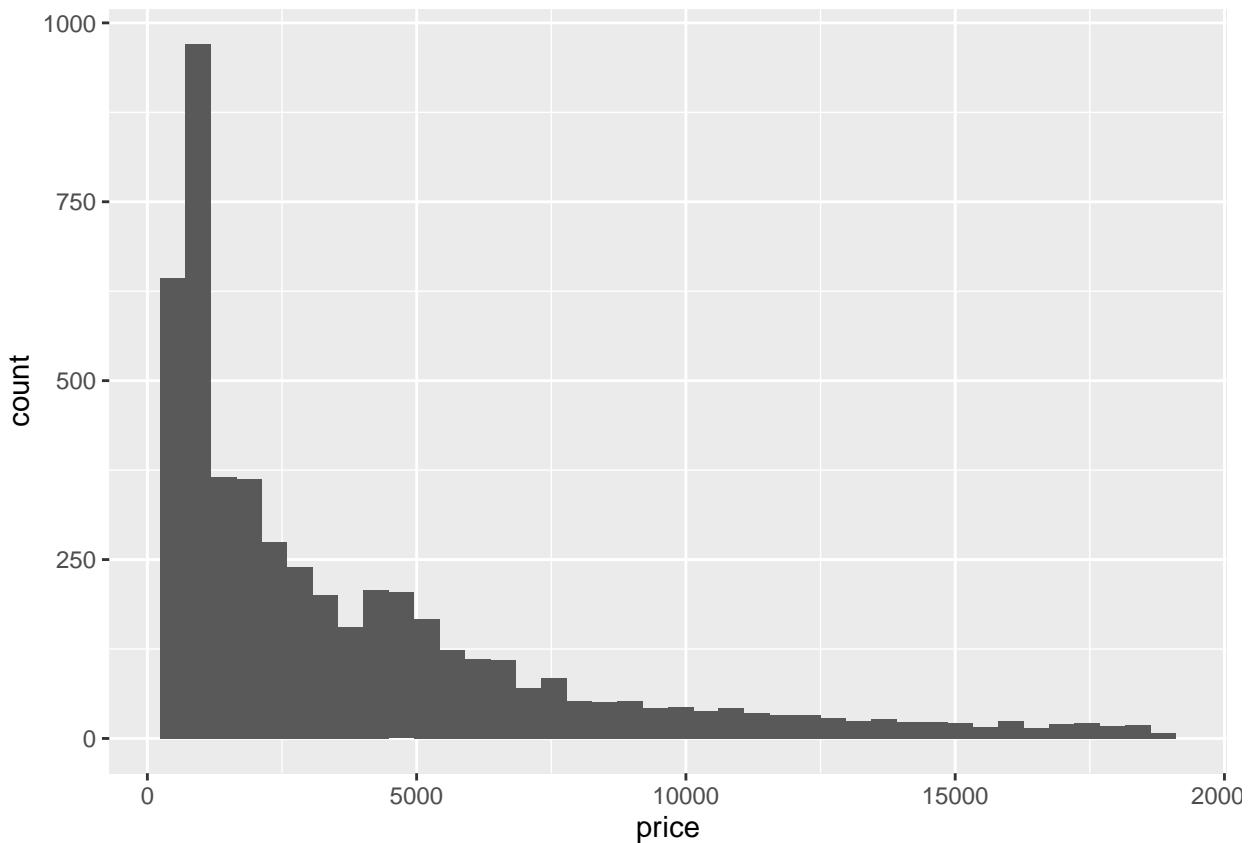


The sine transformation is indicated in **aes**. For more complex transformations, we have to used **statistics**. A **stat** function takes a dataset as input and returns another dataset as output, and so a stat can add new variables to the original dataset. It is then possible to map aesthetics to these new variables. For example, **stat_bin** function (used to build histograms) creates the following variables :

- **count**, the number of observations in each bin ;
- **density**, the density of observations in each bin (percentage of total / bar width) ;
- **x**, the center of the bin.

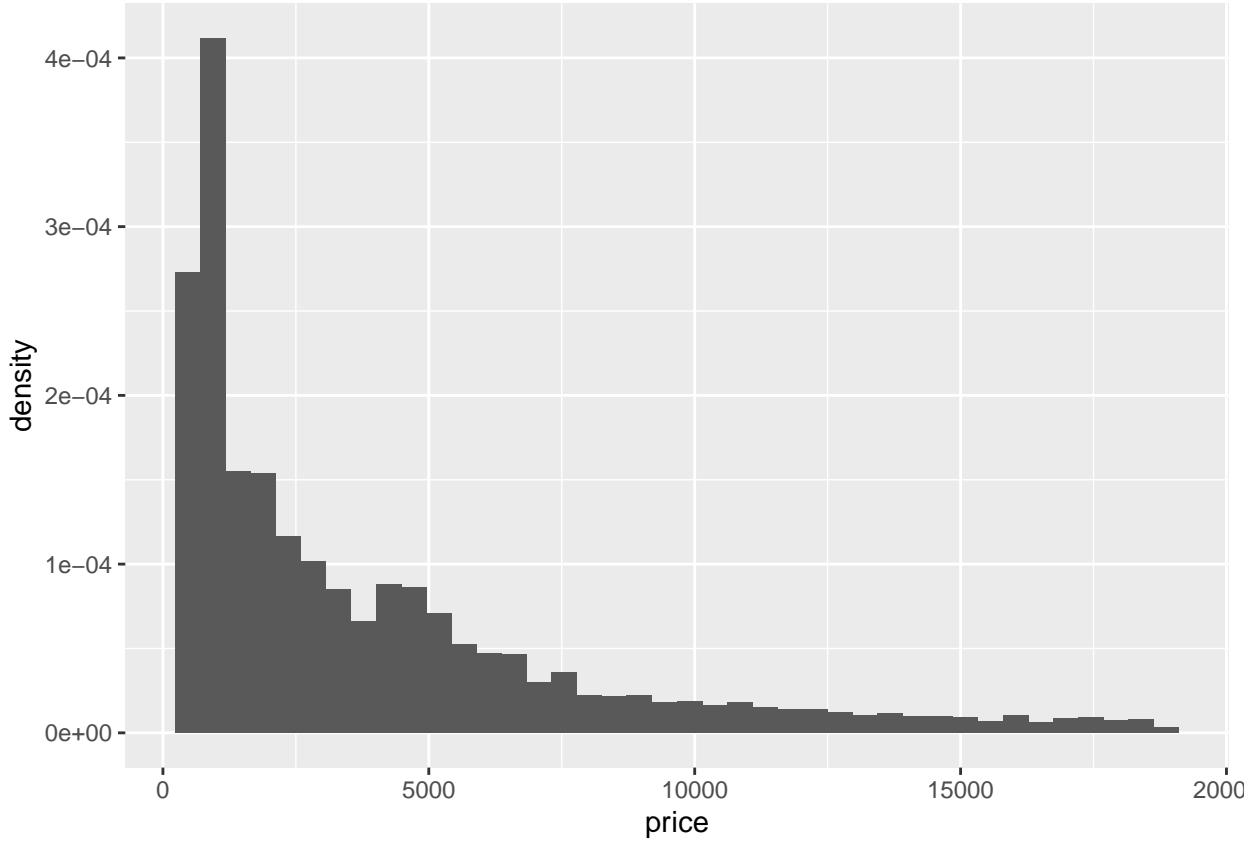
By default *geom_histogram* represents on the *y*-axis the number of observations in each bin (the output **count**).

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



If we are interested by another output of **stat_bin**, the density for instance, we have to use

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



ggplot proposes another way to make the representations : we can use *stat_* instead of *geom_*. Formally, each *stat_...* function has a *geom_...* and each *geom_...* has a *stat_....*. We can draw the same graph as above with

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

Here are some examples of **stat functions** :

TABLE 2 : Examples of statistics.

Stat	Description	Parameters
stat_identity()	No transformation	
stat_bin()	Count	binwidth, origin
stat_density()	Density	adjust, kernel
stat_smooth()	Smoothen	method, se
stat_boxplot()	Boxplot	coef

stat and *geom* are not always easy to combine. For beginners, we recommend to use *geom* for beginners.

Exercise 1.7 (A "very simple" barplot...).

We consider a color variable *X* with probability distribution

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

Draw the barplot of this distribution.

Exercise 1.8 (Lissage).

1. Visualize the non linear smoother for the variable *price* in terms of *carat*. You can use *geom_smooth* or *stat_smooth*.

2. Same question but use a dotted line instead of a solid line.

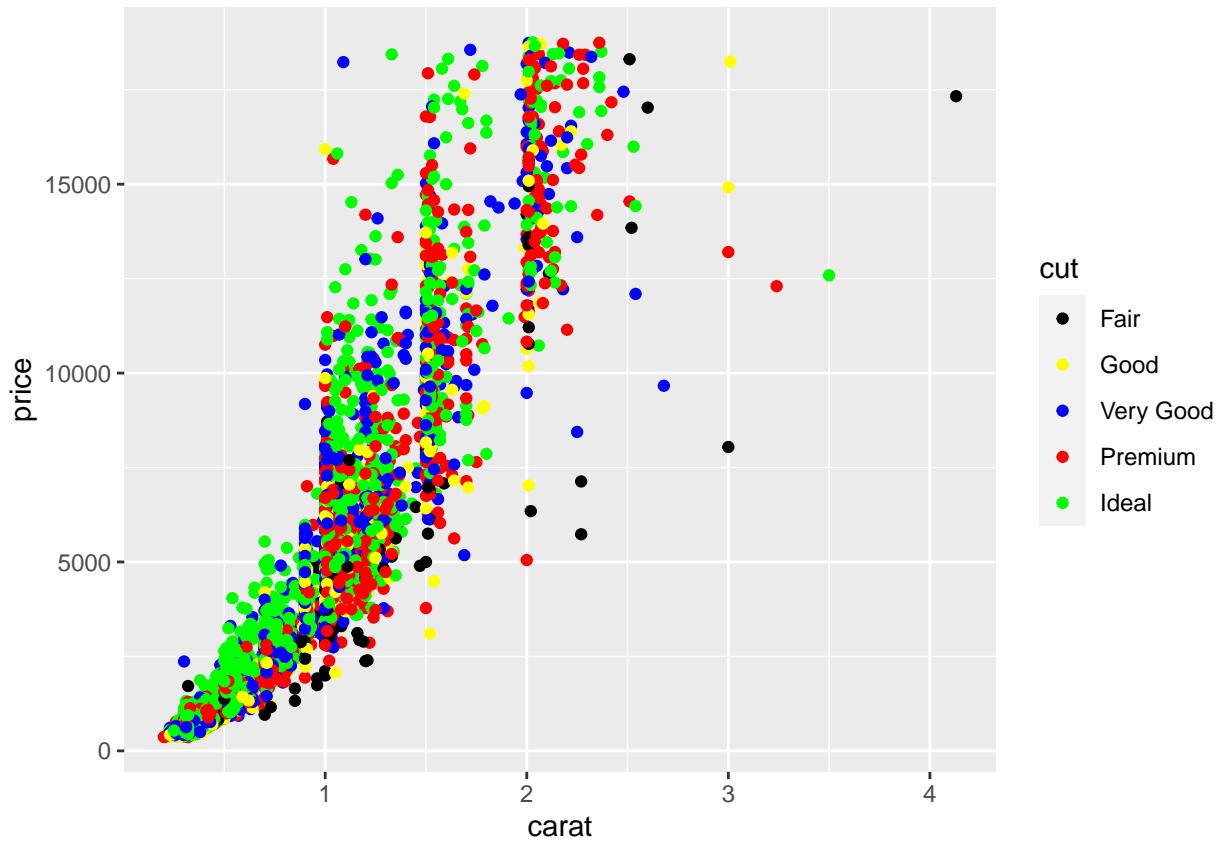
1.2.5 Scales

Scales control the mapping from data to aesthetic attributes (change of colors, sizes...). We generally use this element at the end of the process to refine the graph. Scales are defined as follows :

- begin with `scale_`
- add the aesthetics we want to change (`color`, `fill`, `x_`)
- end with the name of the scale (`manual`, `identity`...)

For instance,

```
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"))
```



Here are the main scales :

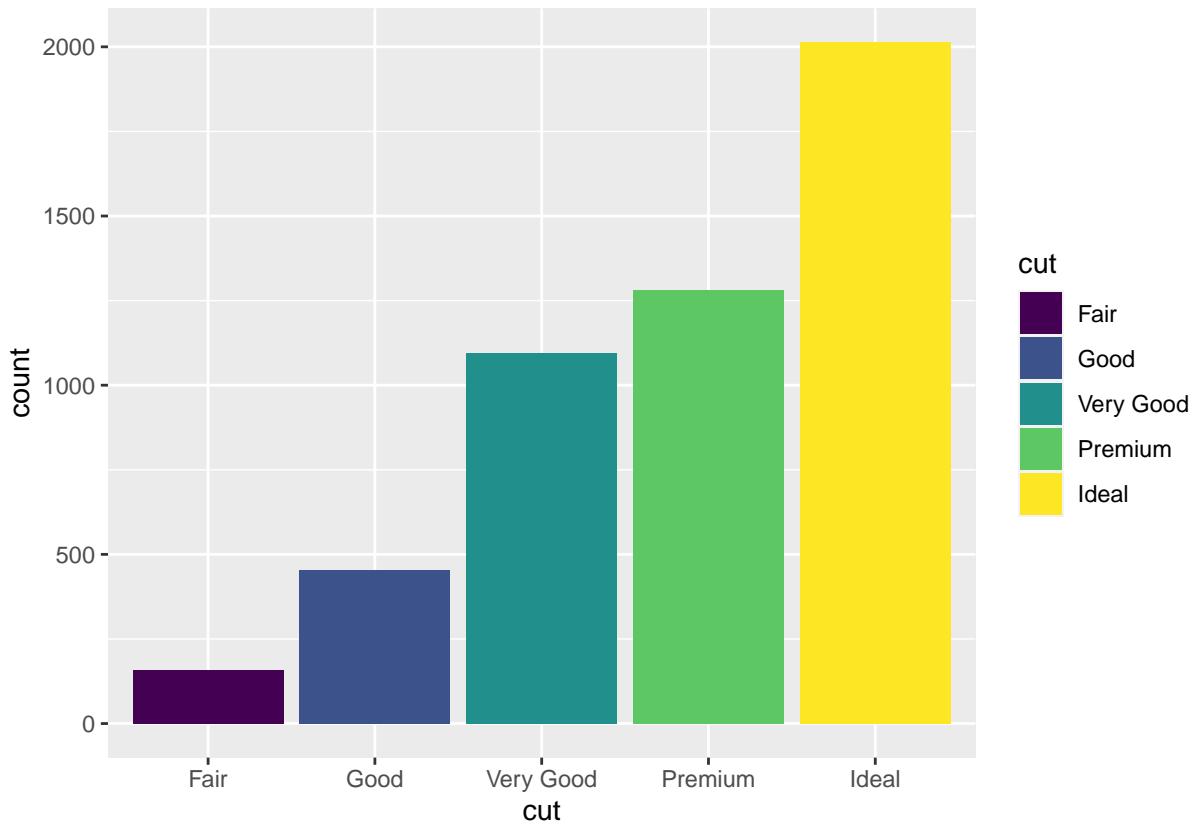
TABLE 3 : Main scales

aes	Discrete	Continuous
Color (color et fill)	brewer	gradient
-	grey	gradient2
-	hue	gradientn
-	identity	
-	manual	
Position (x and y)	discrete	continuous
-		date

aes	Discrete	Continuous
Shape	shape	
-	identity	
-	manual	
Size	identity	size
-	manual	

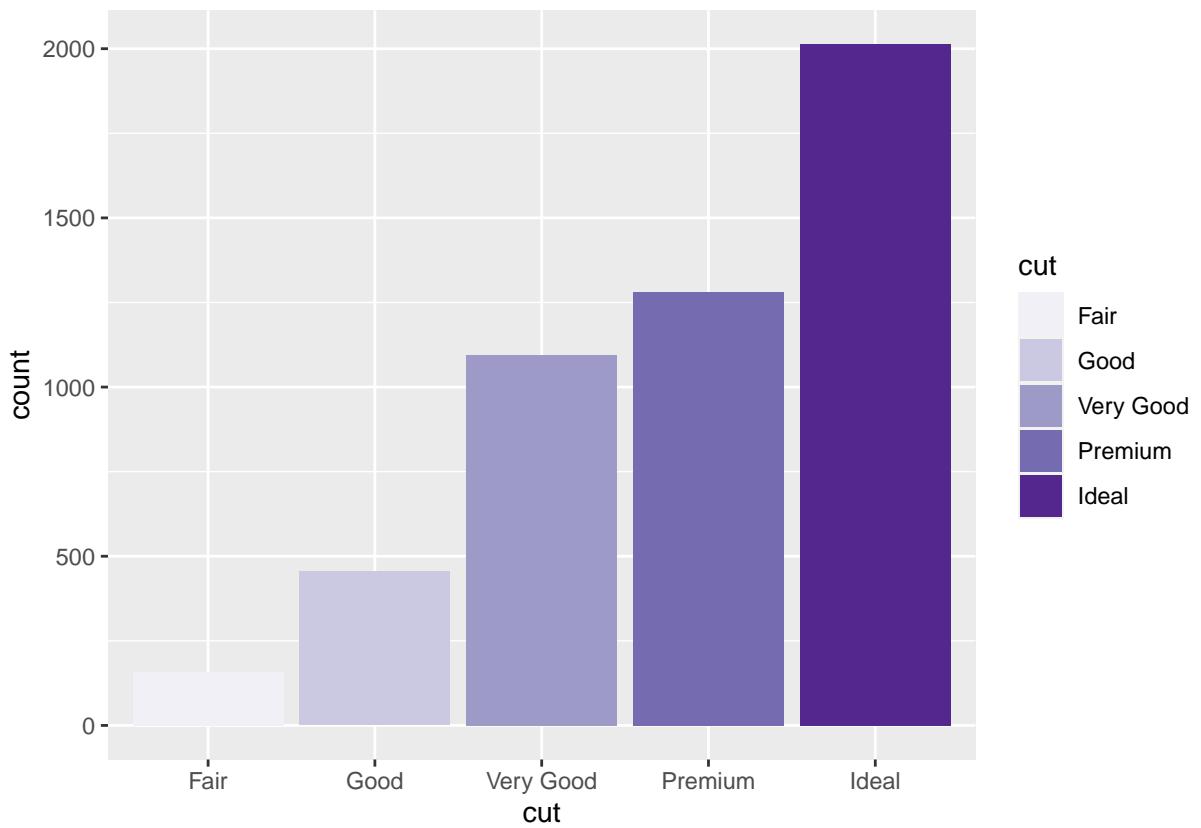
We now present some examples on **scales** :

```
— color in a barplot
p1 <- ggplot(diamonds2)+aes(x=cut)+geom_bar(aes(fill=cut))
p1
```



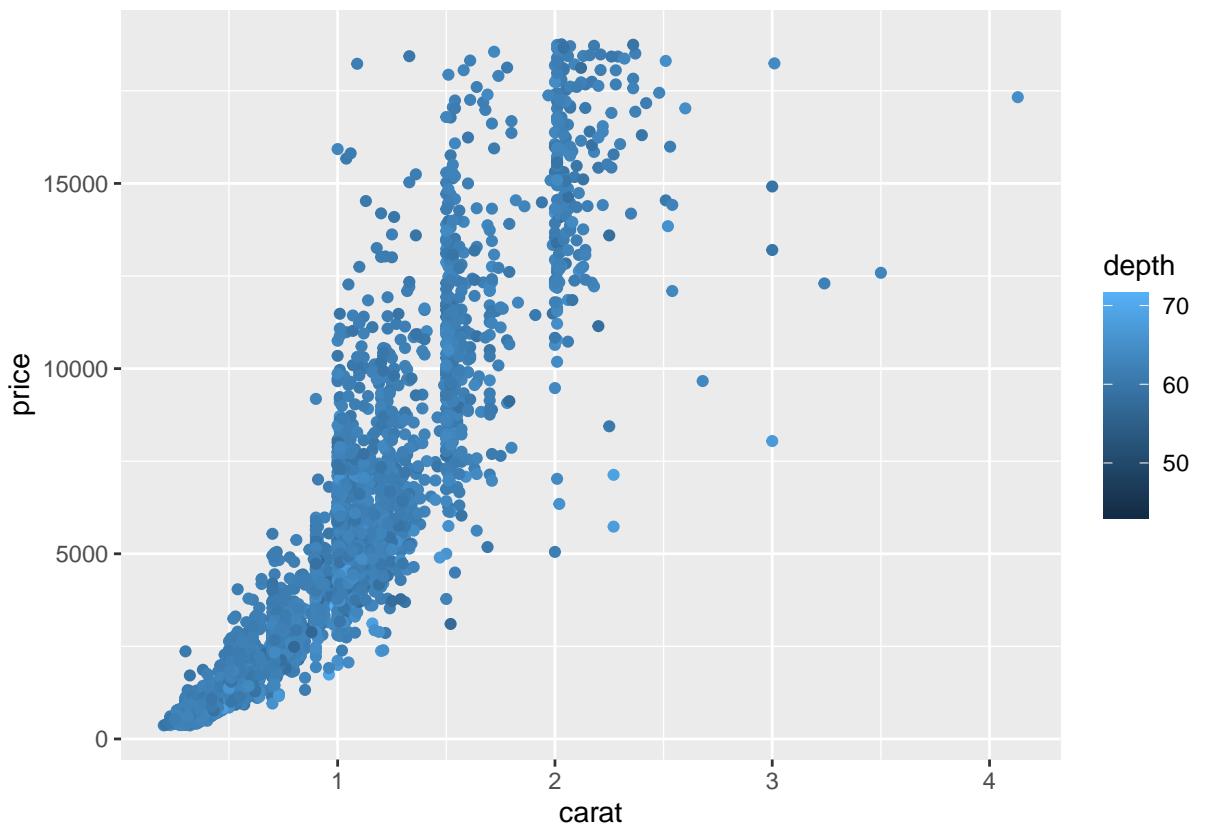
We change colors by using the palette **Purples** :

