

Tutoriel : visualisation avec R

Laurent Rouvière

2020-05-30

Contents

Présentation	1
1 Visualisation avec ggplot2	1
1.1 Fonctions graphiques conventionnelles	2
1.2 La grammaire ggplot2	14
1.3 Compléments	43
1.4 Quelques exercices supplémentaires	51
2 Faire des cartes avec R	71
2.1 Le package ggmap	71
2.2 Cartes avec contours, le format shapefile	78
2.3 Cartes interactives avec leaflet	87
3 Quelques outils de visualisation interactive - Compléments shiny	104
3.1 Représentations classiques avec rAmCharts et plotly	104
3.2 Graphes pour visualiser des réseaux avec visNetwork	114
3.3 Dashboard	119

Présentation

Ce tutoriel présente quelques outils **R** pour la **visualisation de données**. Il nécessite des connaissances de base en **R** et en programmation et se structure en 3 parties :

- **Visualisation avec ggplot2** : présentation du package **ggplot2** pour faire des représentations graphiques avec **R** ;
- **Introduction à la cartographie** : construction de cartes avec les packages **ggmap**, **sf** et **leaflet** ;
- **Visualisation interactive** : présentation de packages qui permettent de faire facilement des graphes interactifs, des tableaux de bord ou des applications web (**shiny**).

On pourra trouver des transparents associés à ce tutoriel ainsi que les données utilisés à l'adresse suivante <https://lrouviere.github.io/VISU/>.

1 Visualisation avec ggplot2

Il est souvent nécessaire d'utiliser des techniques de visualisation à toutes les étapes d'une étude statistique. Un des avantages de **R** est qu'il est relativement simple de mettre en oeuvre 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.



1.1 Fonctions graphiques conventionnelles

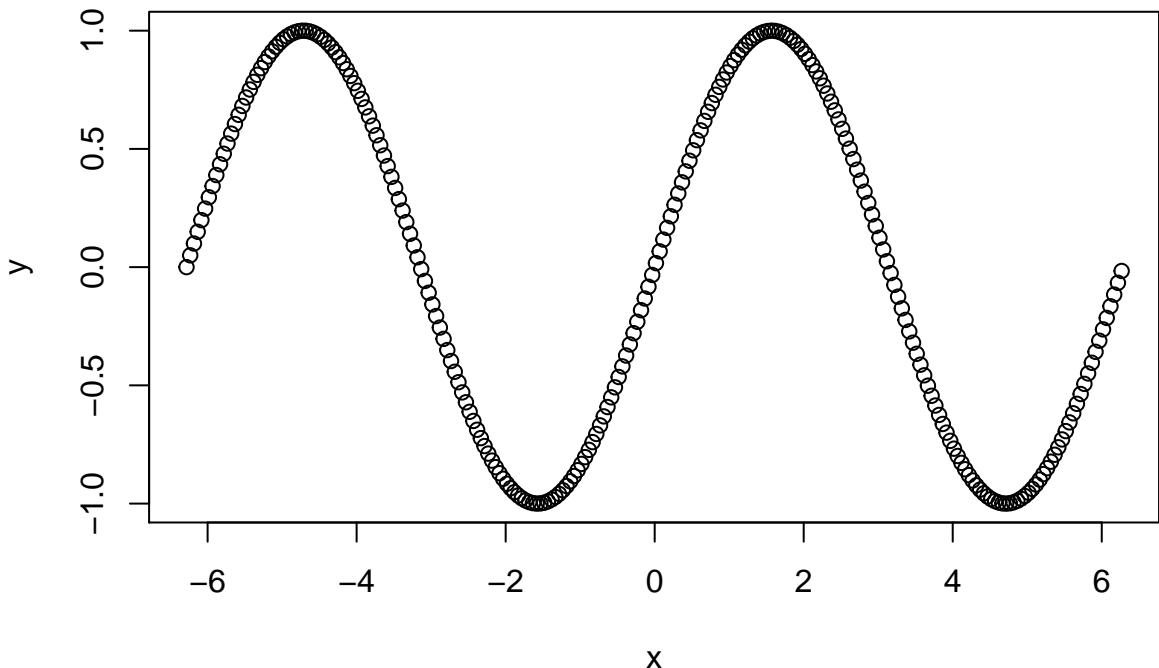
Pour commencer il est intéressant d'examiner quelques exemples de représentations graphiques construits avec **R**. On peut les obtenir à l'aide de la fonction **demo**.

```
> demo(graphics)
```