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



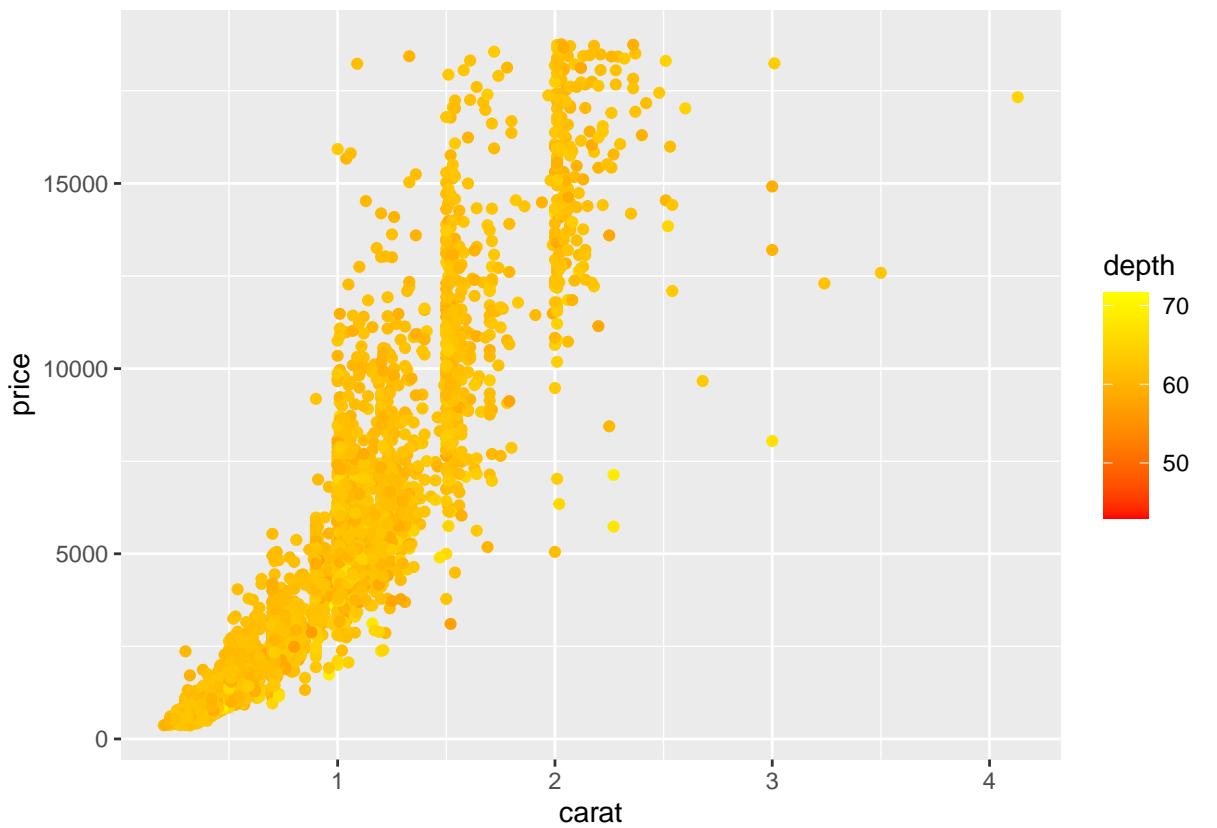
— Gradient color for a scatter plot :

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



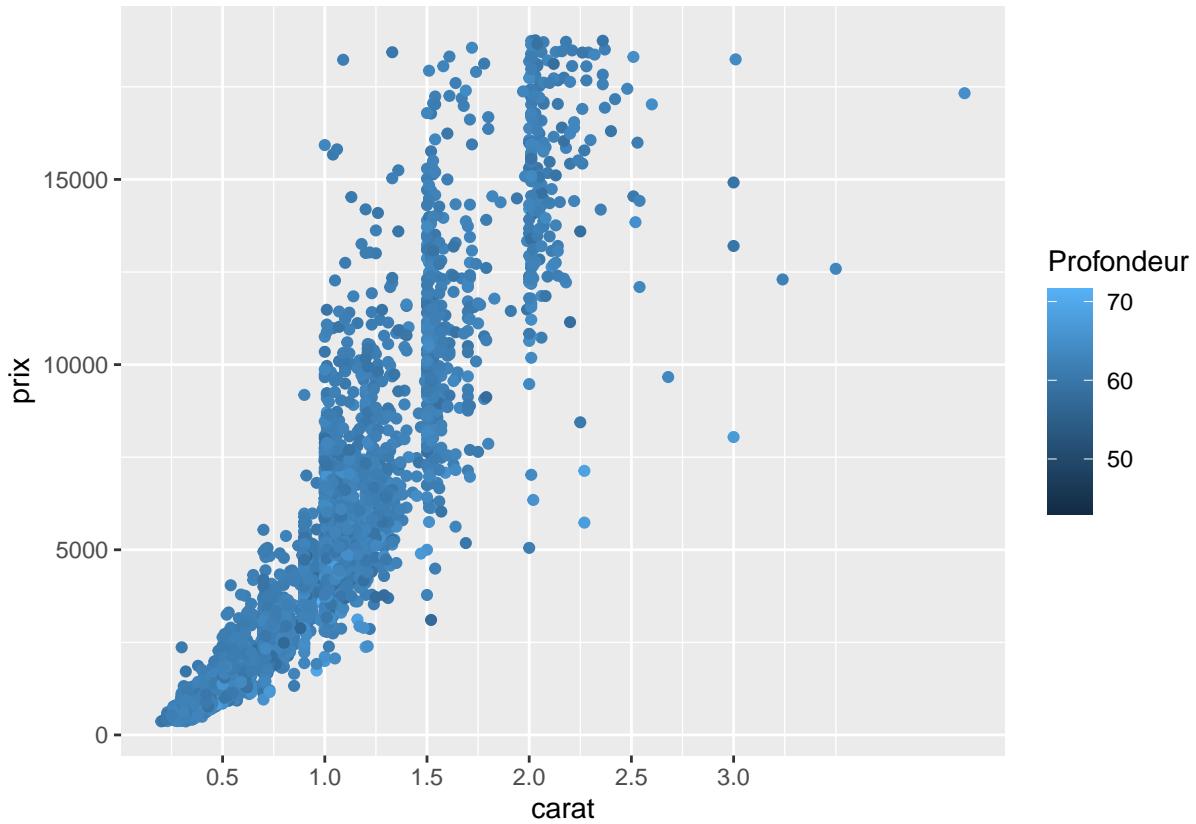
We change the gradient color

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



— Change on the axis

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



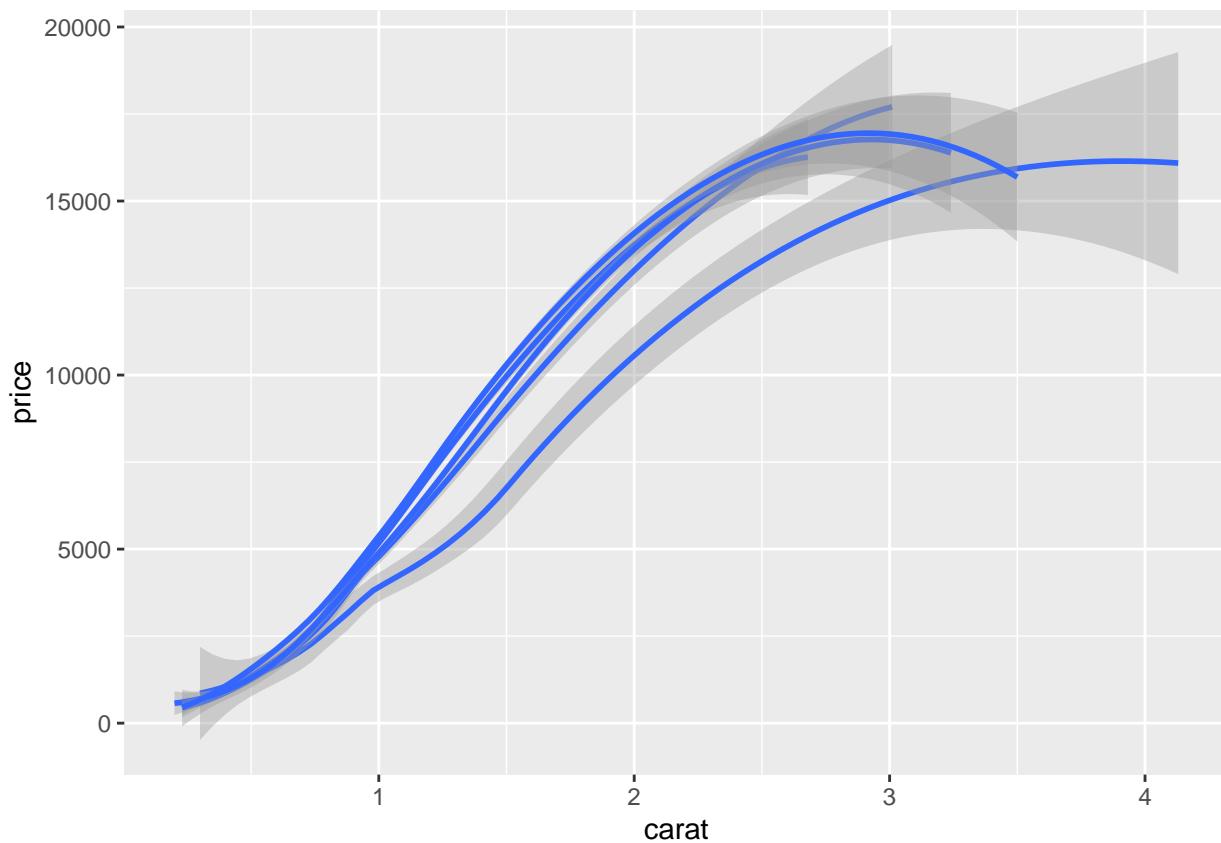
1.2.6 Group and facets

ggplot allows to make representations for subgroup of individuals. We can proceed in two ways :

- to represent subgroup on the same graph => **group** in **aes**
- to represent subgroup on the different graphs => **facets**

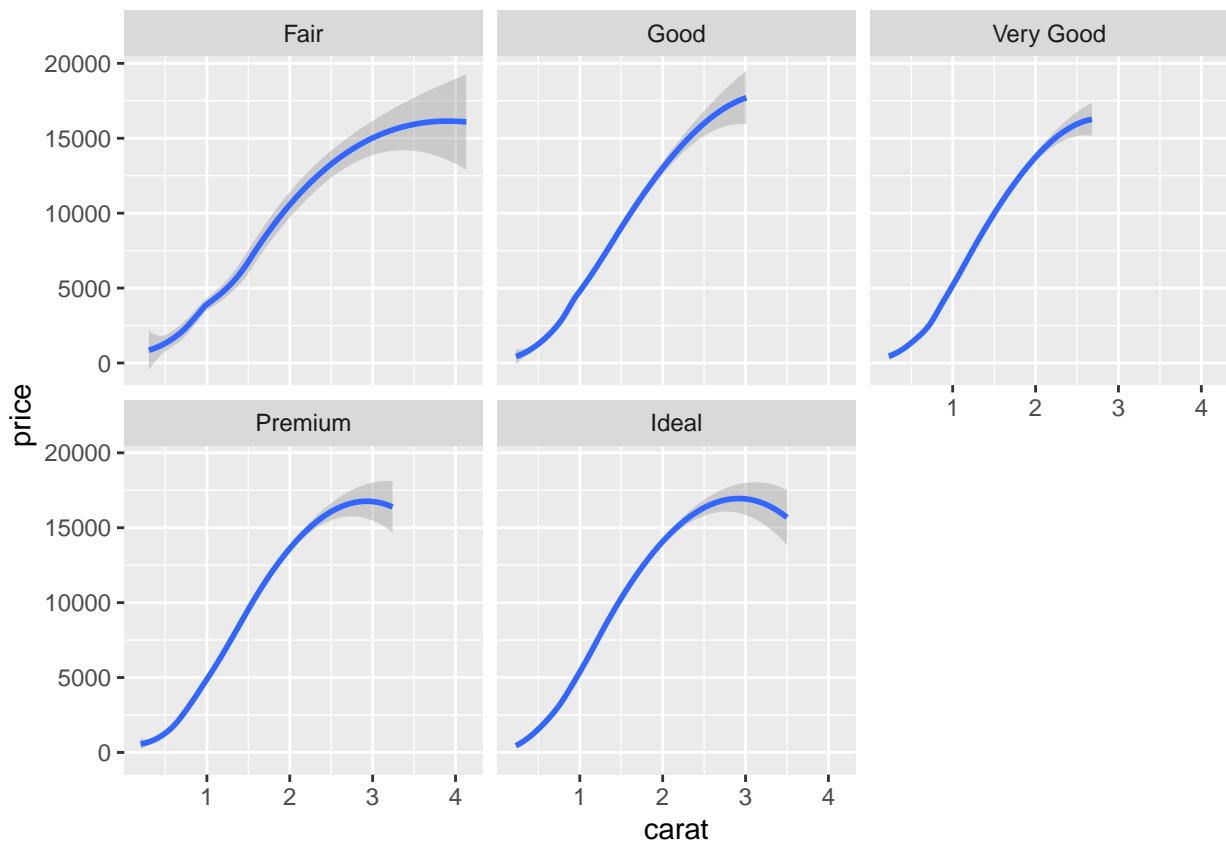
We represent (on the same graph) the smoother **price vs carat** for each modality of **cut** with

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

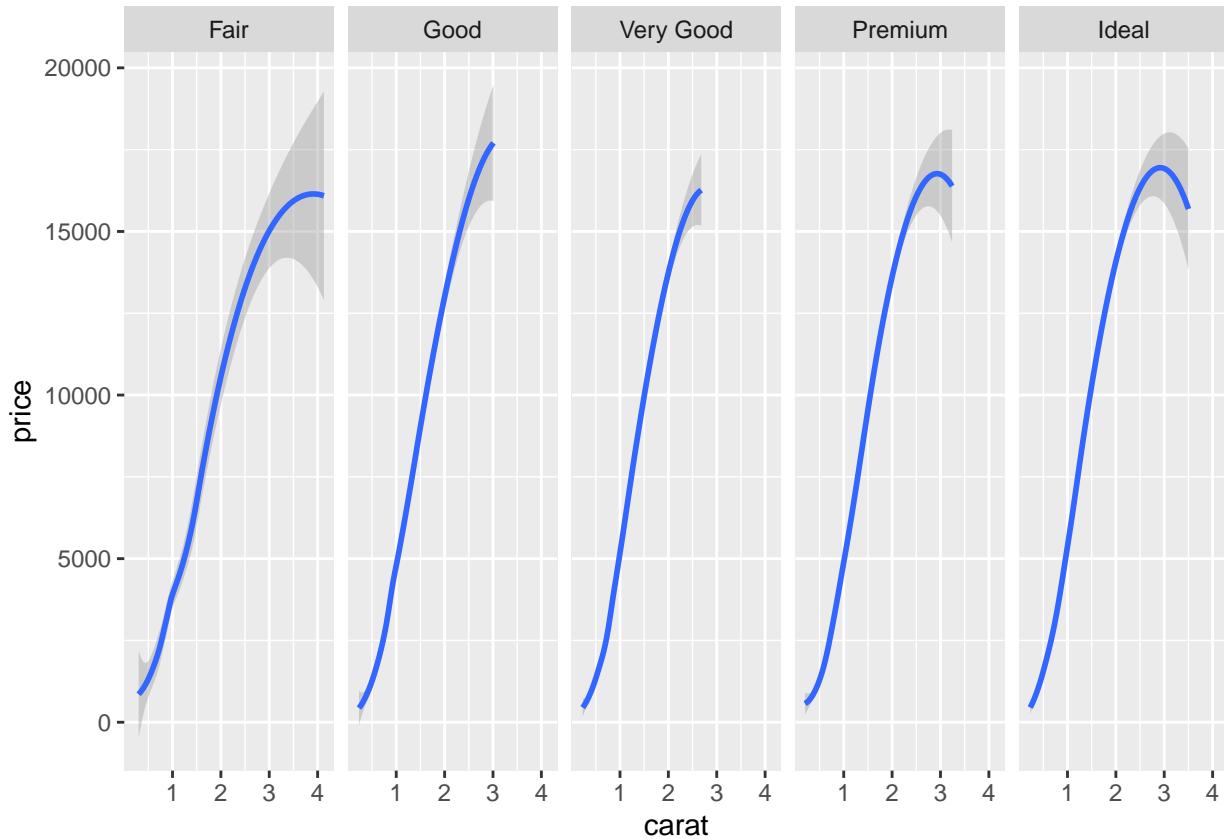


To obtain the representation on many graphs, we use

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

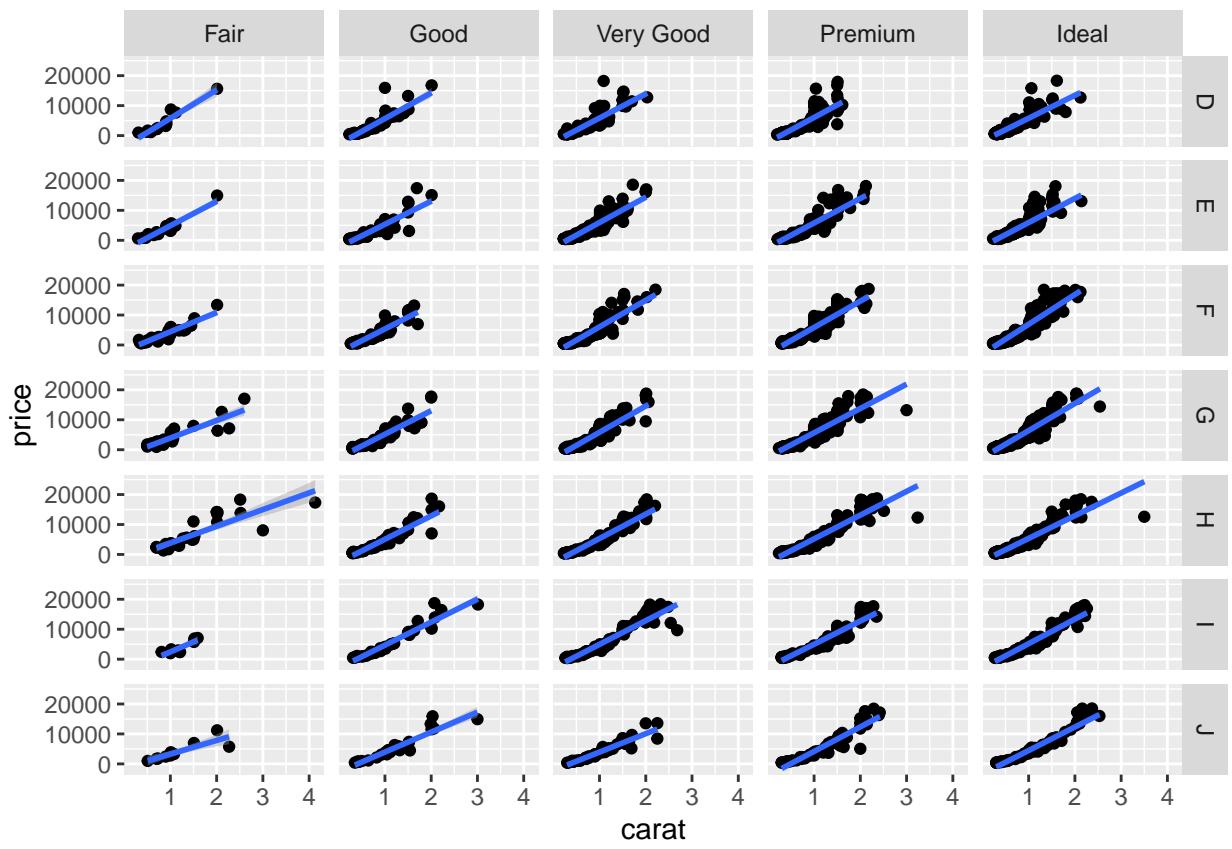


```
ggplot(diamonds2) + aes(x=carat,y=price) +  
  geom_smooth(method="loess") + facet_wrap(~cut,nrow=1)
```

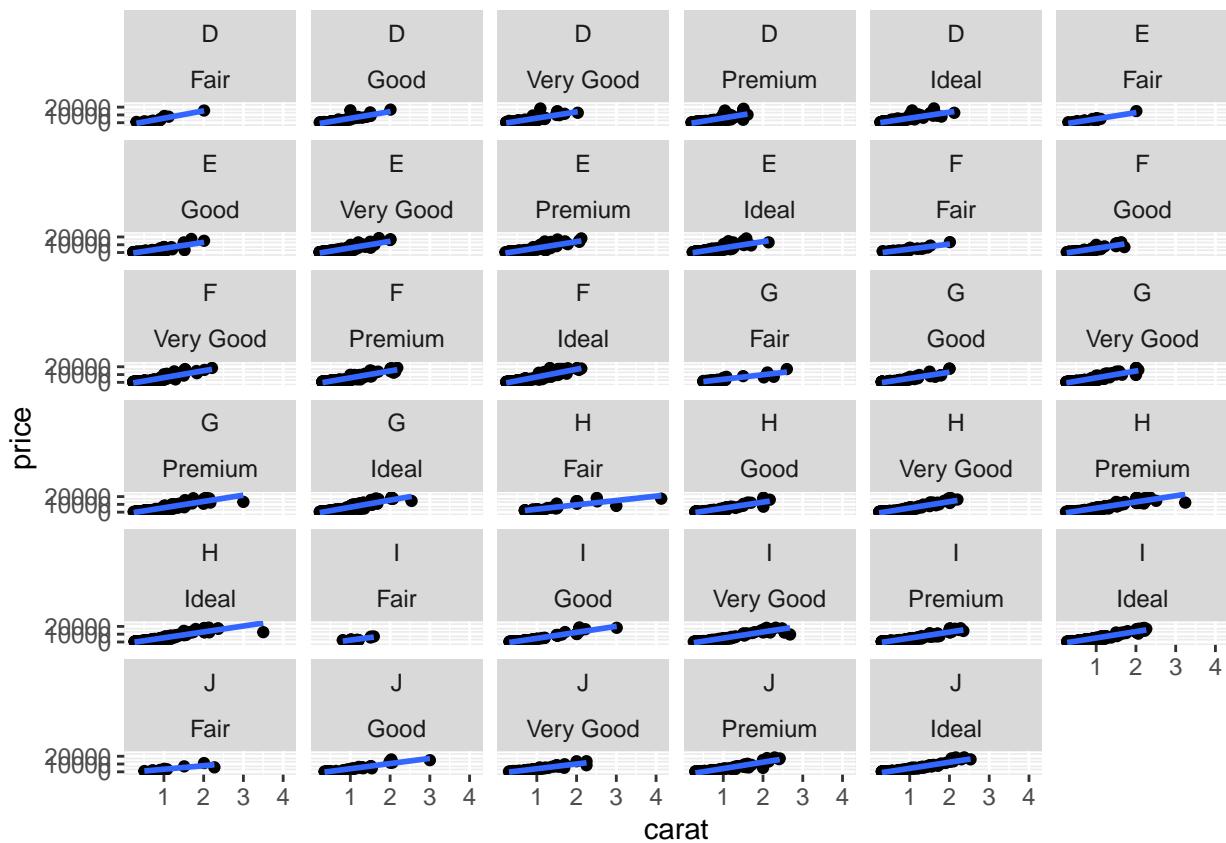


`facet_grid` and `facet_wrap` do the same job but split the screen in different ways :

```
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)
```



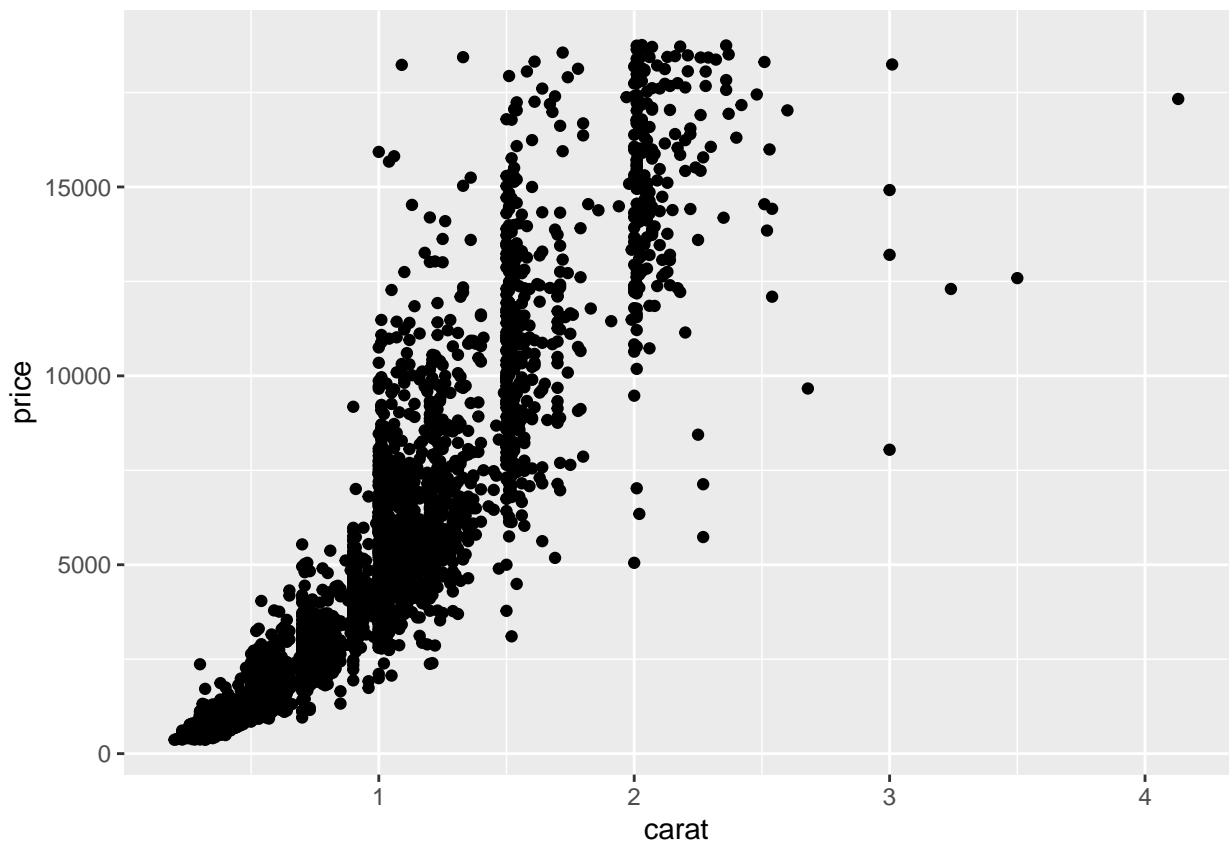
1.3 Complements

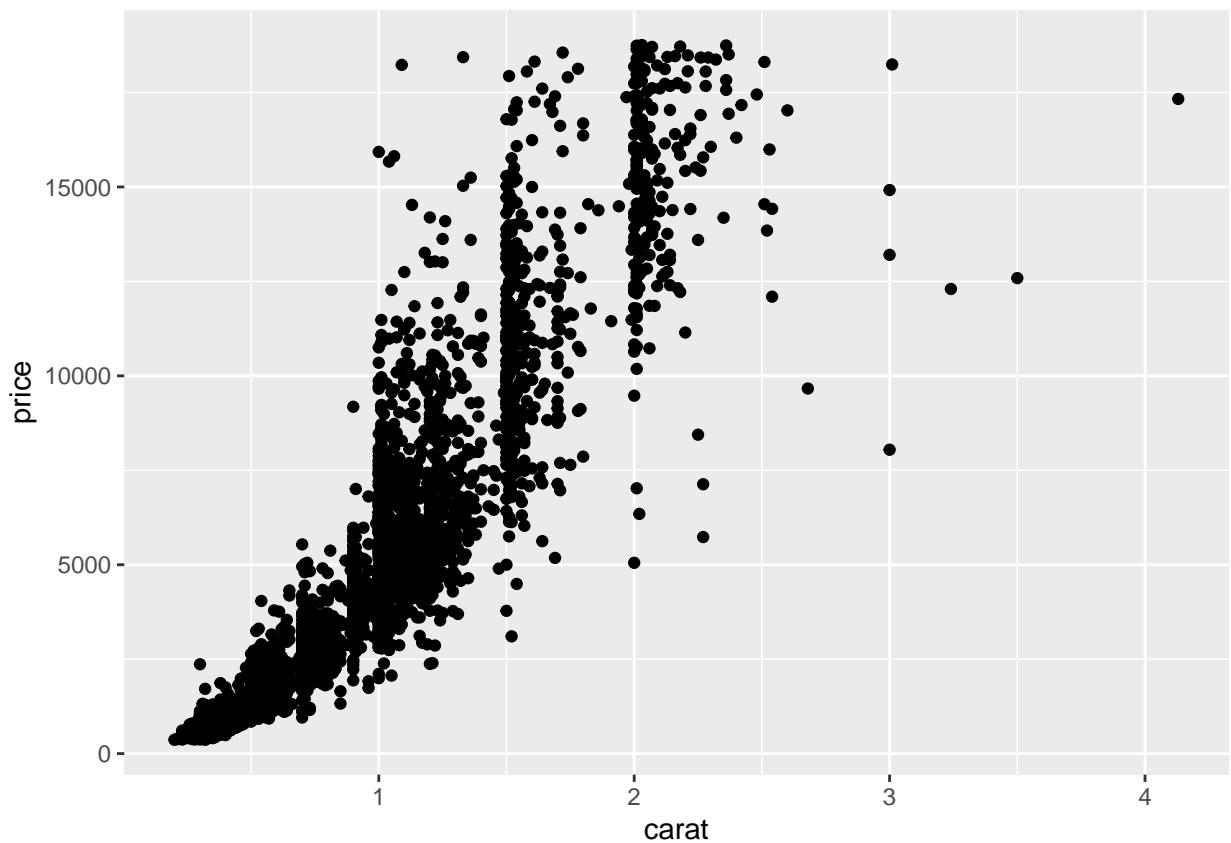
Syntax for **ggplot** is defined according to the following process :

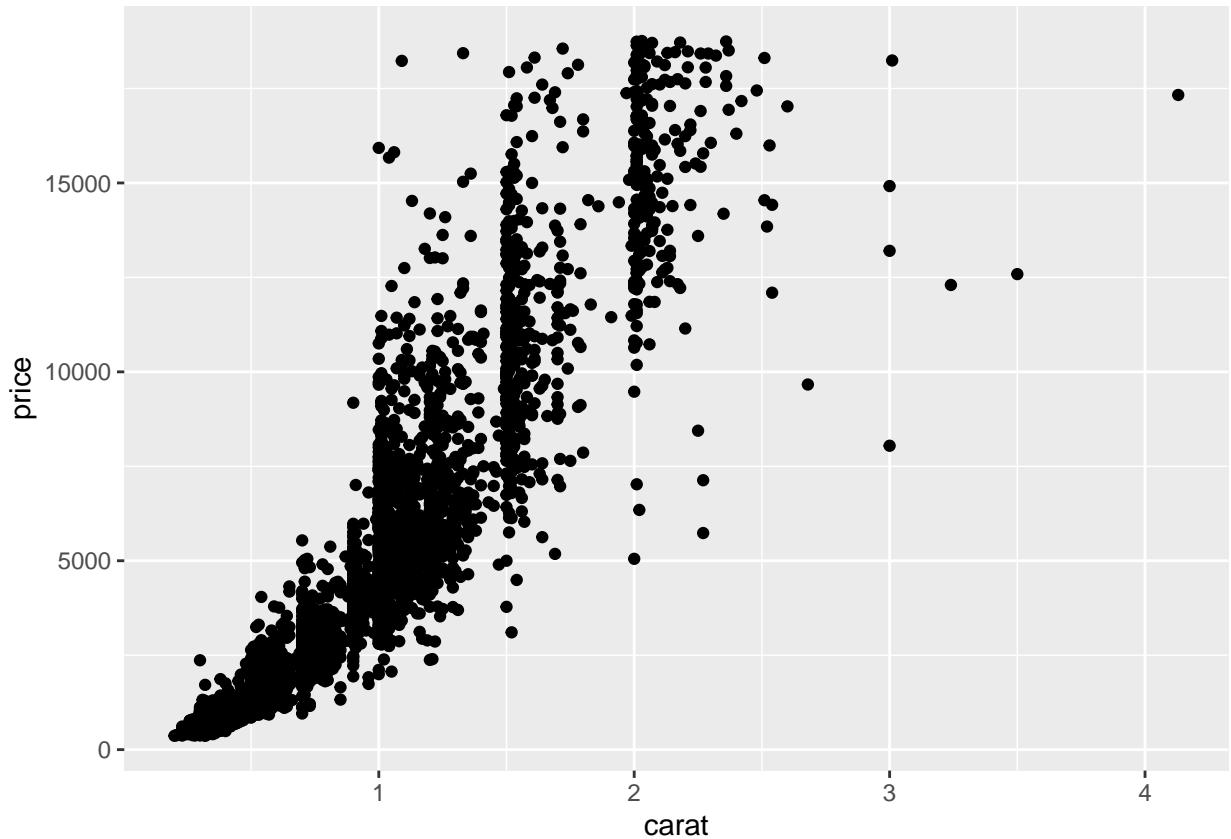
```
ggplot() + aes() + geom_() + scale_()
```

It is really flexible : for instance **aes** could also be specified in **ggplot** or in **geom_**

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



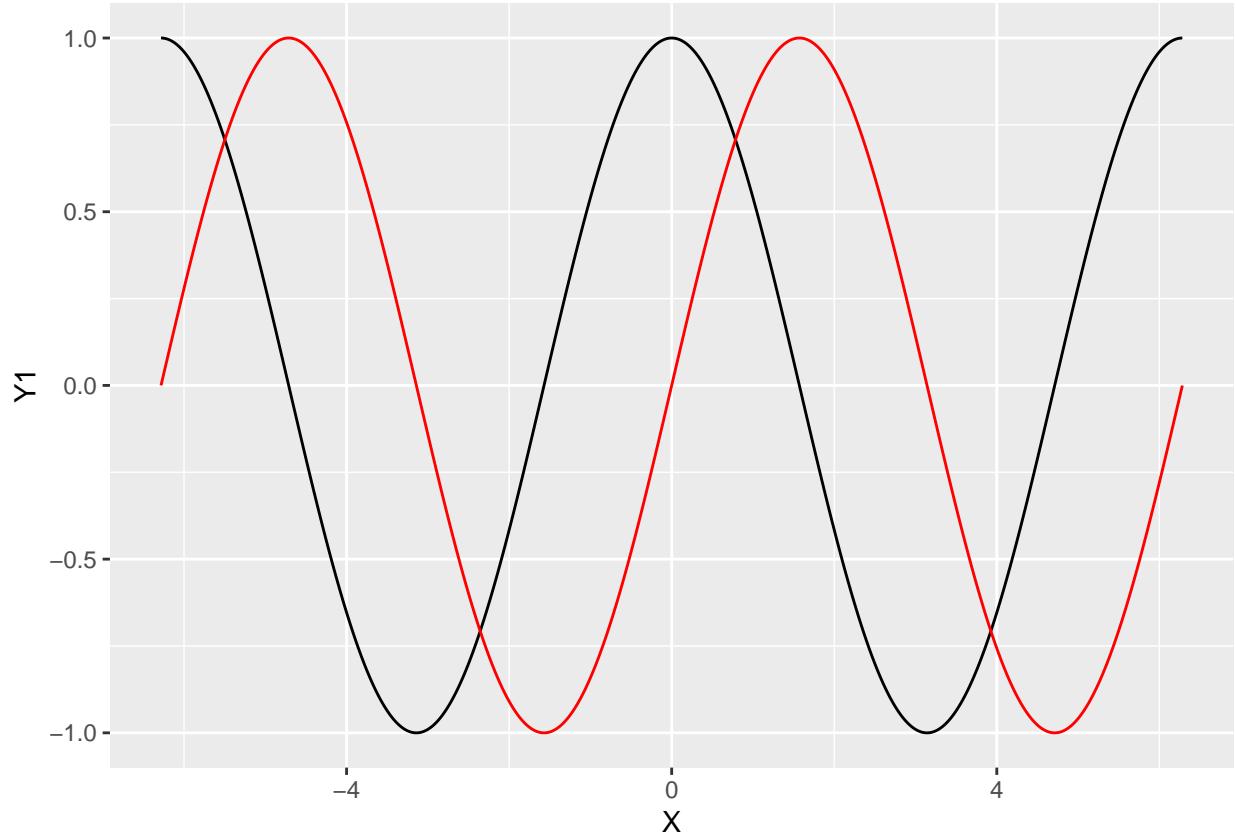




It could be very useful when we want to use different **aes** in the **geom_**

We can also build a chart from many dataset :

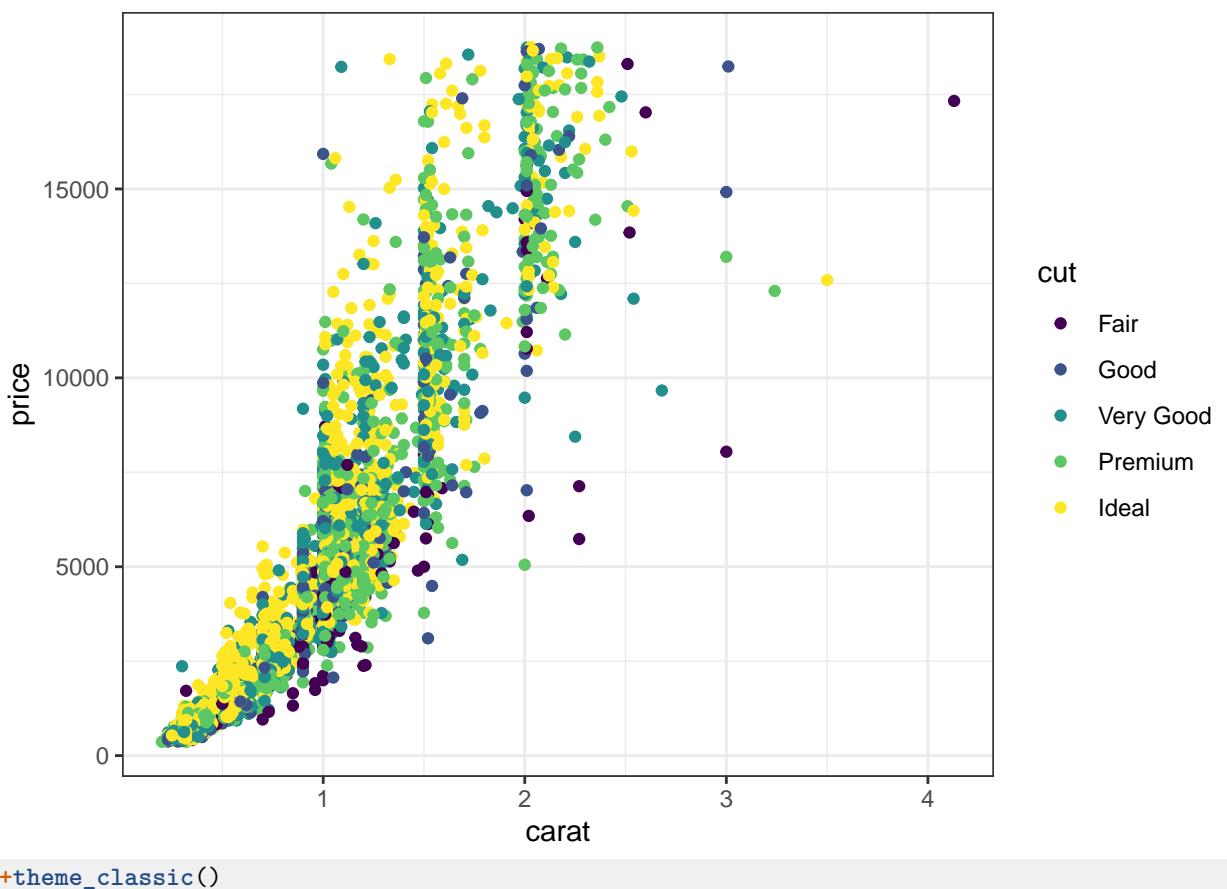
```
X <- seq(-2*pi,2*pi,by=0.001)
Y1 <- cos(X)
Y2 <- sin(X)
donnees1 <- data.frame(X,Y1)
donnees2 <- data.frame(X,Y2)
ggplot(donnees1)+geom_line(aes(x=X,y=Y1))+
  geom_line(data=donnees2,aes(x=X,y=Y2),color="red")
```

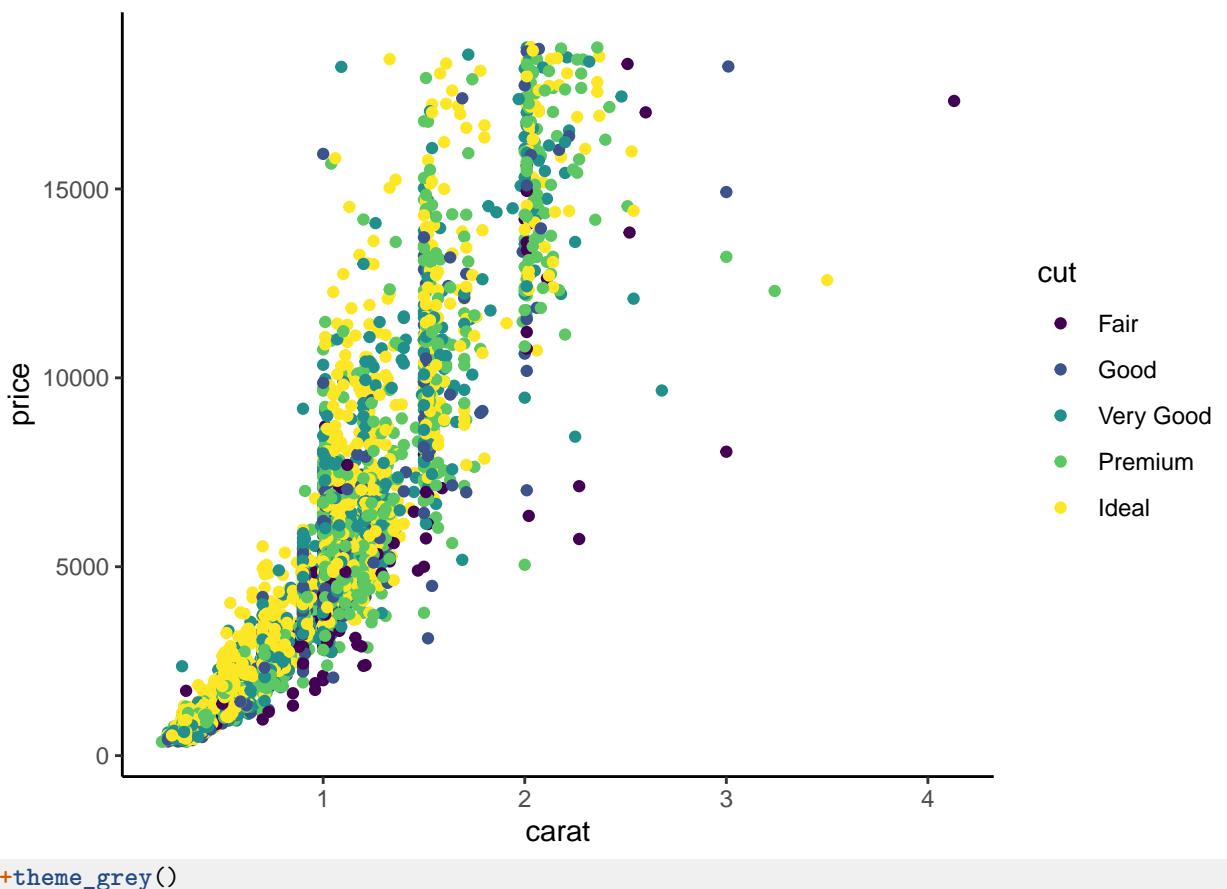


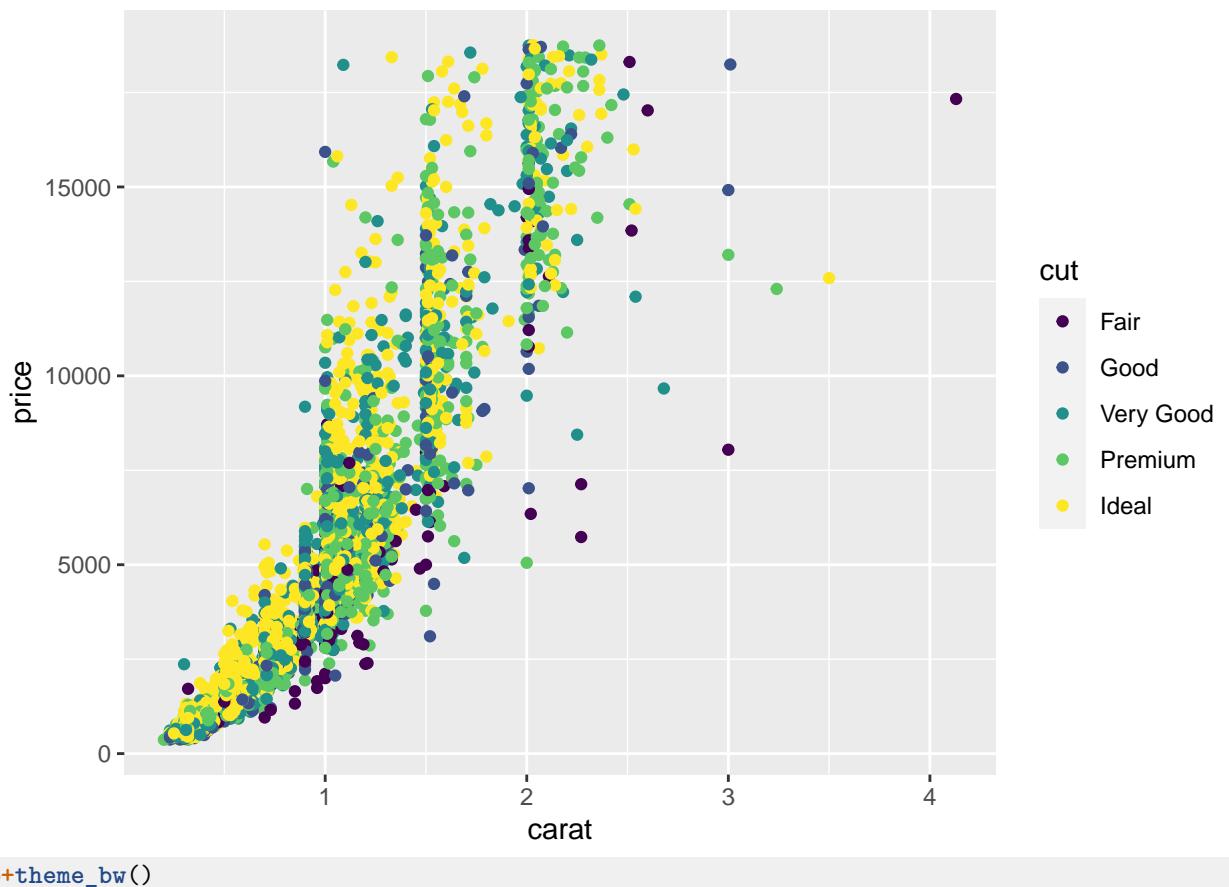
Many other functions are proposed by **ggplot** :

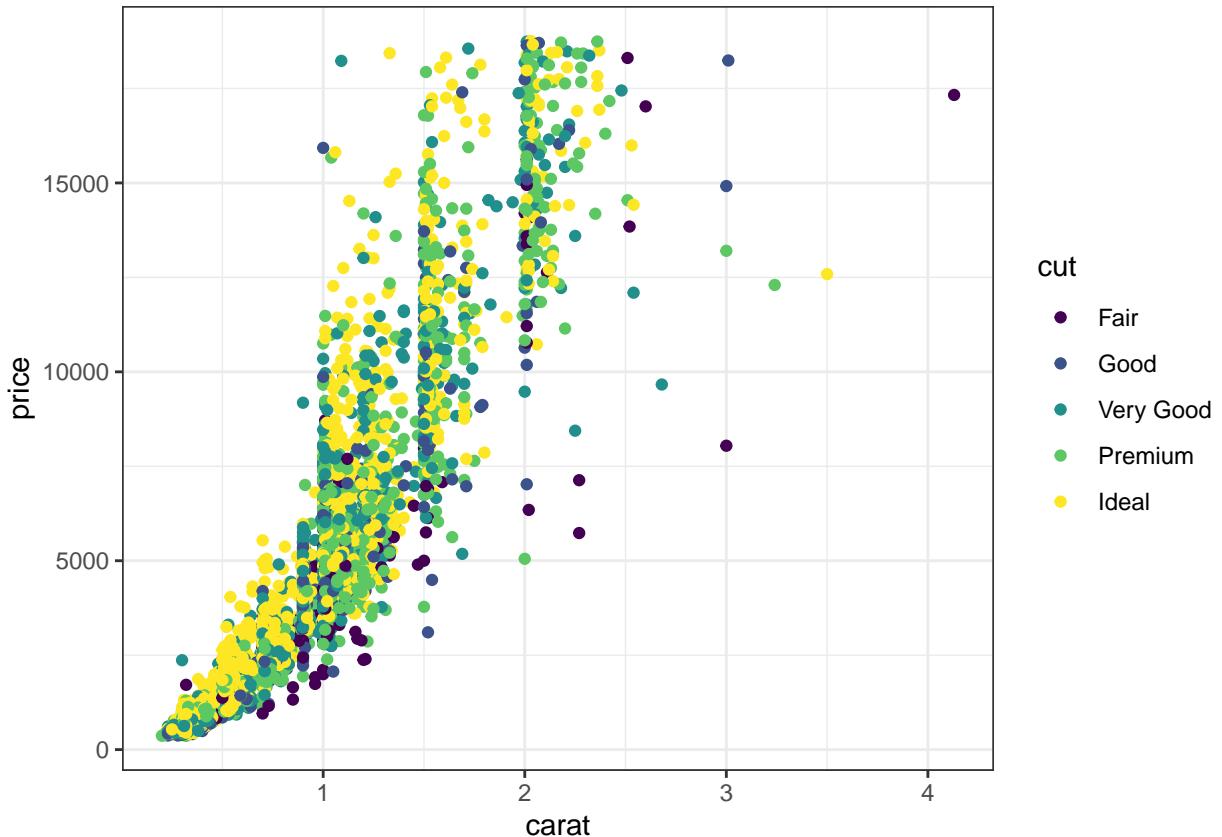
- **ggtitle** to add a title.
- **ggsave** to save a graph.
- **theme_** to change the theme of the graph.