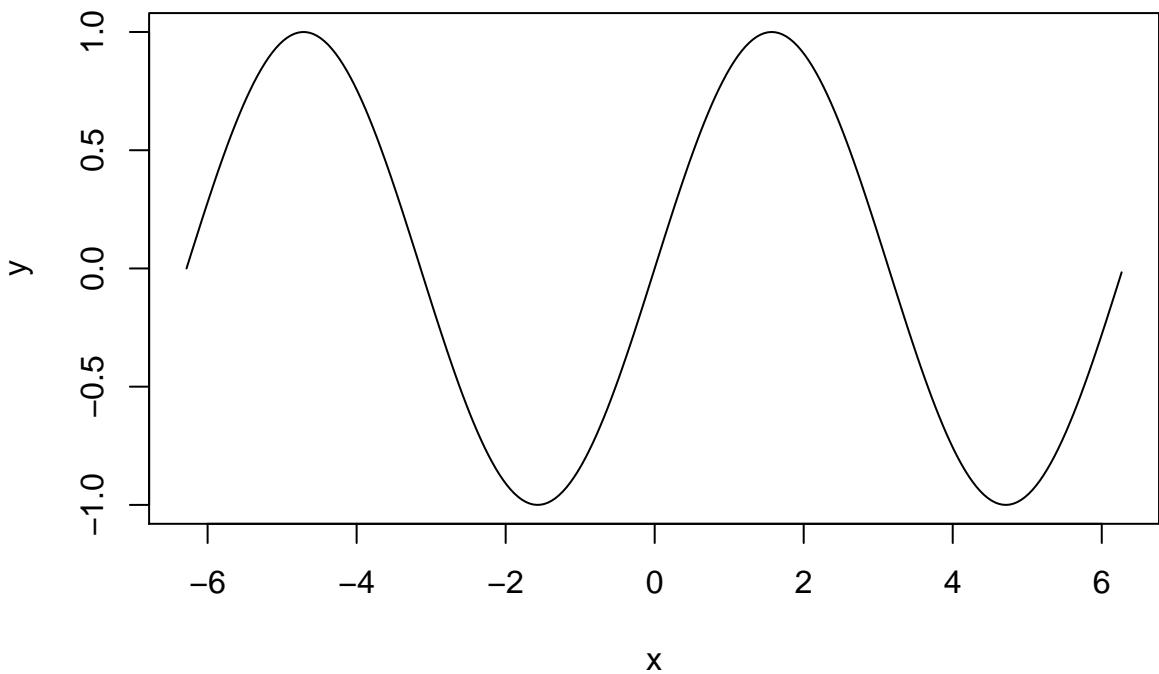
1.1.1 La fonction plot

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

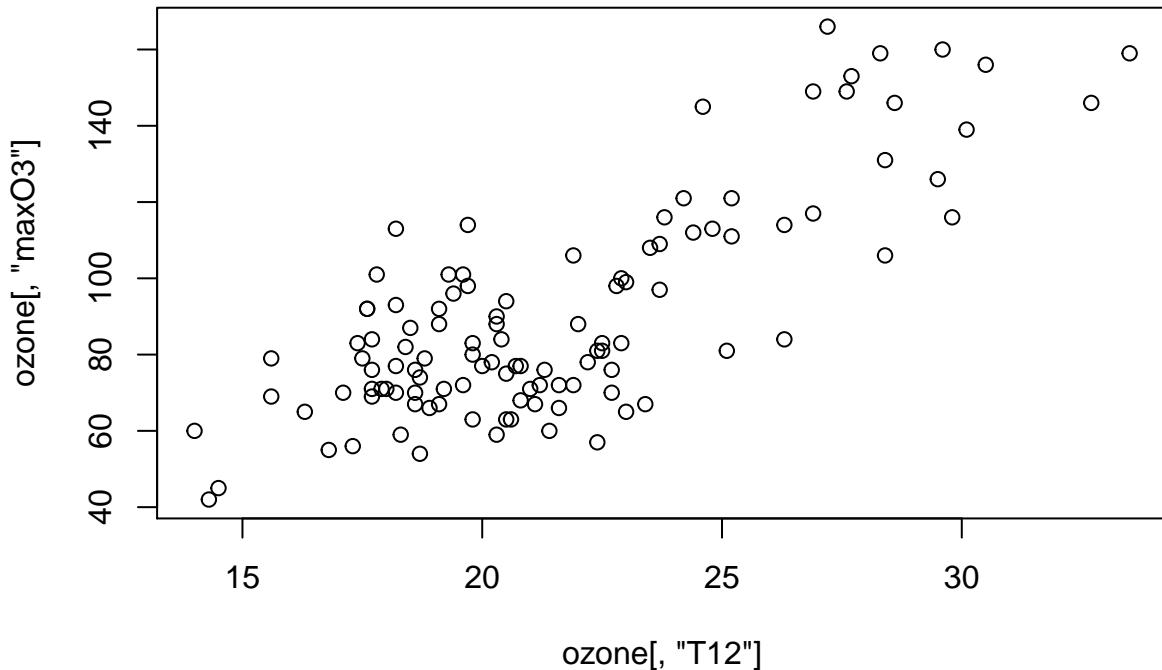
```

##   3rd Qu.:106.00   3rd Qu.:19.93   3rd Qu.:23.55   3rd Qu.:25.40
##   Max.    :166.00   Max.    :27.00   Max.    :33.50   Max.    :35.50
##   Ne9          Ne12          Ne15          Vx9
##   Min.    :0.000   Min.    :0.000   Min.    :0.00   Min.    :-7.8785
##   1st Qu.:3.000   1st Qu.:4.000   1st Qu.:3.00   1st Qu.:-3.2765
##   Median :6.000   Median :5.000   Median :5.00   Median :-0.8660
##   Mean    :4.929   Mean    :5.018   Mean    :4.83   Mean    :-1.2143
##   3rd Qu.:7.000   3rd Qu.:7.000   3rd Qu.:7.00   3rd Qu.: 0.6946
##   Max.    :8.000   Max.    :8.000   Max.    :8.00   Max.    : 5.1962
##   Vx12          Vx15          max03v        vent      pluie
##   Min.    :-7.878   Min.    :-9.000   Min.    : 42.00  Est     :10   Pluie:43
##   1st Qu.:-3.565   1st Qu.:-3.939   1st Qu.: 71.00  Nord    :31   Sec   :69
##   Median :-1.879   Median :-1.550   Median : 82.50  Ouest   :50
##   Mean    :-1.611   Mean    :-1.691   Mean    : 90.57  Sud    :21
##   3rd Qu.: 0.000   3rd Qu.: 0.000   3rd Qu.:106.00
##   Max.    : 6.578   Max.    : 5.000   Max.    :166.00

```

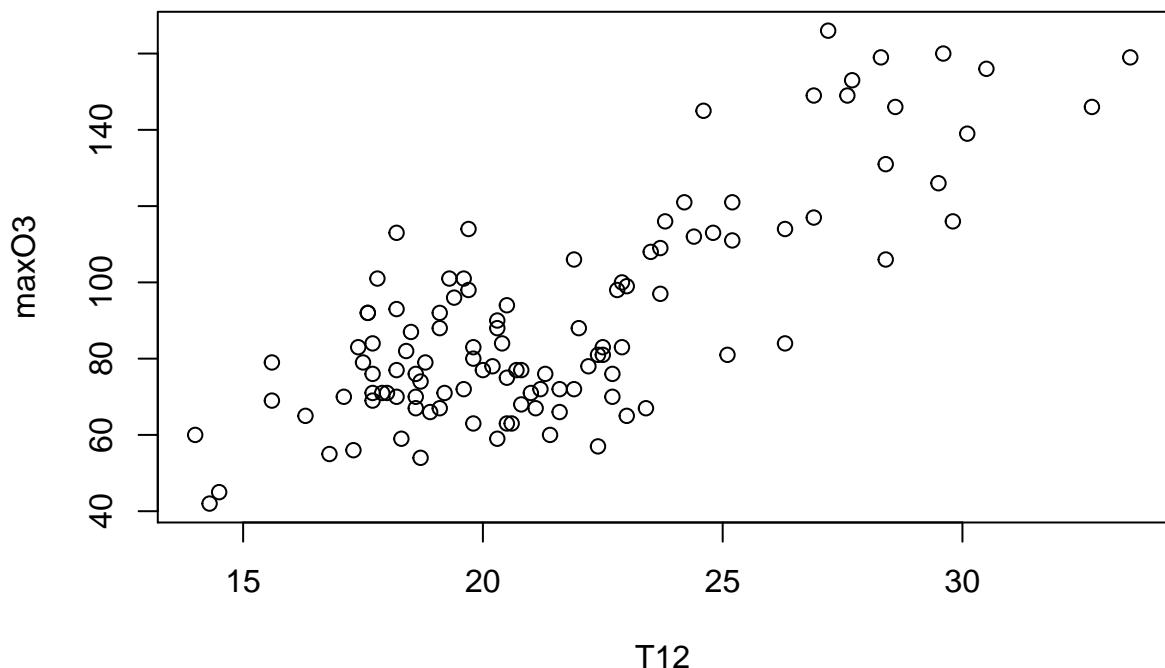
On visualise tout d'abord 2 variables quantitatives à l'aide d'un nuage de points : la concentration en ozone maximale **maxO3** en fonction de la température à 12h **T12**.