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









Many other themes are available in the `ggtheme` package. The function `set_theme` can be used to change the default theme in a **markdown** document.

1.4 Some exercises

Exercise 1.9 (Cosine and sine functions...).

1. Draw the sine and cosine functions on the same graph. You first have to use two datasets (one for the sine function, the other for the cosine function).
2. Do the same with one dataset and 2 `geom_line`.
3. Do the same with one dataset and one `geom_line`. Hint : use the `pivot_longer` function from the `tidyverse` package.
4. Draw the two functions on two different graphs (use `facet_wrap`).
5. Do the same with the function `grid.arrange` from the package `gridExtra`.

Exercise 1.10 (Classical representations for a dataset).

We consider the `mtcars` dataset :

```
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. Draw the histogram of `mpg` (use many number of bins).
2. Always for the histogram, represent the density on the y -axis.
3. Draw the barplot of `cyl`.
4. Draw the scatter plot `disp vs mpg` with one color for each value of `cyl`.
5. Add the linear smoothers, one linear smoother for each value of `cyl`.

Exercise 1.11 (Residuals for a simple regression model).

1. Simulate a sample $(x_i, y_i), i = 1, \dots, 100$ according to the linear model

$$y_i = 3 + x_i + \varepsilon_i$$

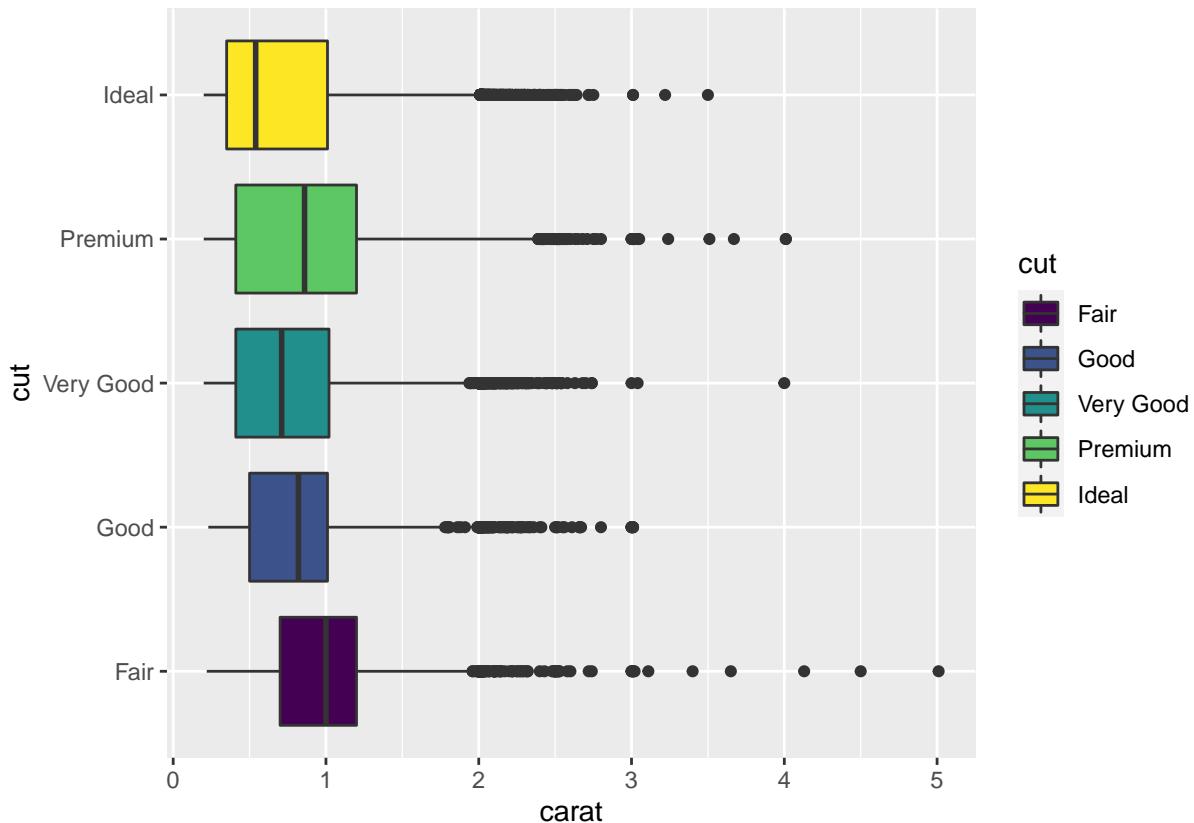
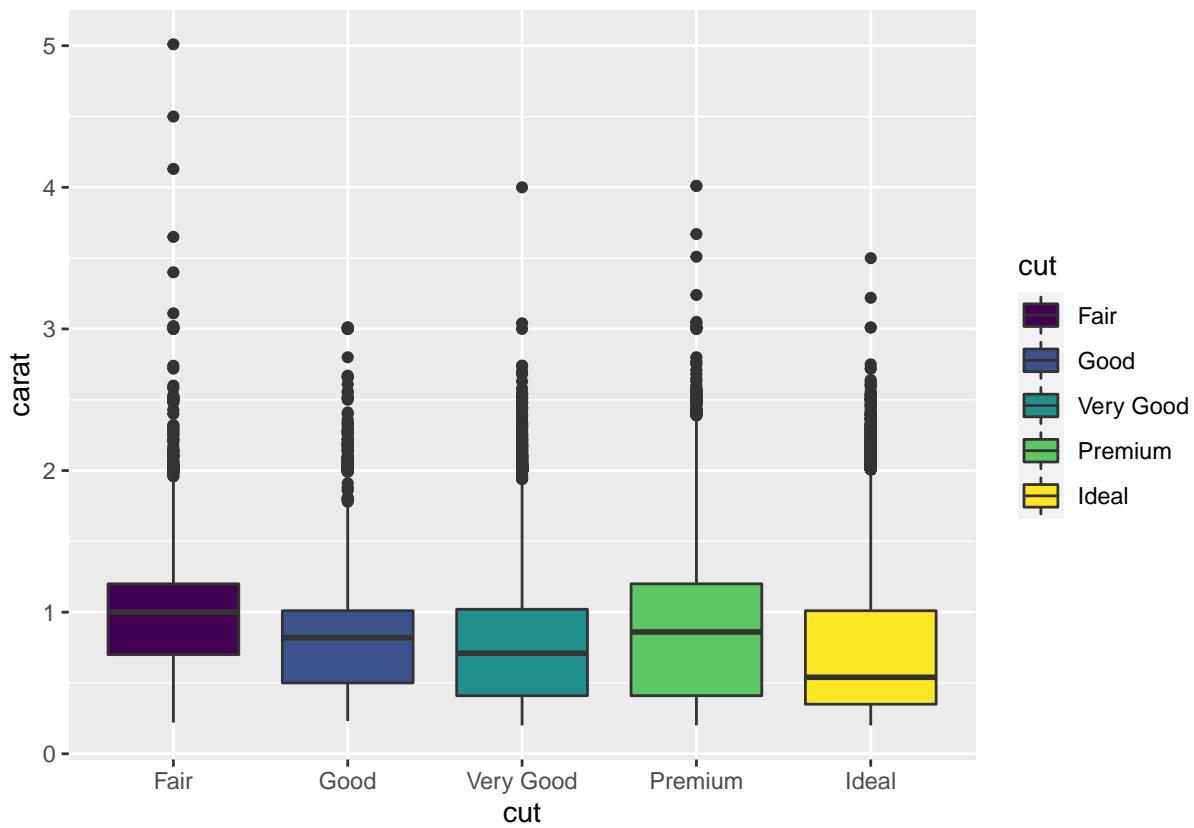
where the x_i 's are i.i.d. with uniform distribution over $[0, 1]$ and ε_i 's are gaussian $N(0, 0.2^2)$ (use `runif` and `rnorm`)

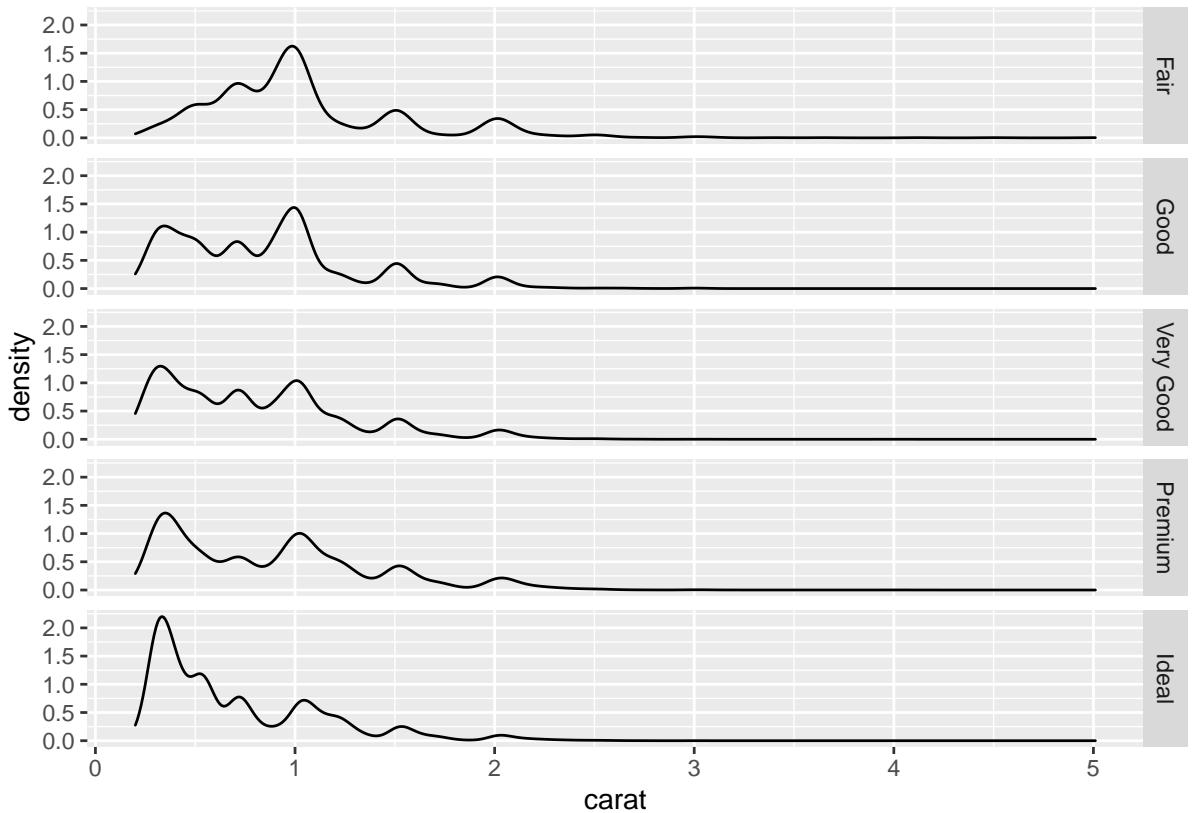
2. Draw the scatter plot `y vs x` and add the linear smoother.
3. Draw the residuals : add a vertical line from each point to the linear smoother (use `geom_segment`).

Exercise 1.12 (Challenge).

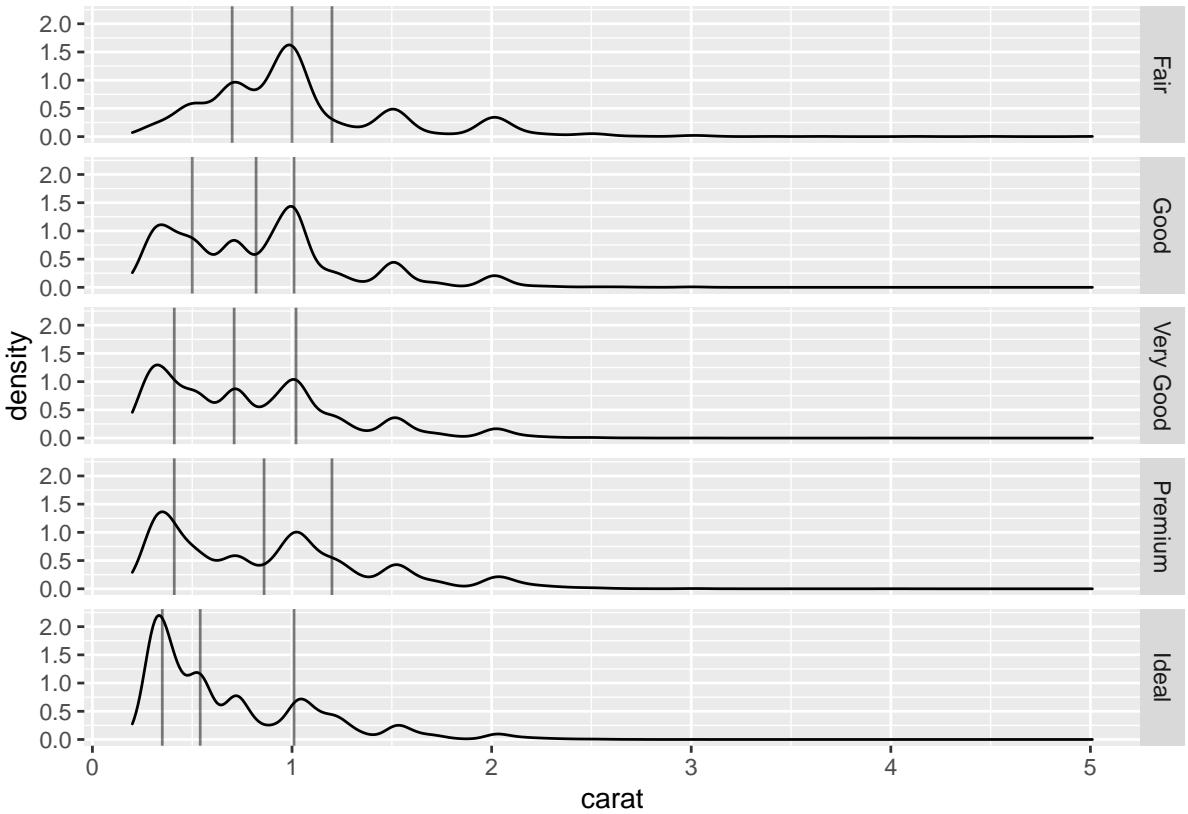
We consider the `diamonds` dataset.

1. Draw the following graphs (use `coord_flip` for the second one).

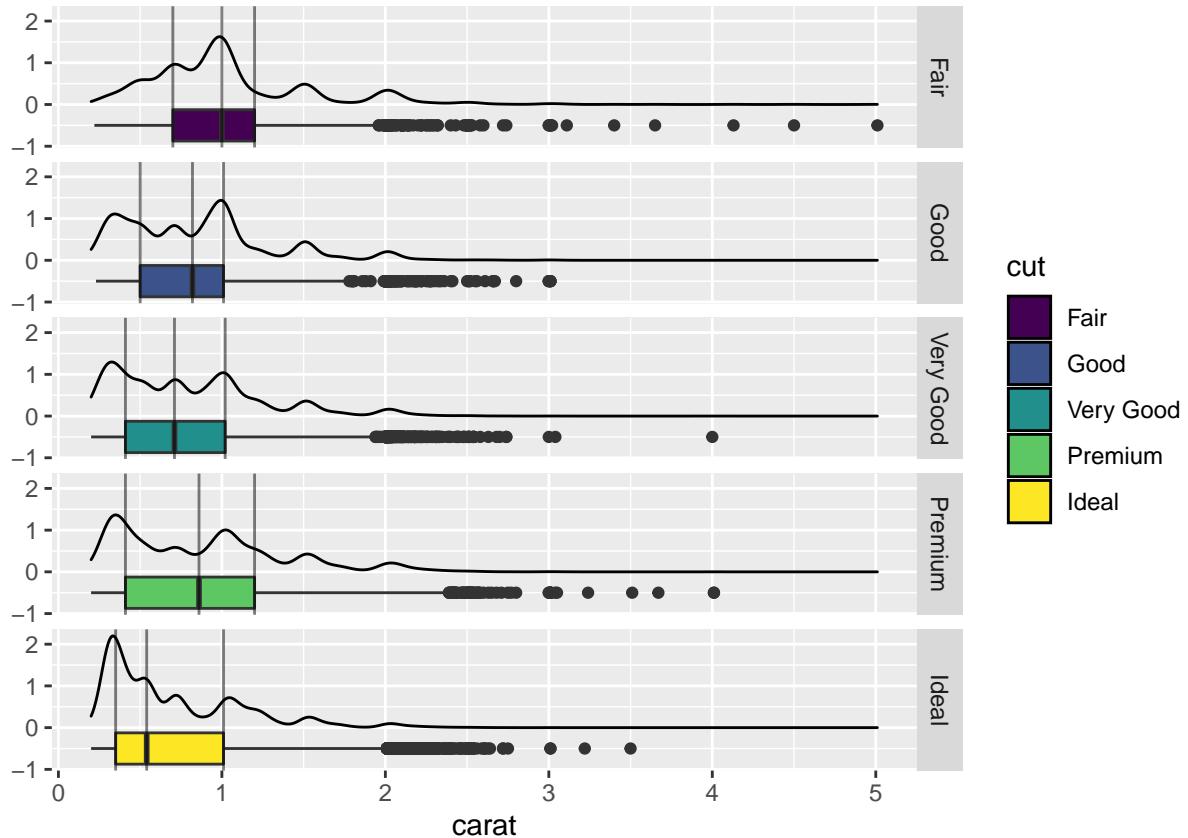




2. Add on the third graph a vertical line for the quartiles of the variable **carat** (for each value of **cut**).



3. Add a horizontal boxplot under each graph (use the `ggstance` package) to obtain the following chart :



2 Mapping with R

Many datasets contains informations about location. It is natural to use map to visualize these data. We can differentiate two kind of maps :

- **static map** : to be exported in **pdf** or **png** format, usefull to publish report ;
- **dynamic** or **interactive** map : to be exported in **html** format for instance. We can visualize these maps in a web browser and add informations when we click on the map.

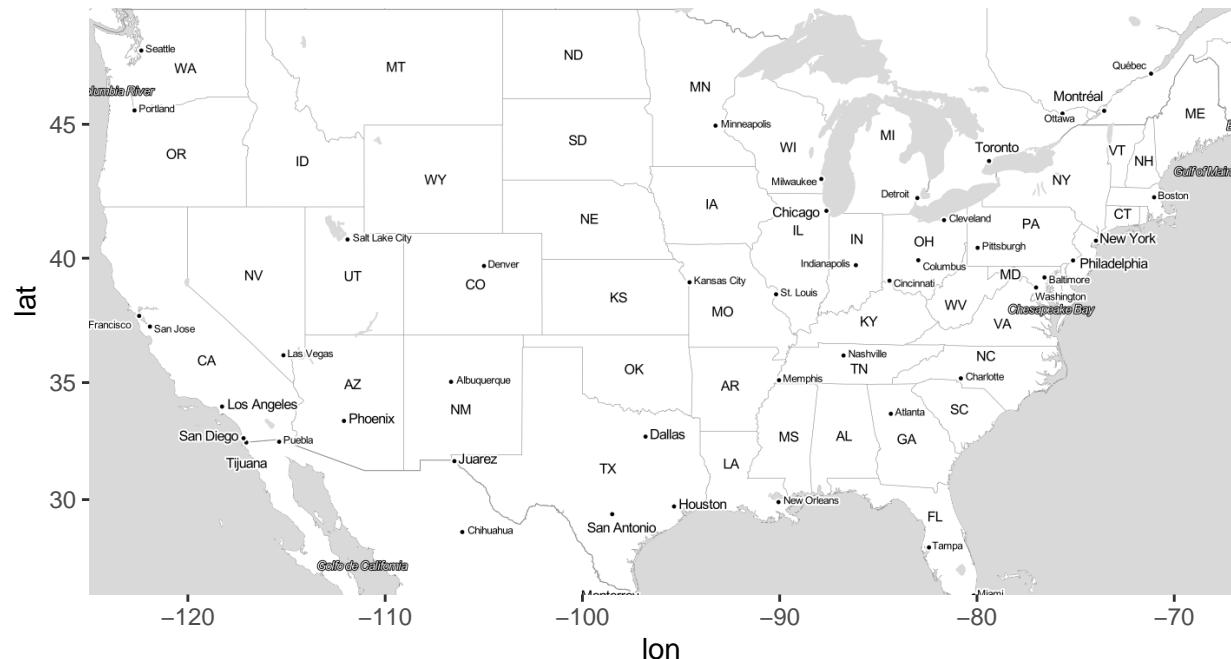
Many **R** packages exist for mapping. In this chapter, we present **ggmap** and **sf** for static map and **leaflet** for dynamic map.

2.1 ggmap package

We show in this part how to get background map and add informations from a dataset with **ggmap** package. For more details, we refer to [this article](#)

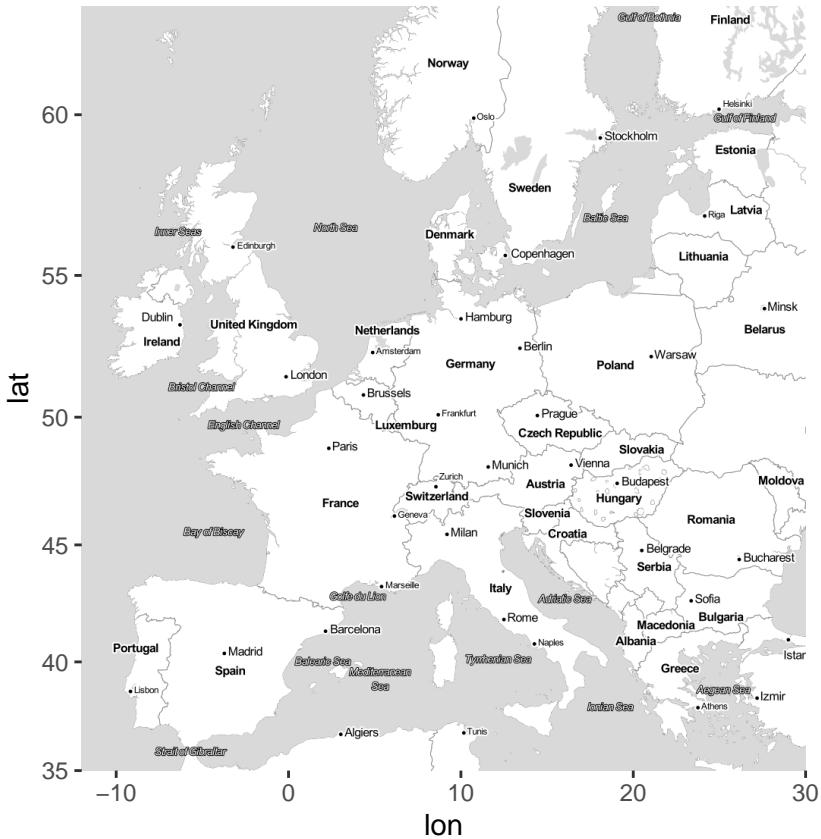
It is quite easy to download background map, for instance

```
library(ggmap)
us <- c(left = -125, bottom = 25.75, right = -67, top = 49)
map <- get_stamenmap(us, zoom = 5, maptype = "toner-lite")
ggmap(map)
```



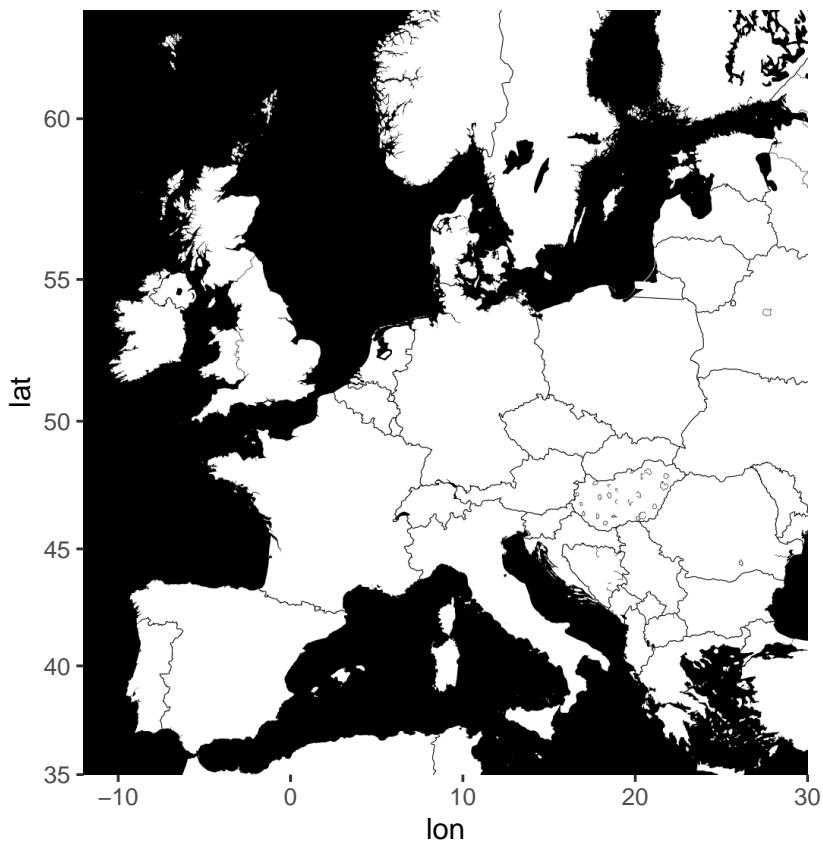
For Europe we can use

```
europe <- c(left = -12, bottom = 35, right = 30, top = 63)
get_stamenmap(europe, zoom = 5, "toner-lite") %>% ggmap()
```



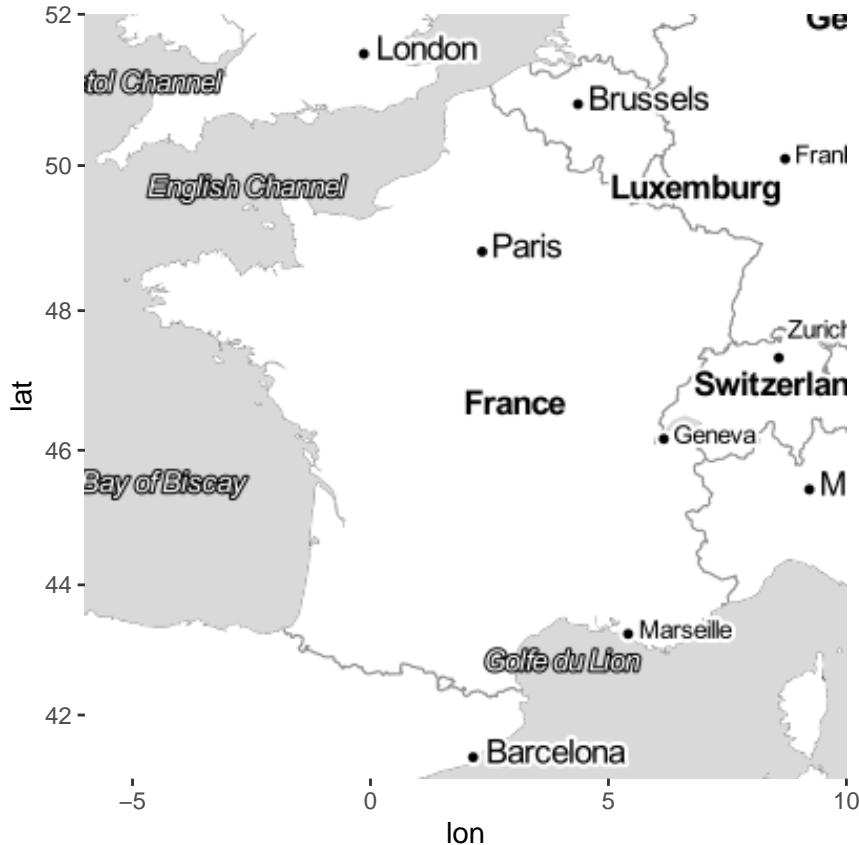
We can change the type of the background map

```
get_stamenmap(europe, zoom = 5,"toner-background") %>% ggmap()
```



Here is an example for France

```
fr <- c(left = -6, bottom = 41, right = 10, top = 52)
get_stamenmap(fr, zoom = 5, "toner-lite") %>% ggmap()
```



geocode function from ggmap made it possible to get latitudes and longitudes from a location. An **API** is now needed to use it (we could have to pay). We propose to use the following function instead :

```
if (!require(jsonlite)) install.packages("jsonlite")
mygeocode <- function(adresses){
# adresses est un vecteur contenant toutes les adresses sous forme de chaine de caracteres
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)
}
```

We can get latitudes and longitudes with

```
mygeocode("the white house")
          lon      lat
```

```

the white house -77.03655 38.8977
mygeocode("Paris")
  lon      lat
Paris 2.351462 48.8567
mygeocode("Rennes")
  lon      lat
Rennes -1.68002 48.11134

```

Exercise 2.1 (Population sizes in France).

1. Find latitudes and longitudes for Paris, Lyon and Marseille and represent these cities on a map (you can use a dot).
2. File **villes_fr.csv** contains population sizes for 30 cities in France. Visualize with a point these cities. We will use different point sizes in term of the populations sizes in 2014.

2.2 Maps with contours, shapefile format

ggmpap makes it easy to download background map and to add informations with **ggplot** functions. However, it is more difficult to take into account of contours (boundaries for countries or department...). We propose here an introduction to the **sf** package. It allows to create “advanced” maps, by managing boundaries through specific objects. Moreover we can manage many coordinate systmes with this package. Recall that earth is not flat... But we usually visualize a map in 2D. We thus have to make projections in order to visualize locations defined by a coordinate (latitude and longitude for instance). These projections are generally managed by the packages, it is the case of **sf**. We can find documentation on this package at :

- <https://statnmap.com/fr/2018-07-14-initiation-a-la-cartographie-avec-sf-et-compagnie/>
- in the **vignettes** of the documentation of the package : <https://cran.r-project.org/web/packages/sf/index.html>

Package **sf** allows to define a new class of **R** objects specific for mapping. Look at the object **nc**

```

library(sf)
nc <- st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
class(nc)
[1] "sf"        "data.frame"
nc
Simple feature collection with 100 features and 14 fields
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
CRS:            4267
First 10 features:
#> #>   AREA PERIMETER CNTY_ CNTY_ID      NAME  FIPS FIPSNO
#> #> 1 0.114     1.442  1825    1825      Ashe 37009 37009
#> #> 2 0.061     1.231  1827    1827  Alleghany 37005 37005
#> #> 3 0.143     1.630  1828    1828      Surry 37171 37171
#> #> 4 0.070     2.968  1831    1831 Currituck 37053 37053
#> #> 5 0.153     2.206  1832    1832 Northampton 37131 37131
#> #> 6 0.097     1.670  1833    1833 Hertford 37091 37091
#> #> 7 0.062     1.547  1834    1834 Camden 37029 37029
#> #> 8 0.091     1.284  1835    1835 Gates 37073 37073
#> #> 9 0.118     1.421  1836    1836 Warren 37185 37185
#> #> 10 0.124    1.428  1837    1837 Stokes 37169 37169
#> #> CRESS_ID BIR74 SID74 NWBIR74 BIR79 SID79 NWBIR79
#> #> 1       5 1091     1     10 1364      0     19

```

```

2      3   487     0     10   542     3    12
3     86  3188     5    208  3616     6   260
4     27   508     1    123   830     2   145
5     66  1421     9   1066  1606     3  1197
6     46  1452     7    954  1838     5  1237
7     15   286     0    115   350     2   139
8     37   420     0    254   594     2   371
9     93   968     4    748  1190     2   844
10    85  1612     1    160  2038     5   176
              geometry
1 MULTIPOINT ((-81.47276 3...
2 MULTIPOLYGON (((-81.23989 3...
3 MULTIPOLYGON (((-80.45634 3...
4 MULTIPOLYGON (((-76.00897 3...
5 MULTIPOLYGON (((-77.21767 3...
6 MULTIPOLYGON (((-76.74506 3...
7 MULTIPOLYGON (((-76.00897 3...
8 MULTIPOLYGON (((-76.56251 3...
9 MULTIPOLYGON (((-78.30876 3...
10 MULTIPOLYGON (((-80.02567 3...

```

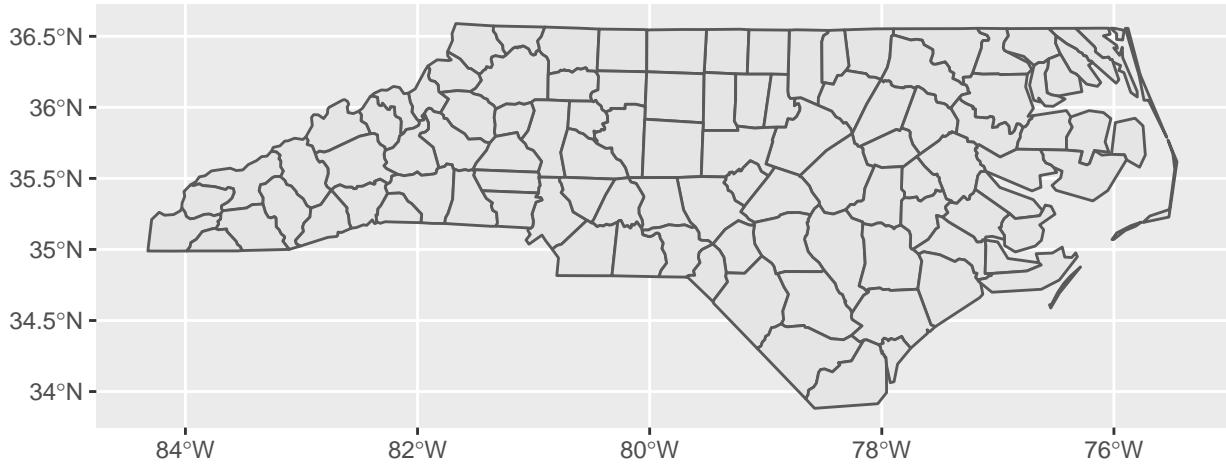
This dataset contains informations about sudden infant death syndrome in North Carolina counties. We observe 2 classes for `nc` : `sf` and `data.frame`. We can manage it as a classical `dataframe`. `sf` class allows to define a particular column (`geometry`) in which we can specify boundaries of the counties with `polygon`. This column makes it easy to plot these boundaries (with polygon) in a map. We can use the classicat `plot` function

```
plot(st_geometry(nc))
```



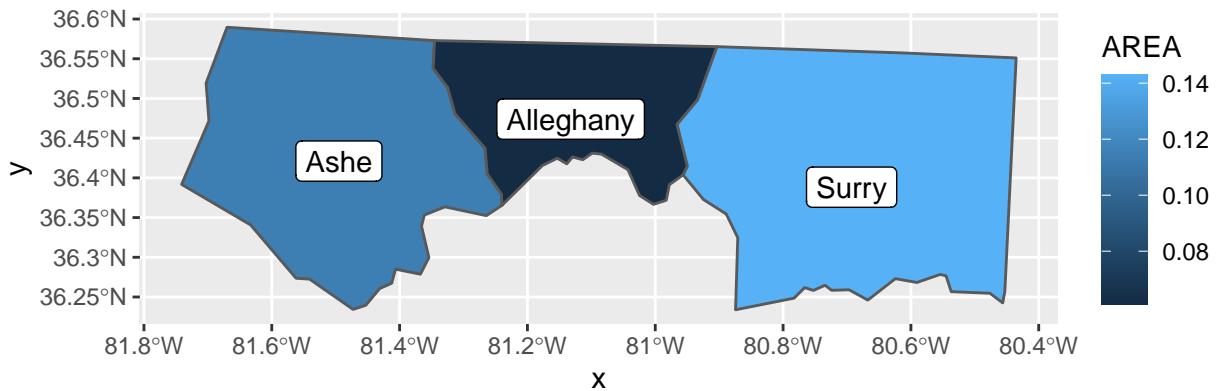
or the `geom_sf` verb if we prefer `ggplot`

```
ggplot(nc)+geom_sf()
```



It is now easy to color counties and to add their names

```
ggplot(nc[1:3,]) + geom_sf(aes(fill = AREA)) + geom_sf_label(aes(label = NAME))
```



`geometry` column of `nc` uses `MULTIPOLYGON`. This allows to represent boundaries. If we want to represent one country by a point, we have to change the format of this column. We can do it as follows :

1. We first compute latitude and longitude for each country :

```
coord.ville.nc <- mygeocode(paste(as.character(nc$NAME), "NC"))
coord.ville.nc <- as.data.frame(coord.ville.nc)
names(coord.ville.nc) <- c("lon", "lat")
```

2. We give to these coordinates the format `MULTIPOINT` (`st_multipoint` function)

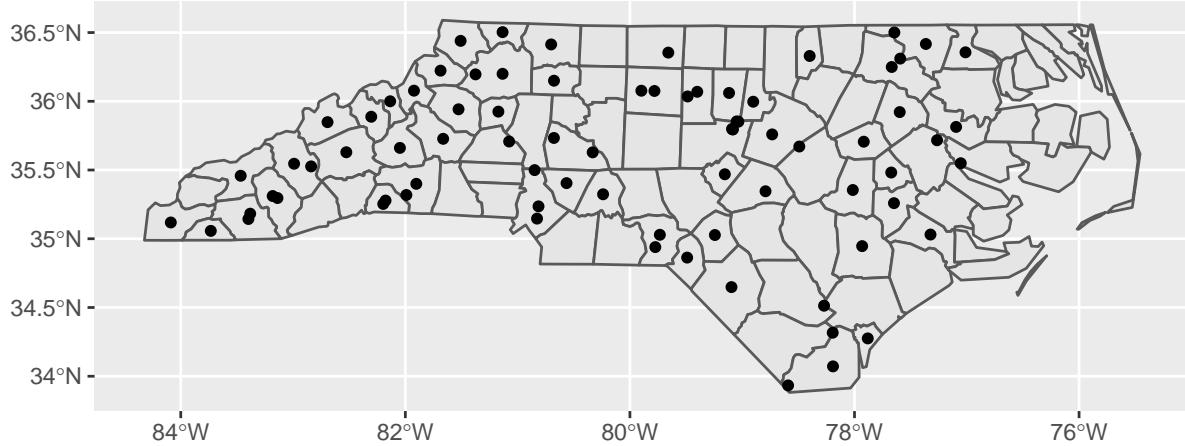
```
coord.ville1.nc <- coord.ville.nc %>%
  filter(lon<=-77 & lon>=-85 & lat>=33 & lat<=37) %>%
  as.matrix() %>% st_multipoint() %>% st_geometry()
coord.ville1.nc <- coord.ville.nc %>%
  filter(lon<=-77 & lon>=-85 & lat>=33 & lat<=37) %>%
  as.matrix() %>% st_multipoint() %>% st_geometry() %>% st_cast(to="POINT")
coord.ville1.nc
Geometry set for 79 features
geometry type:  POINT
dimension:      XY
bbox:           xmin: -84.08862 ymin: 33.93323 xmax: -77.01151 ymax: 36.503
CRS:            NA
First 5 geometries:
```

3. We indicate that these coordinates correspond to longitudes and latitudes (there exists other coordinate systems) :

```
st_crs(coord.ville1.nc) <- 4326
```

4. We can now visualize both the boundaries and the counties :

```
ggplot(nc)+geom_sf()+geom_sf(data=coord.ville1.nc)
```

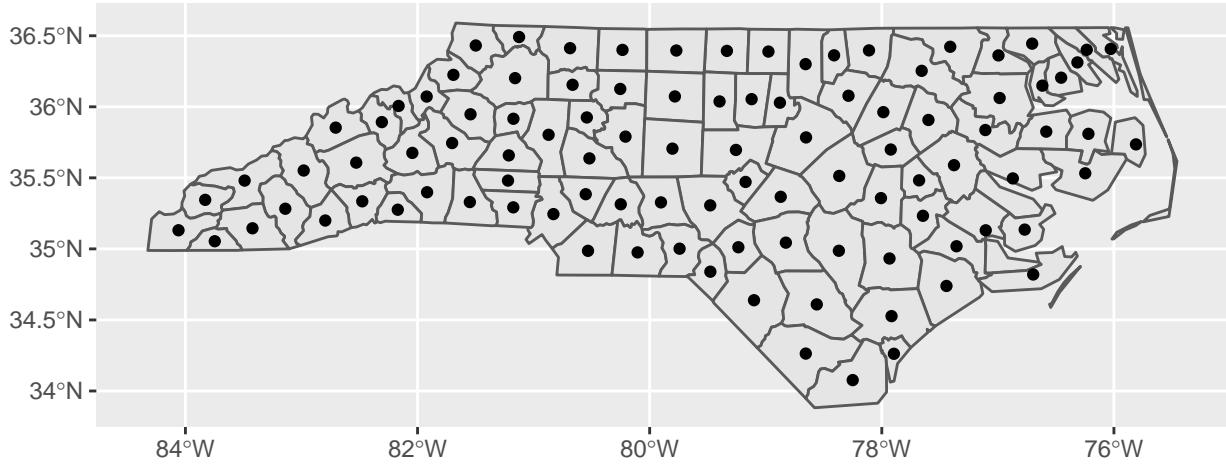


There are many useful functions in **sf** to deal with mapping data, for instance

- **st_distance** to compute (real) distances between coordinates ;
- **st_centroid** to obtain the center of a polygon (or region) ;
- ...

For example we can visualize the centers of the areas identified by the polygons in **nc** with

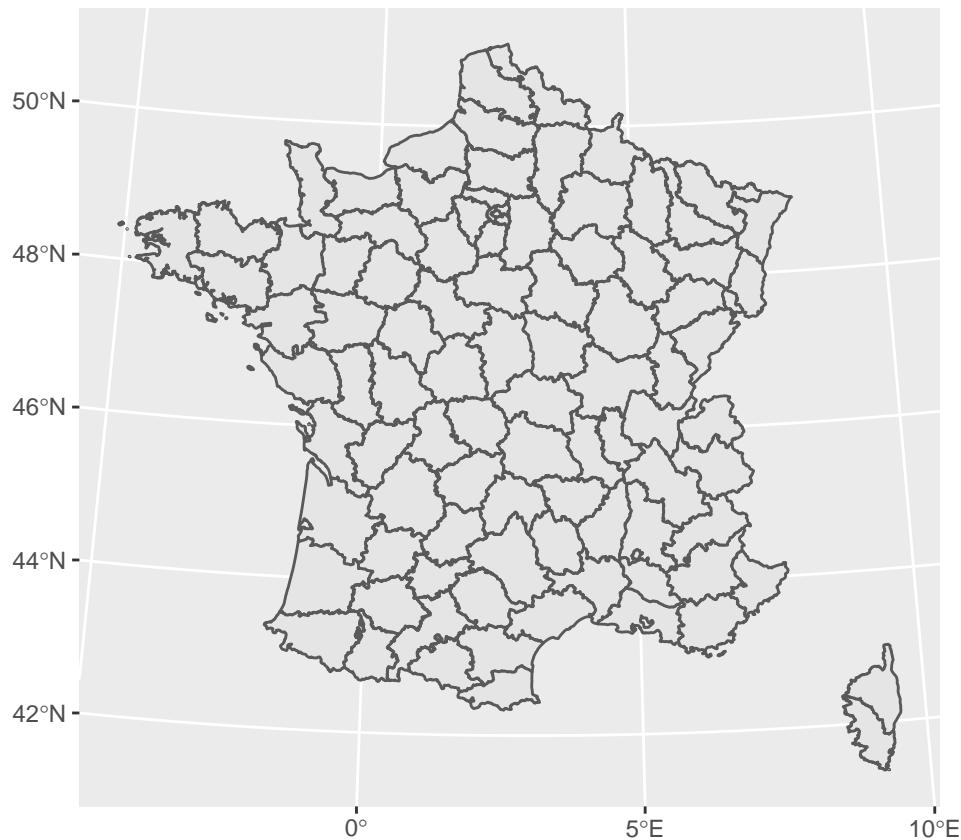
```
nc2 <- nc %>% mutate(centre=st_centroid(nc)$geometry)  
ggplot(nc2)+geom_sf()+geom_sf(aes(geometry=centre))
```



Exercise 2.2 (A first map with **sf**).

We consider the GEOFLAR map proposed by “l’Institut Géographique National” to obtain a shapefile background map of the french departments. This map is available at <http://professionnels.ign.fr/>, we can obtain it in the file **dpt.zip** (you have to unzip this file). We can import this map into a R object with

```
dpt <- read_sf("data/dpt")
ggplot(dpt) + geom_sf()
```



Redo the map of exercise 2.1 with this background map.

Exercise 2.3 (Unemployment rates).

We want to visualize differences between unemployment rates in 2006 and 2011. The data are in the file **tauxchomage.csv**. We are interested in variables TCHOMB1T06 and TCHOMB1T11.

1. Read the dataset.
2. Merge this dataset with the one of the department. You can use **inner_join**.
3. Make a comparison between the unemployment rates in 2006 and 2011 (build one map for 2006 and another one for 2011).

2.2.1 Challenge 1 : temperature map with sf

We want to visualize temperatures in france for a given date. Data are available on the website of [public data of meteofrance](#). In particular, we can obtain temperatures measured in some stations for many dates as well as the locations of these stations.

1. Import the 2 required datasets (you can read those directly on the website). Temperatures are given in Kelvin. Convert them in Celsius (you just have to subtract 273.15).
2. Keep only stations from metropolitan France (remove stations from overseas departments). **Hint** : You can keep only stations with longitude between -20 and 25. We call this dataset **station1**. Visualize stations on a French map with boundaries department.
3. Create a data frame (with **sf** format) which contains temperatures of the stations in one column and a column **geometry** with the coordinate of these stations. We can start with

```

station2 <- station1 %>% select(Longitude, Latitude) %>%
  as.matrix() %>% st_multipoint() %>% st_geometry()
st_crs(station2) <- 4326
station2 <- st_cast(station2, to = "POINT")

```

4. Visualize the stations in the french map. You can color the points (stations) according to the observed temperatures.
5. We obtain coordinates of the centroids of the departments with

```

centro <- st_centroid(dpt$geometry)
centro <- st_transform(centro, crs=4326)

```

We then deduce distances between these centroids and the stations (here **df** is the **sf** table computed in question 3).