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



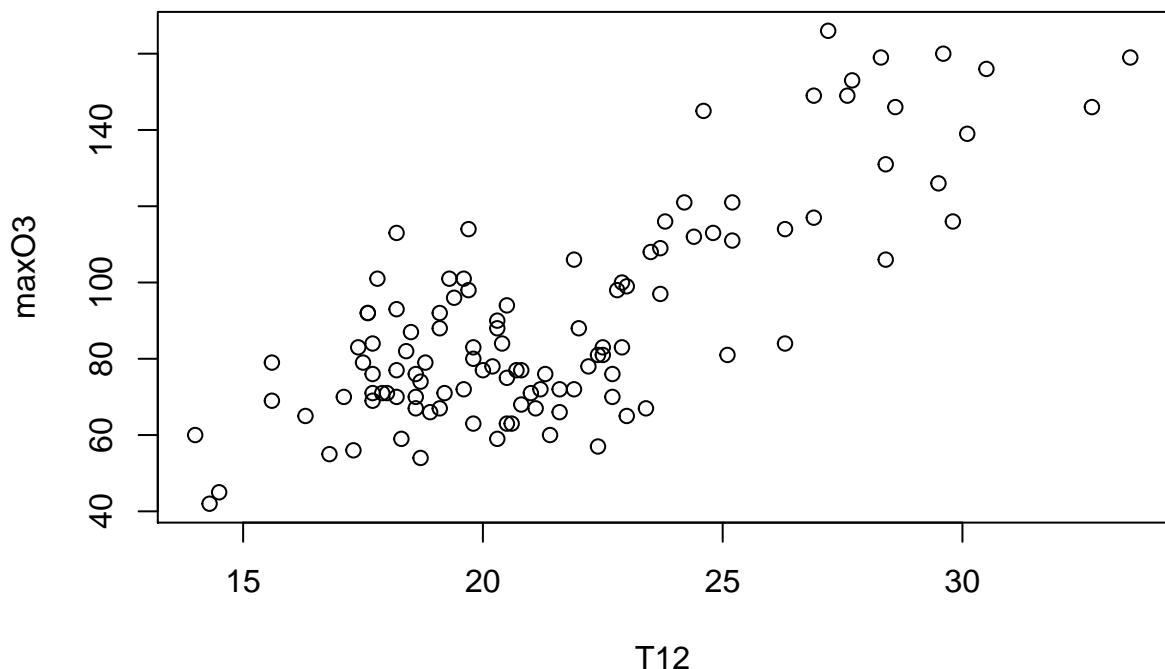
Comme les deux variables appartiennent au même jeu de données, on peut obtenir la même représentation à l'aide d'une syntaxe plus claire qui ajoutent automatiquement les noms des variables sur les axes :

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



Une autre façon de faire (moins naturelle) :

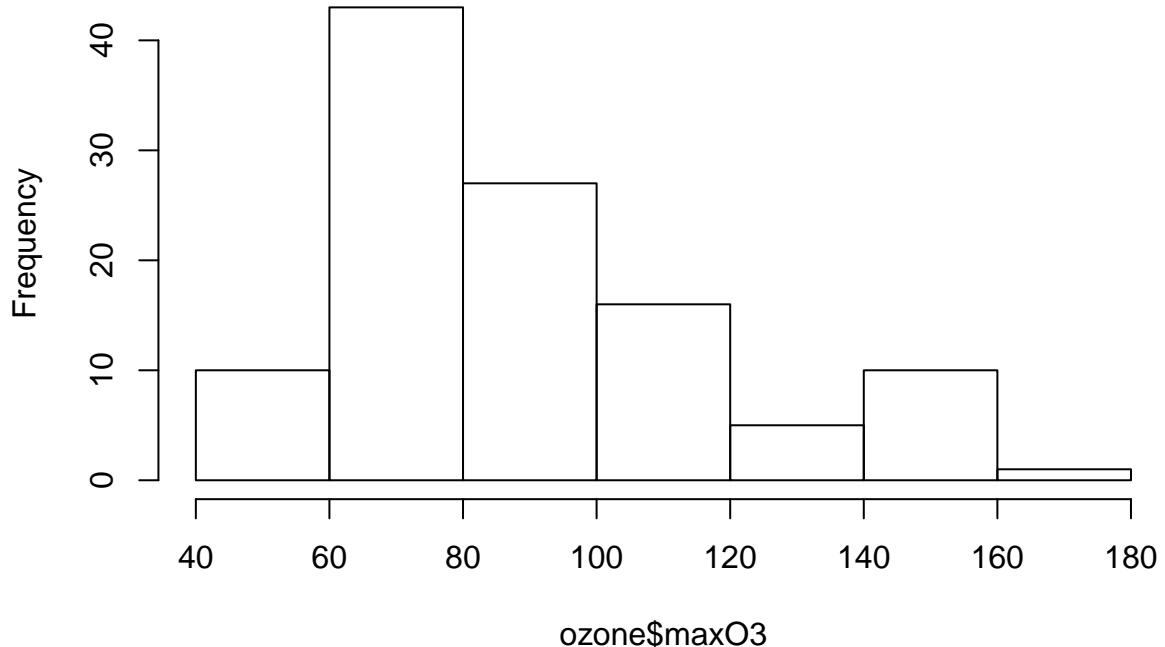
```
> plot(ozone[, "T12"], ozone[, "maxO3"], xlab="T12", ylab="maxO3")
```