```
DD <- st_distance(df, centro)
```

Predict the temperature in each department with a one nearest neighbour rule.

6. Color departments according to the predicted temperatures. We can use a gradient color from yellow (for low temperatures) to red (for high temperatures).

2.2.2 Finding background shapefile map

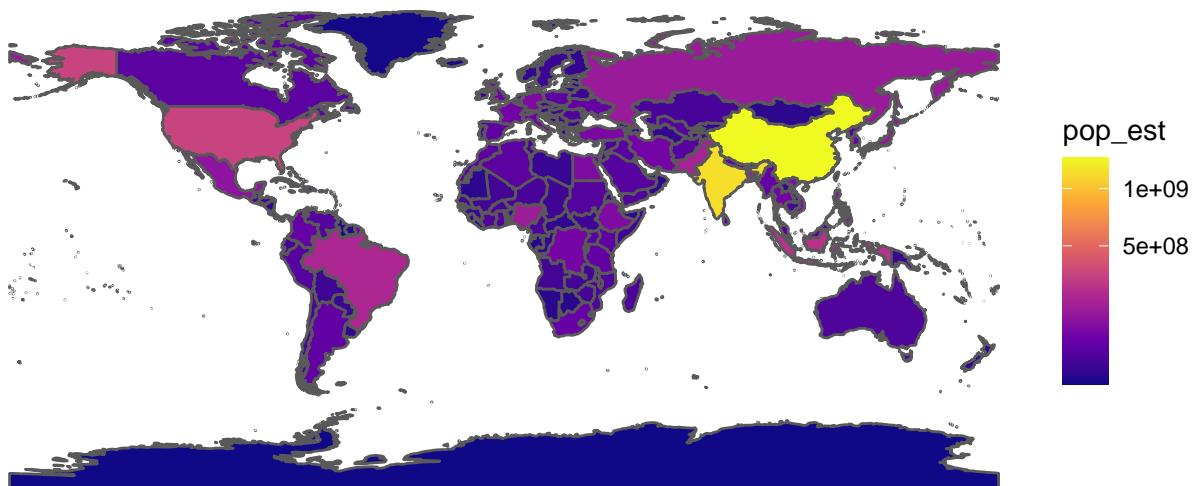
We usually need background shapefile map to build a map with **sf**. Many solution exists :

- Some **R** packages, for instance **rnatualearth** :

```

world <- rnatural-earth::ne_countries(scale = "medium", returnclass = "sf")
class(world)
[1] "sf"           "data.frame"
ggplot(data = world) +
  geom_sf(aes(fill = pop_est)) +
  scale_fill_viridis_c(option = "plasma", trans = "sqrt") + theme_void()

```



We can also visualize earth as a sphere :

```

ggplot(data = world) +
  geom_sf() +
  coord_sf(crs = "+proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +units=m +no"

```



See <https://www.r-spatial.org/r/2018/10/25/ggplot2-sf.html> for more details.

- The **web**, for instance on **data gouv** :

```
regions <- read_sf("data/regions-20180101-shp/")
```

Be careful, sometimes the object size can be very large :

```
format(object.size(regions),units="Mb")
[1] "15.4 Mb"
```

The construction of the map could be time consuming in this case, we have to reduce the size before building the map :

```
library(rmapshaper)
regions1 <- ms_simplify(regions)
format(object.size(regions1),units="Mb")
[1] "0.9 Mb"
ggplot(regions1)+geom_sf()+
  coord_sf(xlim = c(-5.5,10),ylim=c(41,51))+theme_void()
```



2.3 Dynamic maps with leaflet

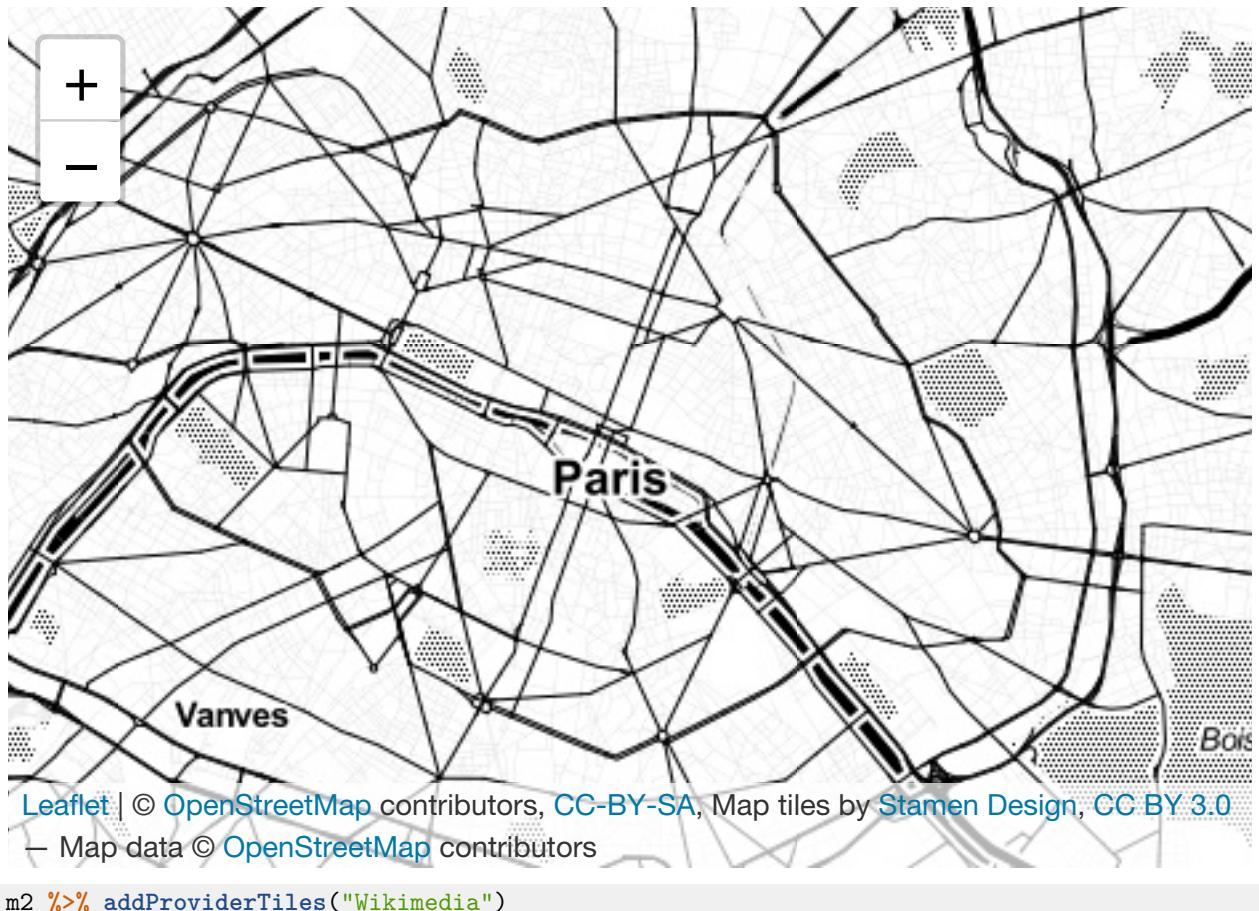
Leaflet is a R package which allows to produce interactive maps. You can find informations about this pakage at <https://rstudio.github.io/leaflet/>. The ideas are quite the same as for `ggmap` and `sf` : maps are defined from many layers. A leaflet background map can be created with

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



Many formats of background maps are available (some examples [here](#)). For instance

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





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

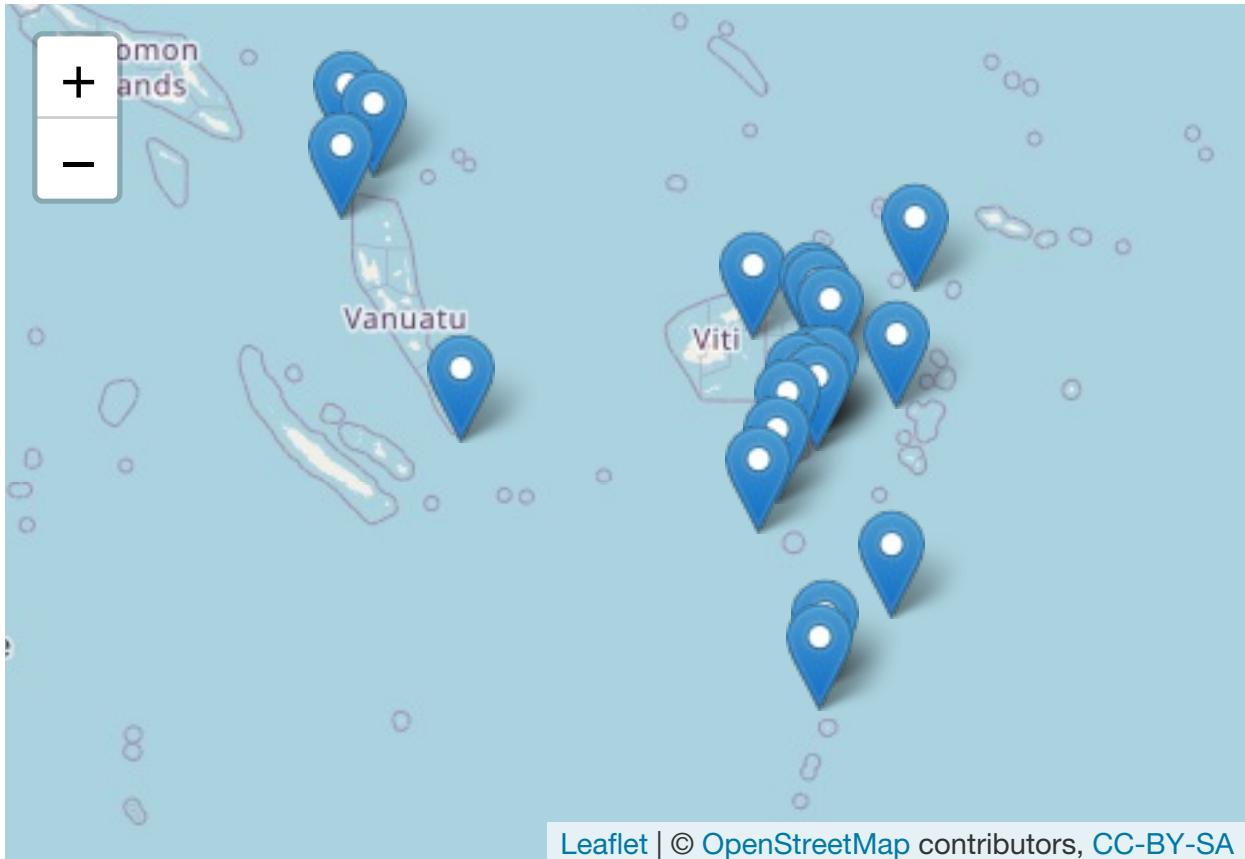


```
m2 %>%
  addProviderTiles("Stamen.Watercolor") %>%
  addProviderTiles("Stamen.TonerHybrid")
```



We can localize places with markers or circles

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

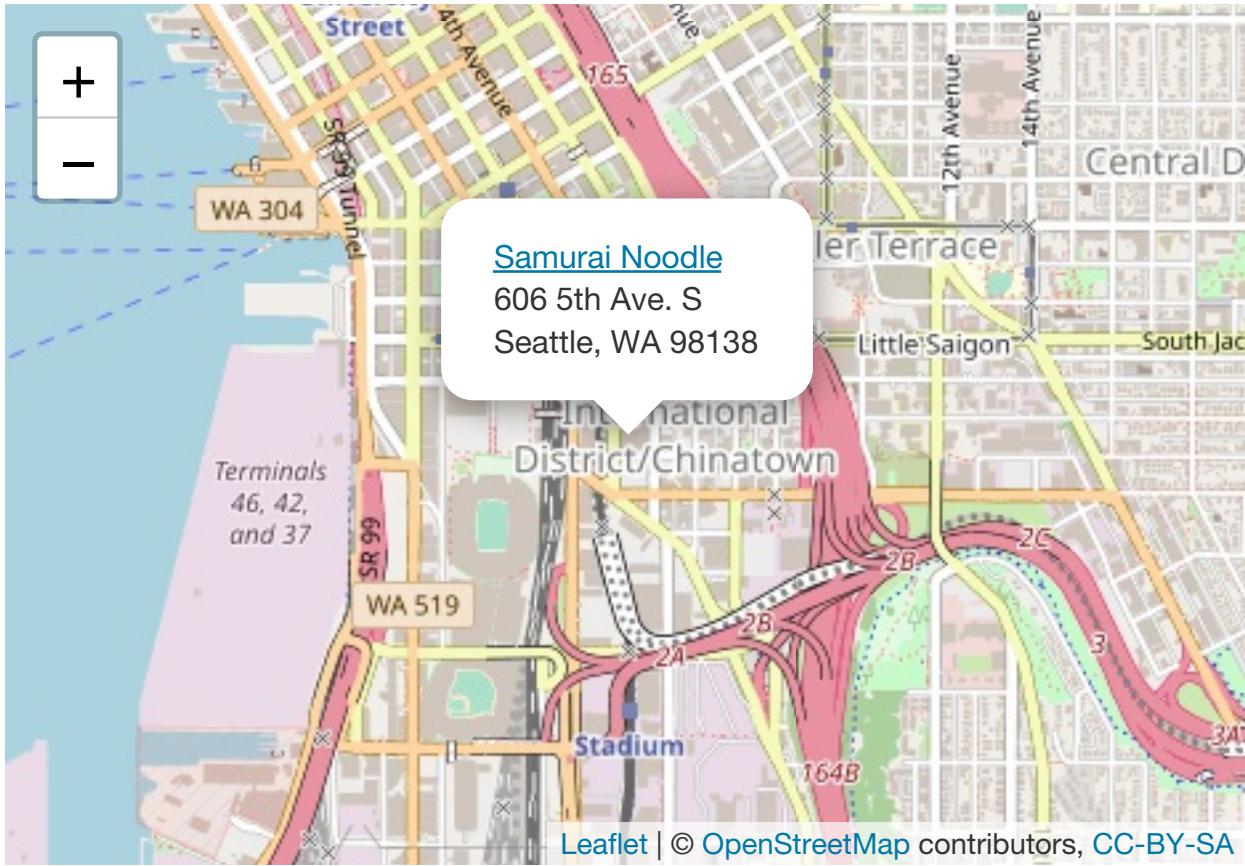


Observe that we have to use the tilda character ~ when we want to use the names of the dataframe to obtain the map.

The dynamic process can be used to add information when we click on a marker (with `popup` option). We can also add **popups** as follows

```
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)
  )
```



Exercise 2.4 (Popup with leaflet).

Add a popup on a leaflet map which localize Ensai. We also specify in the popup the website of the school.

2.3.1 Challenge 2 : city bike in Paris

Many cities all over the world provides informations on bike stations. These data are easily available et are updated in real time. Among other, we have informations about the size, the location, the number of available bikes for each stations. We can obtain these data on

- Decaux
- Open data Paris
- vlstats

1. Import informations onbike stations in Paris fr today : <https://opendata.paris.fr/explore/dataset/velib-disponibilite-en-temps-reel/information/>. You can use `read_delim` function with `delim=";"`.
2. Describe the dataset.
3. Create variables `latitude` and `longitude` from the column `geo`. You can use the verb `separate` from the `tidyverse` package.
4. Visualize the stations on a leaflet background map.
5. Add a popup which indicates the number of available bikes (electric+mecanic) when we click on the station (you can use the `popup` option in the function `addCircleMarkers`).
6. Add the name of the station in the popup.
7. Do the same with different colors to visualize the proportion of available bikes in the station. You can use the color palettes

```

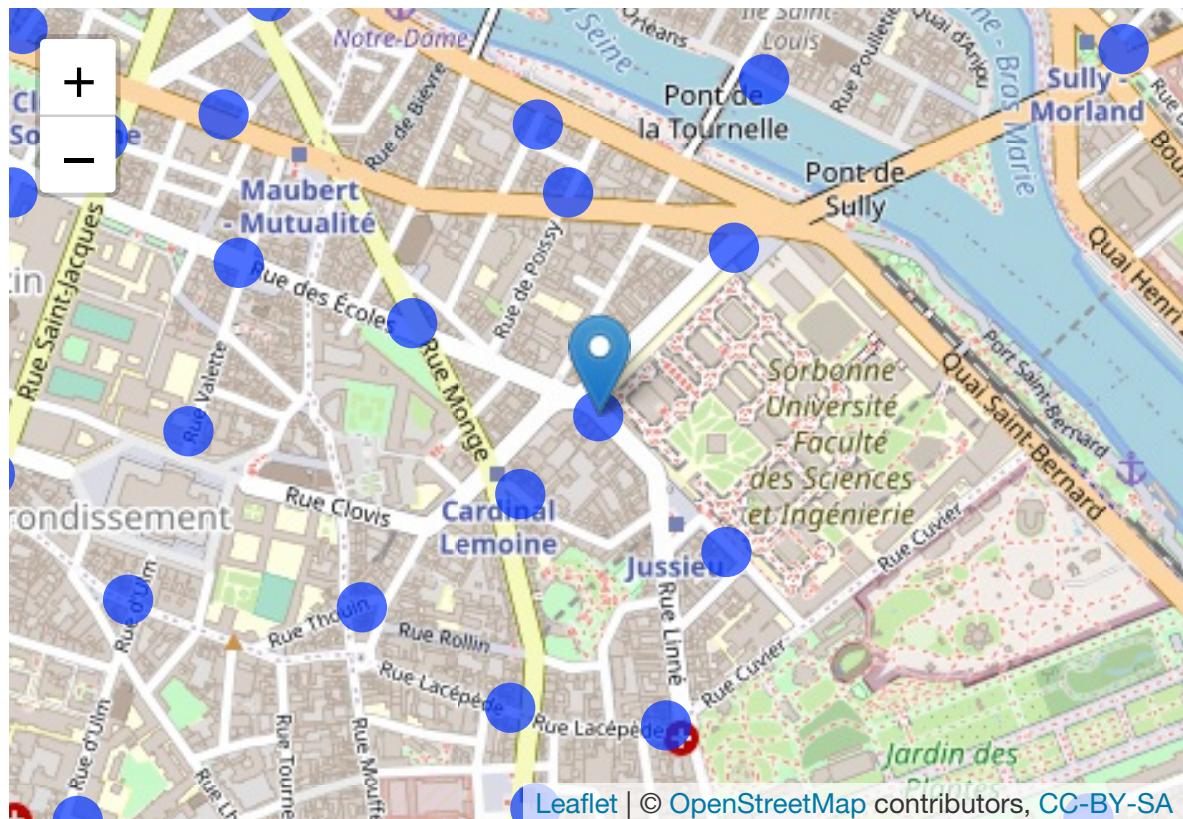
ColorPal1 <- colorNumeric(scales::seq_gradient_pal(low = "#132B43", high = "#56B1F7",
                                                 space = "Lab"), domain = c(0,1))
ColorPal2 <- colorNumeric(scales::seq_gradient_pal(low = "red", high = "black",
                                                 space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

```

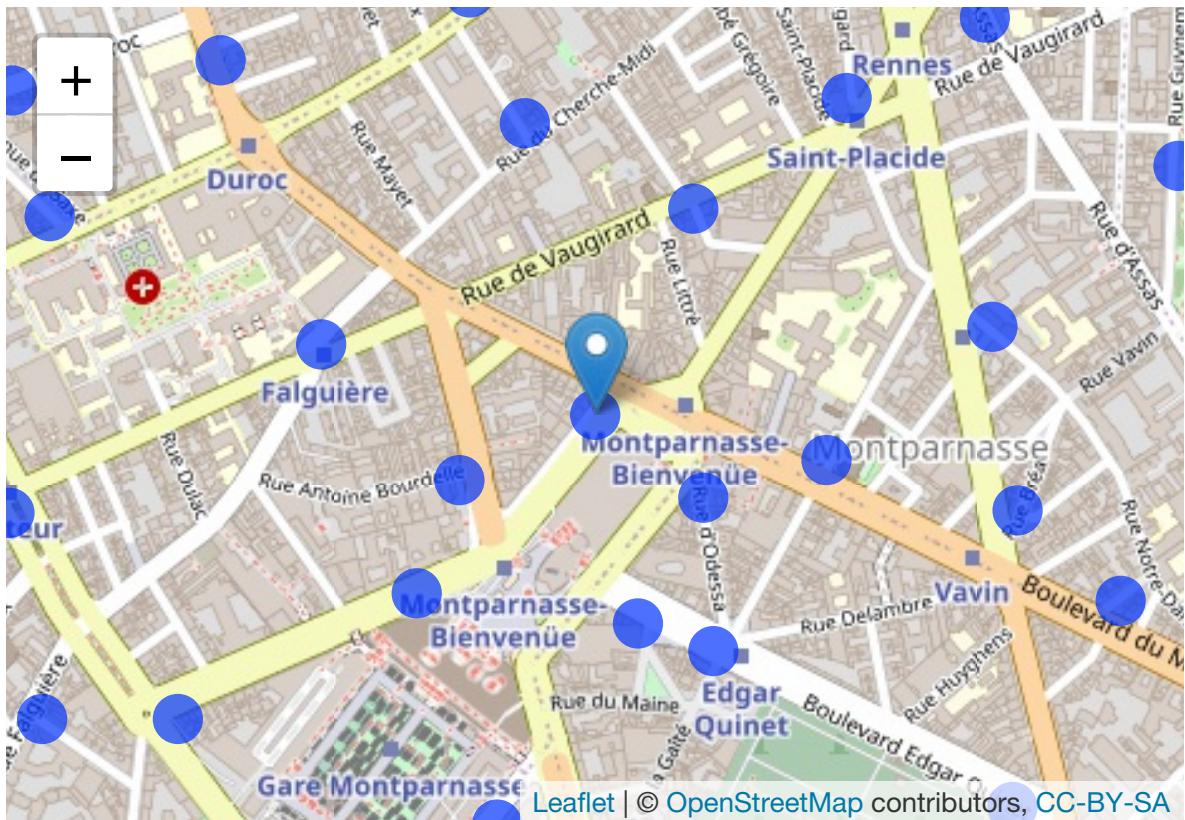
8. Create a **R** function `local.station` which allows to visualize the closest stations of a given station.

For instance

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



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



2.3.2 Temperature map with leaflet

Exercise 2.5 (Challenge).

Redo the temperature map of the first challenge (see section 2.2.1) with **leaflet**. We will use the table build at the end of the challenge with the function `addPolygons`. We can also add a popup to visualize the name of the department and the predicted temperature when we click on the map.

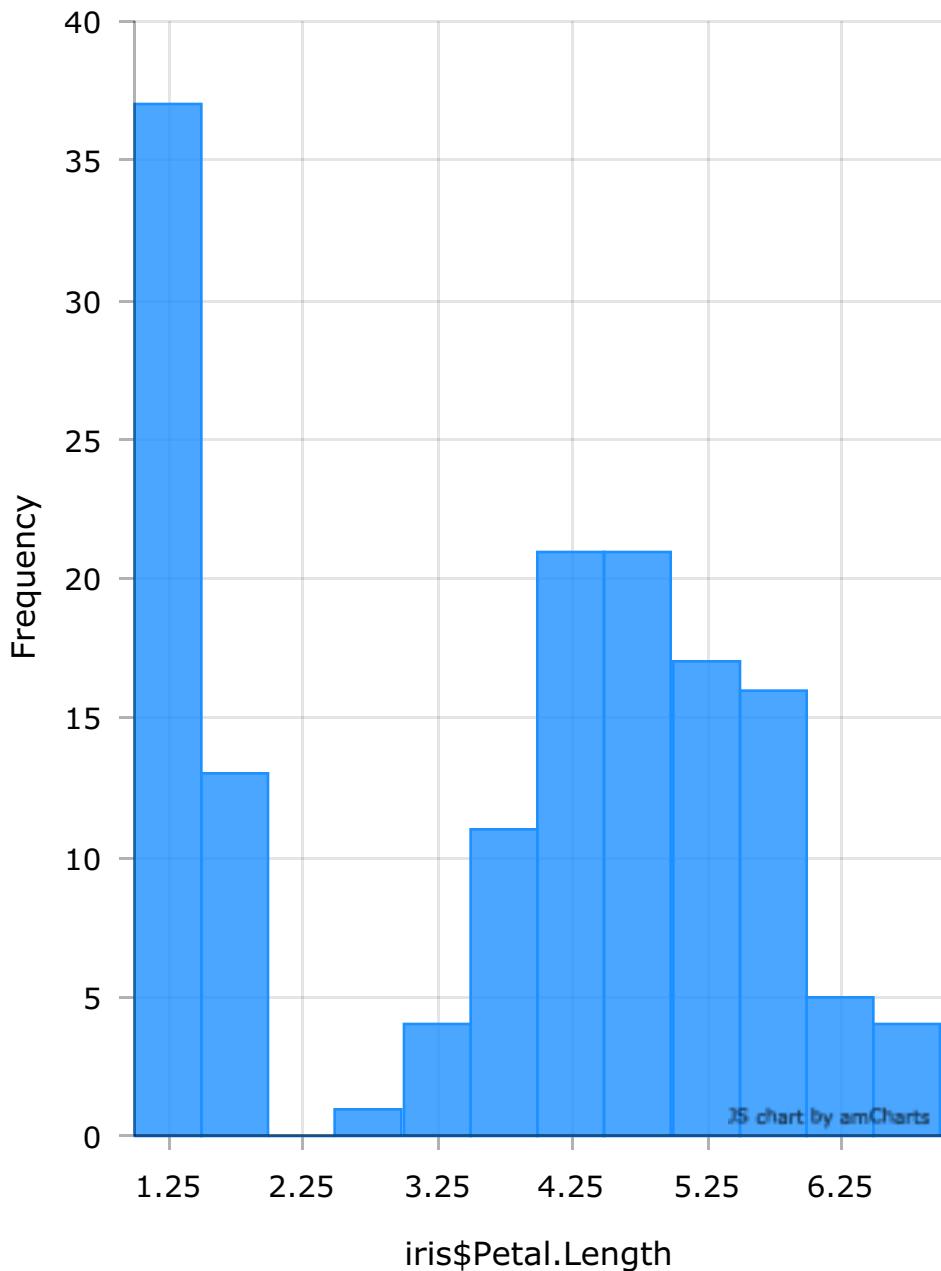
3 Dynamic visualization

As for **leaflet** for mapping, there exists many **R** packages for **dynamic** or **interavtive** visualization. We present some of them in this part.

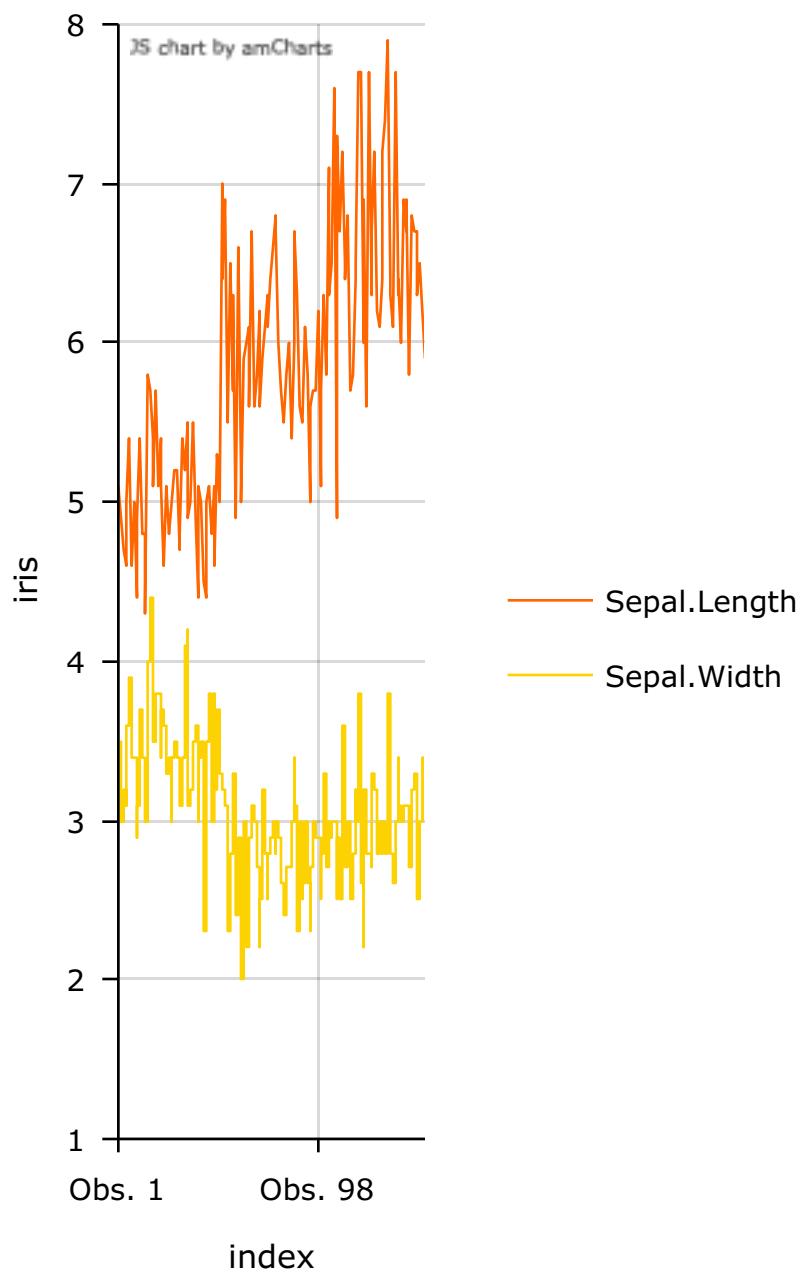
3.1 Classical charts with rAmCharts and plotly

rAmCharts is user-friendly for standard graphs (scatterplot, times series, histogram...). We just have to use classical **R** functions with the prefix **am**. For instance

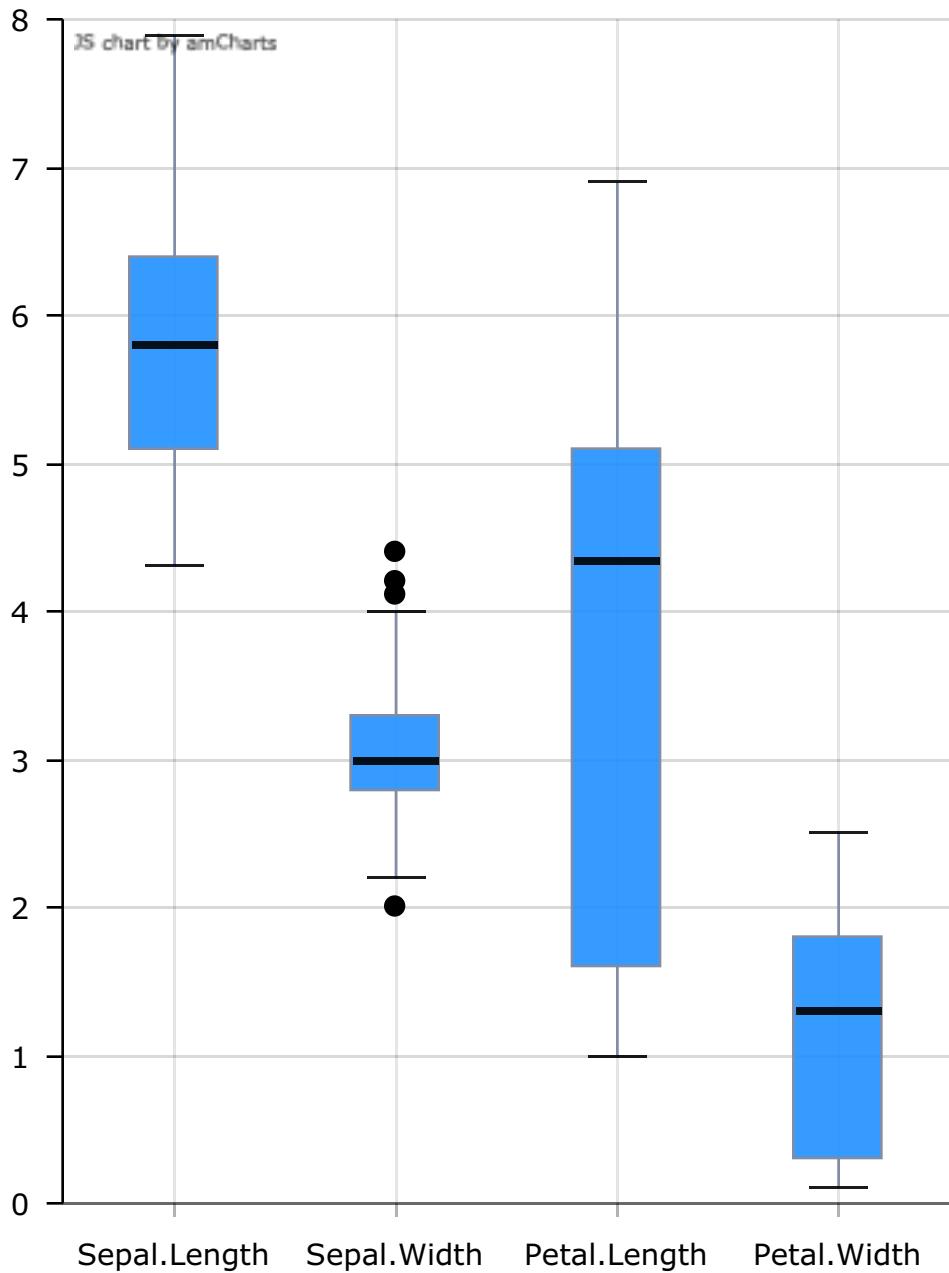
```
library(rAmCharts)
amHist(iris$Petal.Length)
```



```
amPlot(iris, col = colnames(iris)[1:2], type = c("l", "st"),
       zoom = TRUE, legend = TRUE)
```



```
amBoxplot(iris)
```



plotly produces similar things but with a specific syntax. **plotly** commands are expanded into 3 parts :

- dataset and variables (**plot_ly()**);
- additional representations (**add_trace**, **add_markers...**);
- options (axis, titles...) (**layout**).

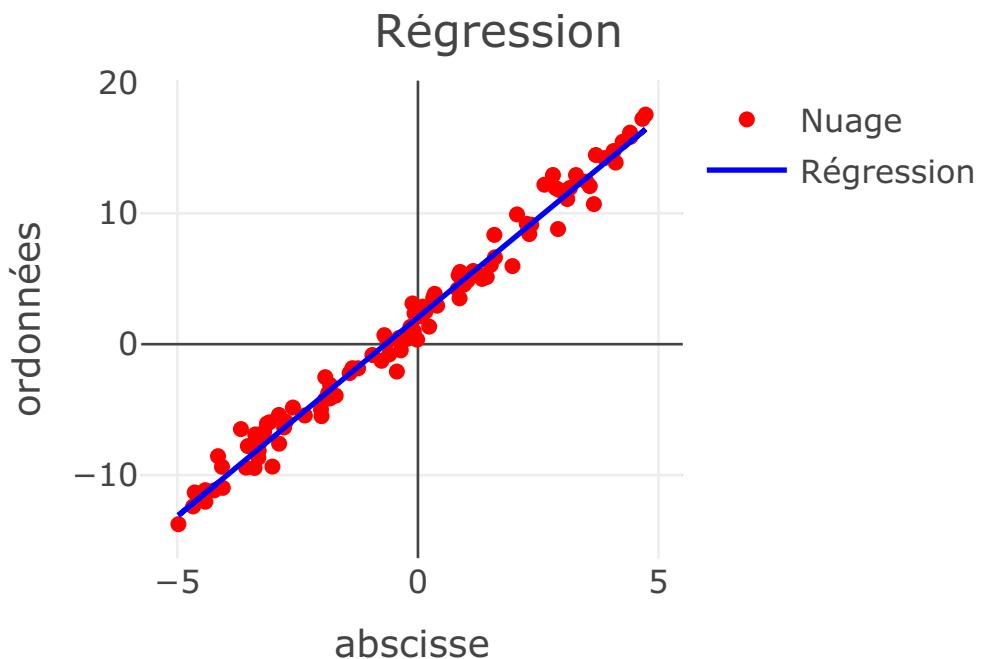
We can find a description for each part at <https://plot.ly/r/reference/>. As a first chart, we propose to represent a scatterplot with the linear smoother. We start by generating the data and computing the linear model :

```
library(plotly)
n <- 100
```

```
X <- runif(n, -5, 5)
Y <- 2+3*X+rnorm(n, 0, 1)
D <- data.frame(X, Y)
model <- lm(Y~X, data=D)
```

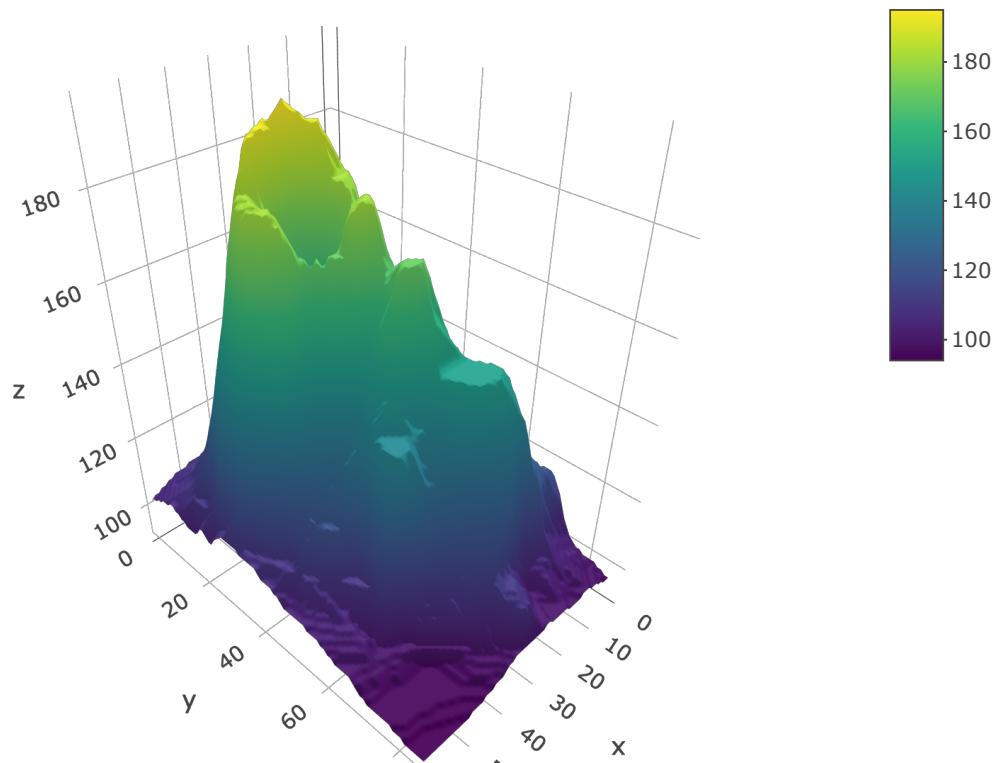
We obtain the required graph with

```
D %>% plot_ly(x=-X, y=-Y) %>%
  add_markers(type="scatter", mode="markers",
              marker=list(color="red"), name="Nuage") %>%
  add_trace(y=fitted(model), type="scatter", mode='lines',
             name="Régression", line=list(color="blue")) %>%
  layout(title="Régression", xaxis=list(title="abscisse"),
         yaxis=list(title="ordonnées"))
```

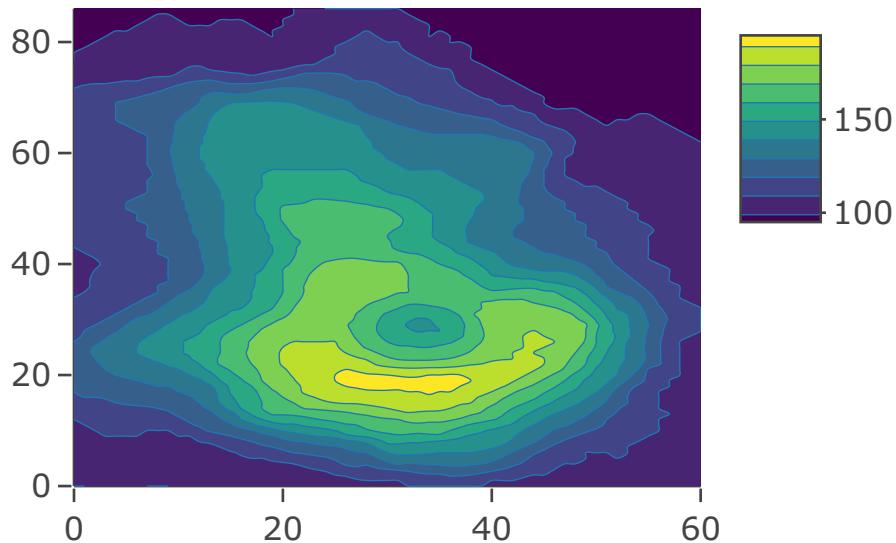


Unlike **ggplot**, we can make 3D with **plotly**. For instance

```
plot_ly(z = volcano, type = "surface")
```

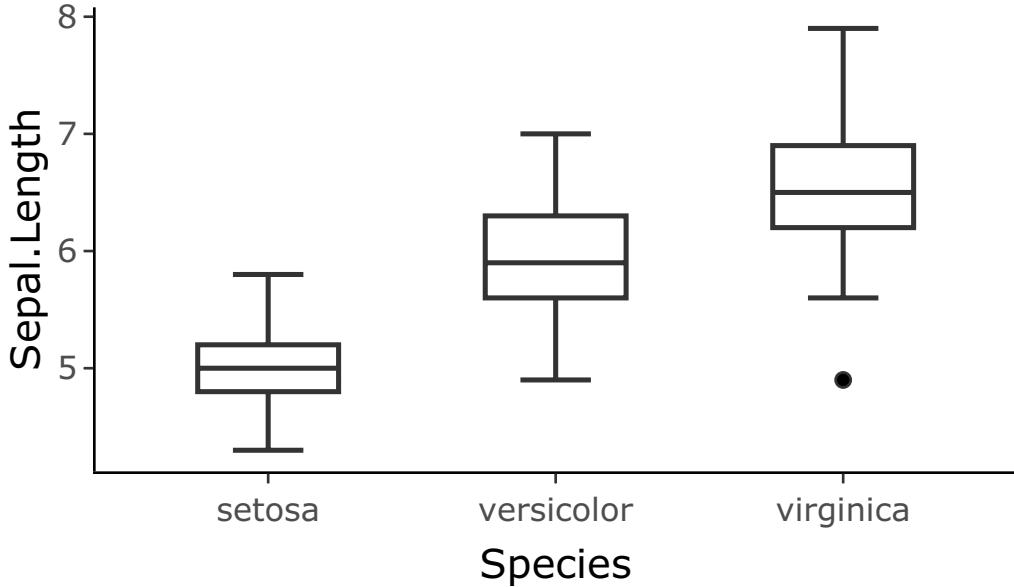


```
plot_ly(z = volcano, type = "contour")
```



We can also convert **ggplot** graph in **plotly** graph with **ggplotly** :