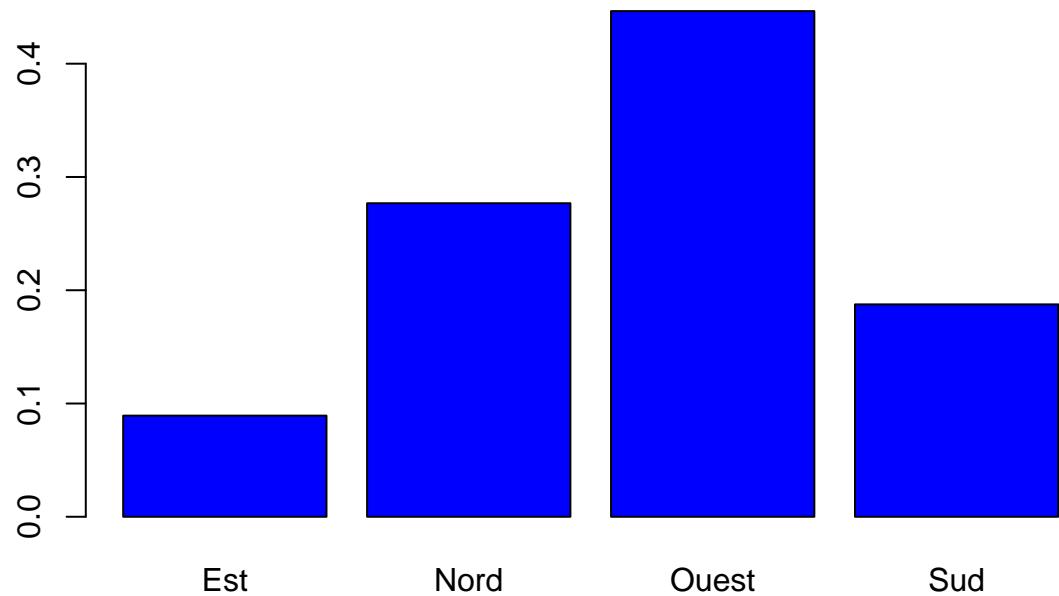
Il existe des fonctions spécifiques pour chaque type de graphs, par exemple **histogram**, **barplot** et **boxplot** :

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

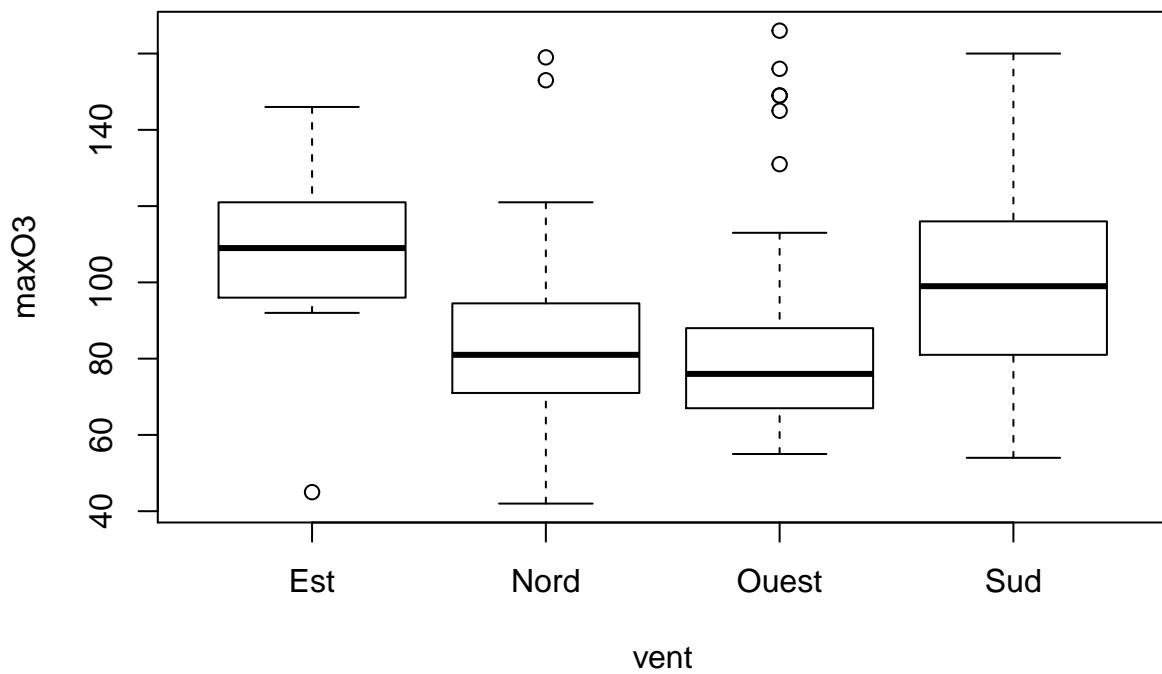
Histogram



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



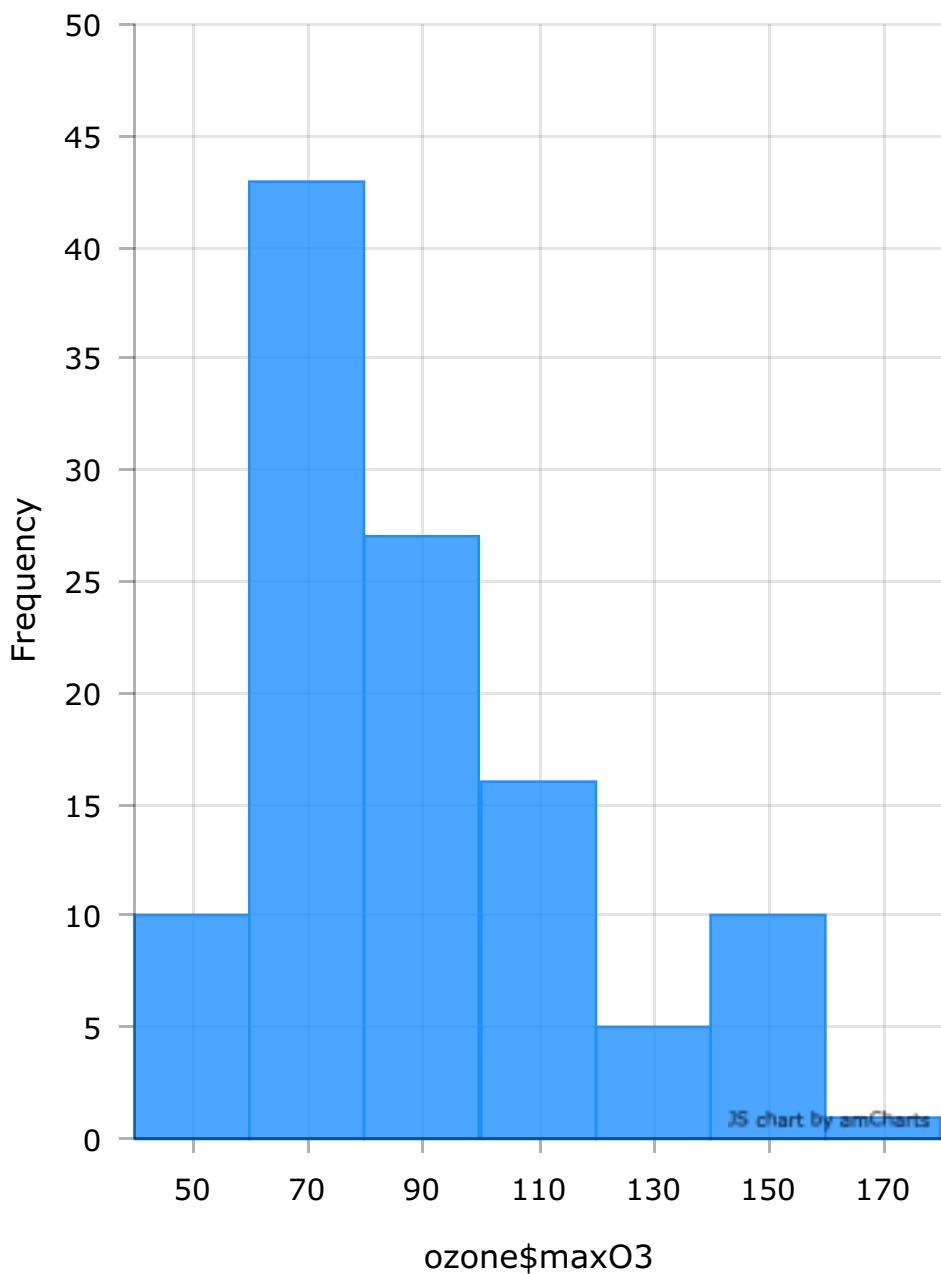
```
> boxplot(max03~vent, data=ozone)
```