```
p <- ggplot(iris)+aes(x=Species,y=Sepal.Length)+geom_boxplot() +theme_classic()
ggplotly(p)
```



Exercise 3.1 (Basic charts with ‘rAmCharts’ and ‘plotly’).

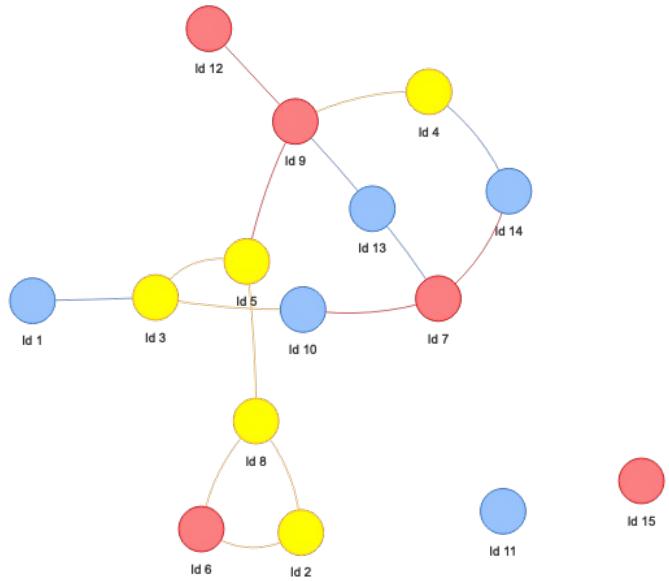
We consider the `iris` dataset. Build the following graph with `rAmCharts` and `plotly`.

1. Scatterplot `Sepal.Length` in term of `Sepal.Width`. Use different colors for each species.
2. Boxplot to visualize the distribution of `Petal.Length` for each species.

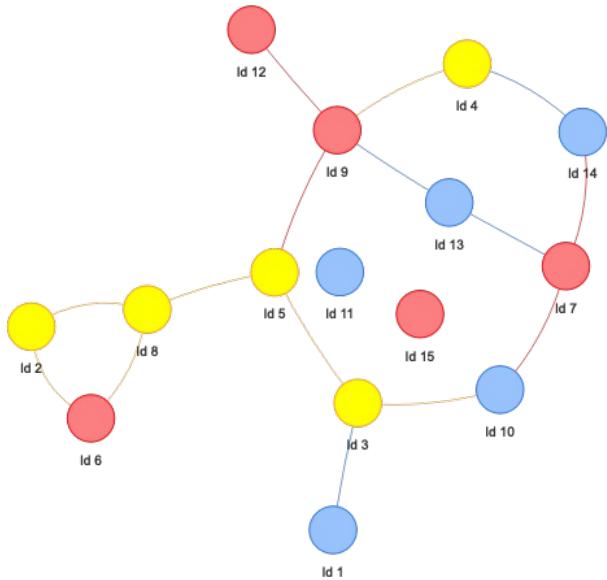
3.2 Graphs to visualize networks with visNetwork

Many datasets can be visualized with graphs, especially when one has to study connexions between individuals. In this case, each individual is represented by a **node** and we use **edges** for the connexions. `igraph` package proposes static representations for graph. For dynamic graphs, we can use `visNetwork`. To obtain dynamic graphs, we first have to specify nodes and edges, for instance

```
nodes <- data.frame(id = 1:15, label = paste("Id", 1:15),
                      group=sample(LETTERS[1:3], 15, replace = TRUE))
edges <- data.frame(from = trunc(runif(15)*(15-1))+1,to = trunc(runif(15)*(15-1))+1)
library(visNetwork)
visNetwork(nodes,edges)
```

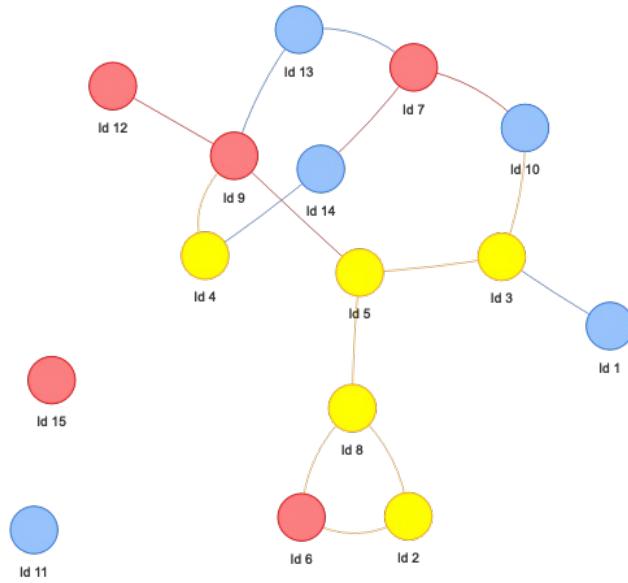


```
visNetwork(nodes, edges) %>% visOptions(highlightNearest = TRUE)
```



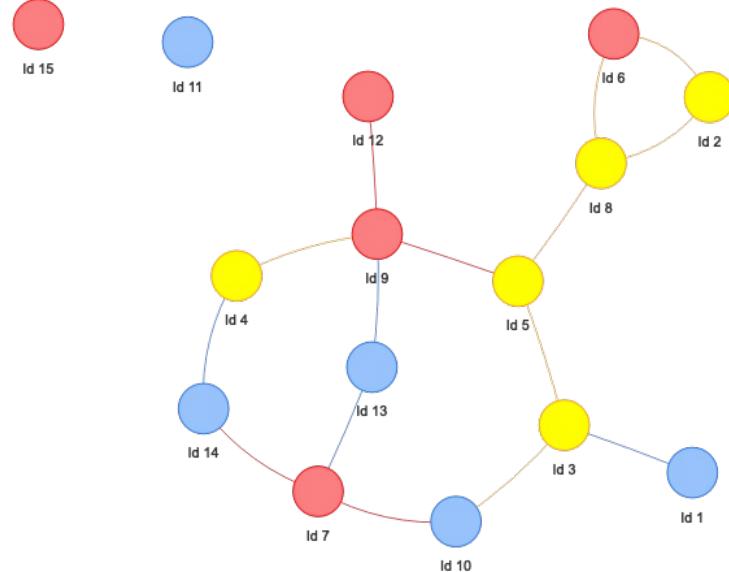
```
visNetwork(nodes, edges) %>% visOptions(highlightNearest = TRUE, nodesIdSelection = TRUE)
```

Select by id ▾



```
visNetwork(nodes, edges) %>% visOptions(selectedBy = "group")
```

Select by group ▾



Exercise 3.2 (Connexions between medias).

We consider a graph which represents connexions between medias. Data are available [here](#). We can import them with

```
nodes <- read.csv("data/Dataset1-Media-Example-NODES.csv", header=T, as.is=T)
links <- read.csv("data/Dataset1-Media-Example-EDGES.csv", header=T, as.is=T)
head(nodes)
  id      media media.type type.label
1 s01    NY Times       1   Newspaper
2 s02 Washington Post     1   Newspaper
```

```

3 s03 Wall Street Journal      1  Newspaper
4 s04          USA Today       1  Newspaper
5 s05          LA Times        1  Newspaper
6 s06          New York Post   1  Newspaper
audience.size
1          20
2          25
3          30
4          32
5          20
6          50
head(links)
  from  to weight      type
1  s01 s02    10 hyperlink
2  s01 s02    12 hyperlink
3  s01 s03    22 hyperlink
4  s01 s04    21 hyperlink
5  s04 s11    22 mention
6  s05 s15    21 mention

```

`nodes` objects represents the nodes (normal) while `links` is for the edges. We can obtain a `graph` object with

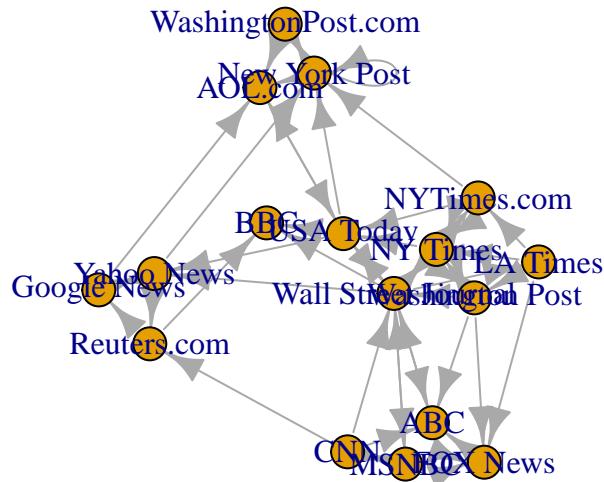
```

library(igraph)
media <- graph_from_data_frame(d=links, vertices=nodes, directed=T)
V(media)$name <- nodes$media

```

and we can visualize the (static) graph with a simple plot :

```
plot(media)
```



1. Visualize this graph with `VisNetwork`. Hint : use `toVisNetworkData`.
2. Add an option which allows to select the type of media (Newspaper, TV or Online).
3. Use different colors for each media.
4. Use arrows with different width in term of the variable `weight`. We can also add the option `visOptions(highlightNearest = TRUE)`.

3.3 Dashboard

Dashboards are very often used in datascience. It is a powerful tool to provide important messages in a dataset and/or a model. We can build dashboard in **R** with the package **flexdashboard**. The syntax is based on **Rmarkdown**, we don't have to learn new tools. We can find a very nice tutorial on this package at <https://rmarkdown.rstudio.com/flexdashboard/>. You can use this tutorial to make the following exercise.

Exercise 3.3 (A Dashboard for linear models).

We consider the dataset **ozone.txt**. The goal is to explain the maximum daily ozone concentration (variable **maxO3**) by the other variables (information about temperatures, nebulosity, wind...). We want to make a dashboard to

- visualize the data : the database and two or three graphs about the output variables (**maxO3**) ;
- visualize simple linear models : we choose one input and we obtain the scatterplot and the linear smoother ;
- visualize the full linear model : a summary of the models with some graphs about the residuals ;
- select the inputs in the linear models ;
- ...

1. As a first step, we propose to write some simple functions for the dashboard.
 - a. We only consider numeric variables. Visualize correlations between the variables with the **corrplot** function of the **corrplot** package.
 - b. Draw the histogram of **maxO3** with **ggplot**, **rAmCharts** and **plotly** (use **ggplotly**).
 - c. Fit the linear model with output **maxO3** (all the other variables as input). Calculate the Studentized residuals (**rstudent**) and visualize these residuals in term of **maxO3**. You can also add a linear smoother on the graph.
2. We can now start the dashboard. Use **File -> Rmarkdown -> From Template -> Flex Dashboard** dialog to open a script.
 - a. Build a first dashboard which allows to visualize
 - the dataset on a column (use **datatable** function from **DT** package) ;
 - the histogram of **maxO3** and the correlation matrix on a second column.
 - b. Add a second page to visualize simple linear models. We can select the input with **Shiny**, use

```
radioButtons("variable1",
             label="Choisir la variable explicative",
             choices=names(df)[-1],
             selected=list("T9"))
```

Don't forget to add

```
runtime: shiny
```

in the header.

- c. Add a page to visualize the full linear model and the graph of residuals defined at question 1.c.
- d. Add a last page for selecting the inputs variables. Here again, you can use **shiny** commands, for instance

```
checkboxGroupInput("variable",
                   label="Choisir la variable",
                   choices=names(df)[-1],
                   selected=list("T9"))
```

For the selected inputs, we will see in this pages the coefficients of the linear model and the Studentized residuals. The final dashboard may look like



It is available at <https://lrouviere.shinyapps.io/dashboard/>.

4 Shiny web applications

This part comes from the tutorial of Benoît Thieurmel. It is available here https://github.com/datastorm-open/tuto_shiny_rennes.

4.1 First step

Create a new repository for the application (**in RStudio**) :

File → **New Project** → **New Directory** → **Shiny Web Application**

Choose an application **Multiple File**.

If the option **Multiple File** is not possible, you can use

File > **New File** > **Shiny Web App** > **Multiple File**

Two files are automatically created : **ui.R** and **server.R**. Run the application with the button **Run App**.

- Change the title of the application. You can call it **My first application**.
- Update the application and check the new title.

4.2 Input - output

For the moment, we will add :

- new **inputs** in `sidebarPanel`, after the `sliderInput`. Don't forget to add commas for separate the inputs;
- new **outputs** in `mainPanel`, after the `plotOutput`. Don't forget to add commas for separate the outputs.

1. Add in `ui.R` an input which allow to change the color of the histogram. You can use this command :

```
selectInput(inputId = "color", label = "Couleur :",
            choices = c("Rouge" = "red", "Vert" = "green", "Bleu" = "blue"))
```

2. Add a new output to visualize the `summary` of the dataset `faithful`. You can use :

```
# ui.R
verbatimTextOutput("summary")

# server.R
output$summary <- renderPrint({
  summary(faithful)
})
```

Exercise 4.1 (Adding somme inputs/outputs).

Add some inputs and outputs on the previous application to

1. propose to the user to change the title of the histogram (use a `textInput` in the `ui` and the option `main` in the histogram) ;
2. select the variable of `faithful` for the histogram (use a `radioButtons` with choices `colnames(faithful)`) ;
3. visualize the whole dataset `faithful` (`renderDataTable` & `dataTableOutput`) ;
4. add a text under the histogram which indicates the number of class (use `renderText` and `paste` functions in `server`, ‘textOutput“ in `ui`) .
5. Replace the `selectInput` of the color choice by a `colourInput` (use package `colourpicker`) ;
6. Add an input to export the graph (`downloadButton` & `jpeg`).

4.3 Organizing the application

Consider the application `app_structure` It is quite the same as the previous one with a `navbarPage` :

- a tab *Data* to visualize the data (table + summary)
- a tab *Visualisation* : inputs + histogramm.

Exercise 4.2 (Structure of an application).

1. Tab **Data** : use `navlistPanel` to separate the `summar` and the `table` into two tabs :

```
# rappel de la structure (ui.R)
navlistPanel(
  "Title of the structure",
  tabPanel("Title of the tab", ... "(content of the tab"),
  tabPanel("Title of the tab", ... "(content of the tab"))
)
```

2. Tab **Visualisation** : change `sidebarLayout - sidebarPanel - mainPanel` by a `fluidRow` with 2 columns :

- 1/4 : for the `sidebarPanel`
- 3/4 : for the `mainPanel`.

```

fluidRow(
  column(width = 3, ...), # column 1/4 (3/12)
  column(width = 9, ...)  # column 3/4 (9/12)
)

```

Hint : use wellPanel for the left column.

3. Add a boxplot in the visualisation tab (same variable and same color). You can also use `tabsetPanel` to obtain two different tabs for the histogram and the boxplot.

```

# rappel de la structure (ui.R)
tabsetPanel(
  tabPanel("Title of the tab", ... "(content of the tab"),
  tabPanel("Title of the tab", ... "(content of the tab")
)

```

Taking things further : use `shinydashboard` (<https://rstudio.github.io/shinydashboard/>) for building the same application.

4.4 Adding interactive charts

In the previous application, replace histogram and boxplots by javascript charts obatin with `rAmCharts`. You can use

```

# server.R
output$distPlot <- renderAmCharts({
  x      <- faithful[, input$var]
  bins <- round(seq(min(x), max(x), length.out = input$bins + 1), 2)

  amHist(x = x, control_hist = list(breaks = bins),
         col = input$error, main = input$titre,
         export = TRUE, zoom = TRUE)
})

# ui.R
amChartsOutput("distPlot")

```

4.5 Reactive, isolation, observe, html, ...

For the previous application

- add `actionButton` combined with `isolate` to update the application when we click on a button.
- Use `observeEvent` to force the visualization of the histogram when we update the application. You can use

```

# think to add "session"
shinyServer(function(input, output, session)

  # an id
  tabsetPanel(id = "viz",
  tabPanel("Histogramme", ...

  # and finally
  observeEvent(input$go, {
    updateTabsetPanel(session, inputId = "viz", selected = "Histogramme")
  })
)

```

- Use `reactive` function to store the selected variable

```
# rappel de la syntaxe
data <- reactive({
  ...
})

output$plot <- renderPlot({
  # recuperation des donnees
  x <- data()
  ...
})
```

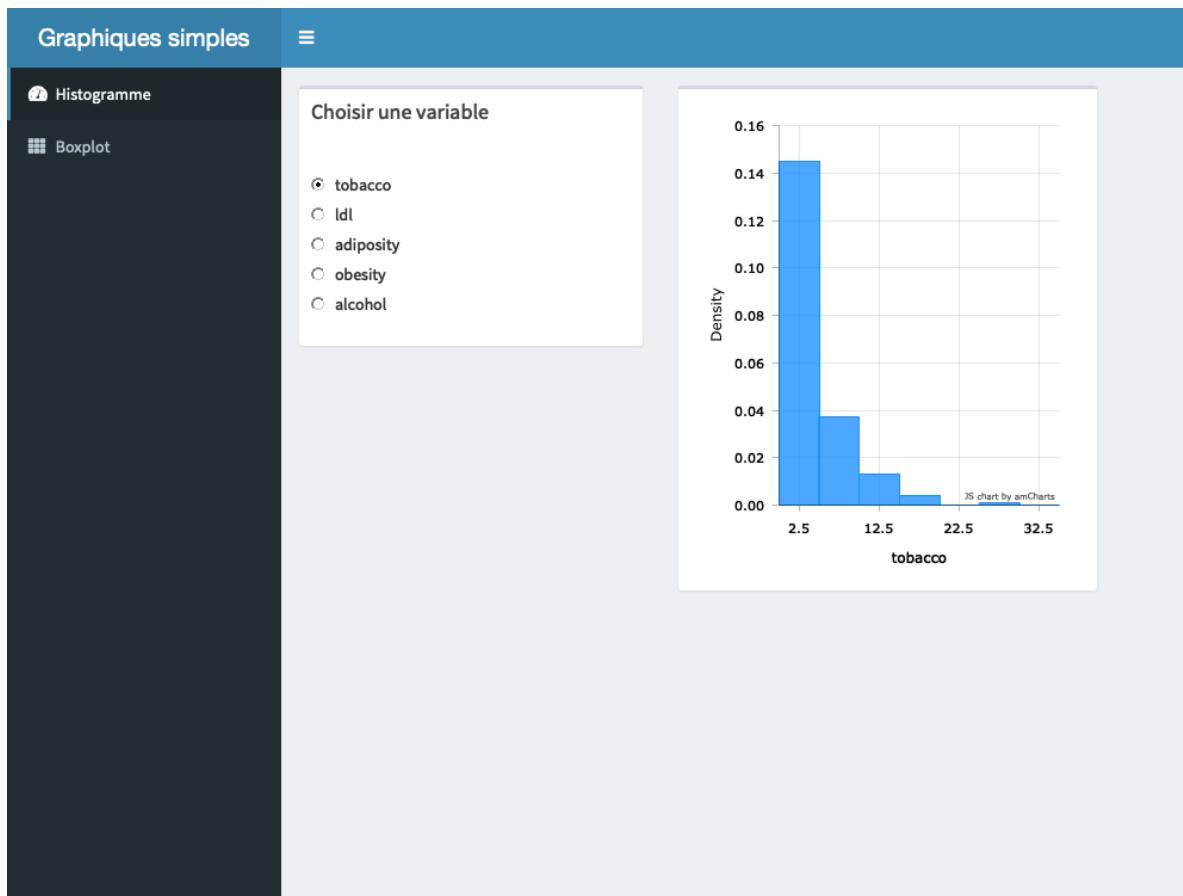
- Add a title on the dataset with blue color. You can use `h1` and a `css` style.
- Add a third tab where you present a summary of you. You can also add an image and a link to a website.
- **Taking things further** : change the rheme of the application with a `.css` file. You can use bootswatch for instance <http://bootswatch.com/3>.

4.6 Some additional exercises

Exercise 4.3 (Descriptive statistics).

We consider the dataset **SAheart** from the **bestglm** package.

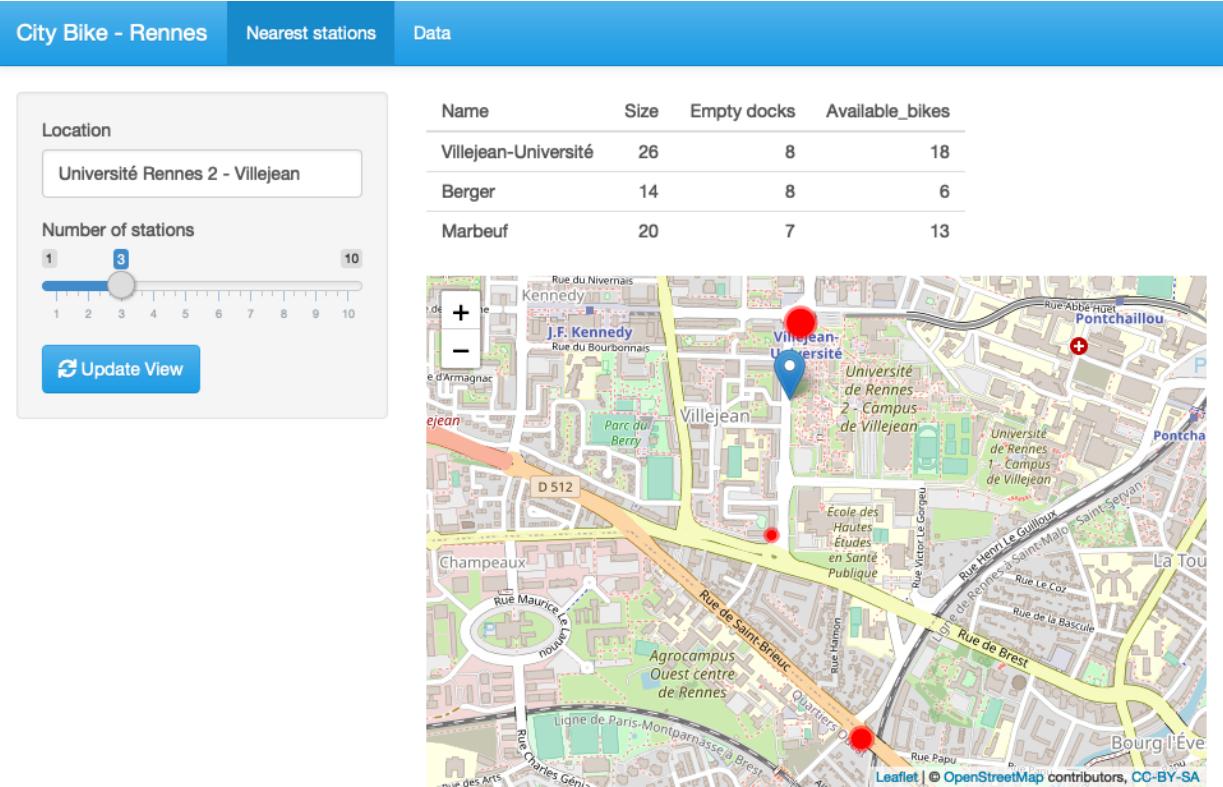
1. Use **rAmCharts** to visualize histograms of the numeric variables of the dataset and boxplots in terms of the variable `chd`.
2. Create a shiny web application which allow to
 - select one variable among the numeric variables of the dataset. You can use `radioButtons` with
`choices=names(SAheart)[sapply(SAheart,class)=="numeric"]`
 - visualize the histogram and the boxplot (in terms of `chd`) of the selected variable. These graphs should be done with **rAmCharts** (use `amChartsOutput`).
 - The required application may look like



It is also available at https://lrouvriere.shinyapps.io/DESC_APP.

Exercise 4.4 (Bije stations in Rennes).

Create an application which allows to visualize bike stations in Rennes. It may look like



It is also available at <https://lrouviere.shinyapps.io/velib/>. You can obtain the data on the Rennes métropole website : <https://data.rennesmetropole.fr/explore/dataset/etat-des-stations-le-velo-star-en-temps-reel/export/>