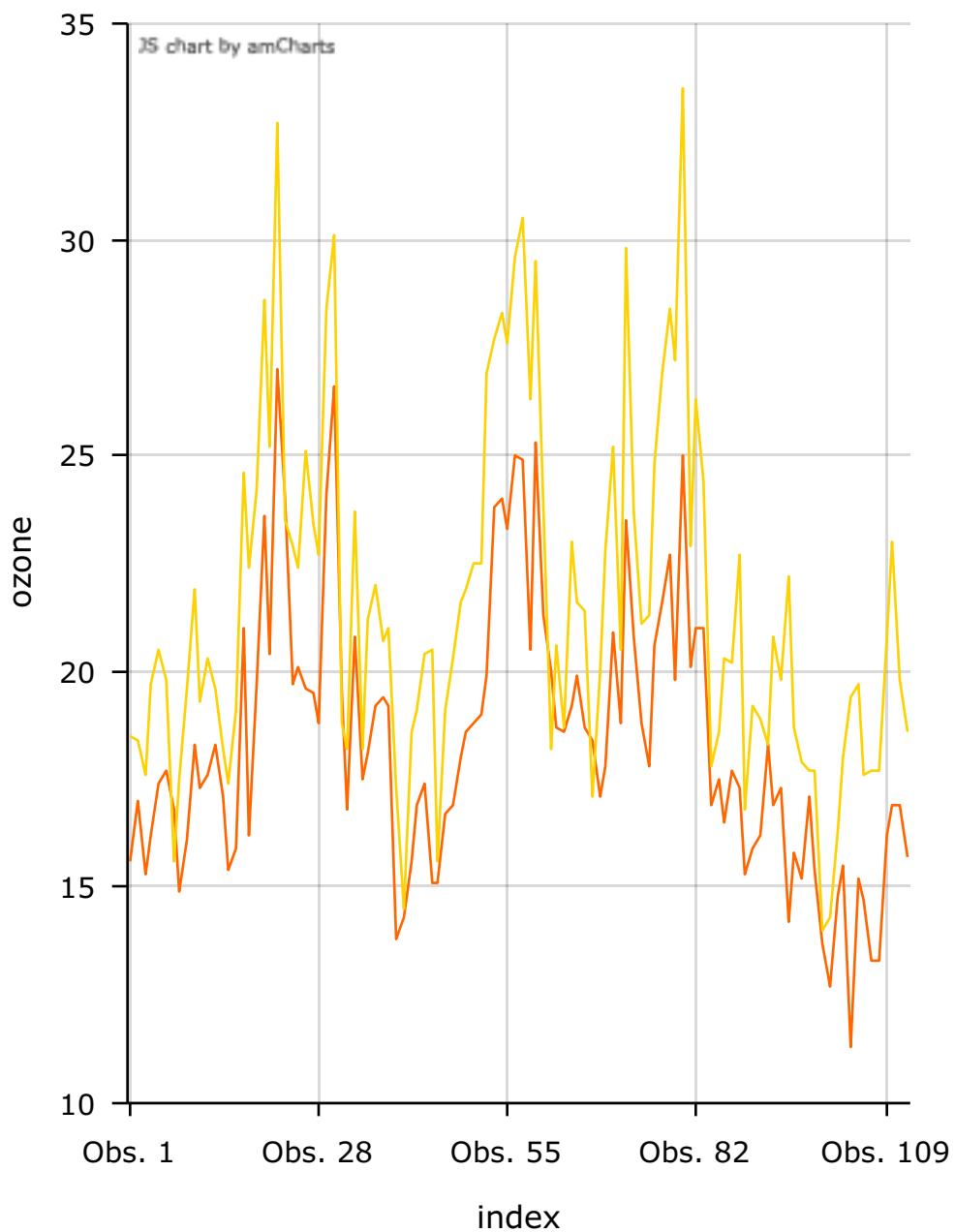
1.1.2 Graphes interactifs avec rAmCharts

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

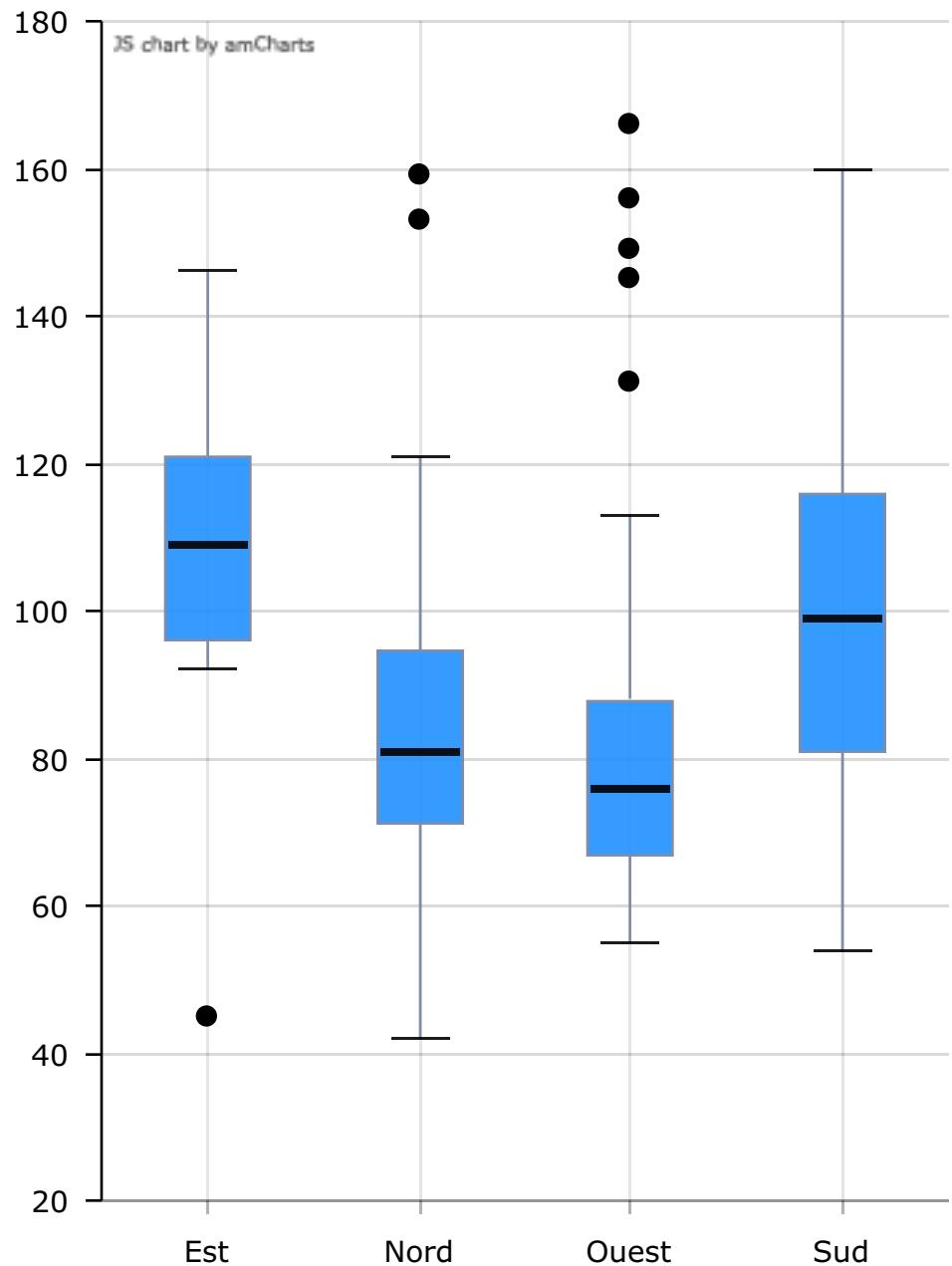
```
> library(rAmCharts)
> amHist(ozone$maxO3)
```



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



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

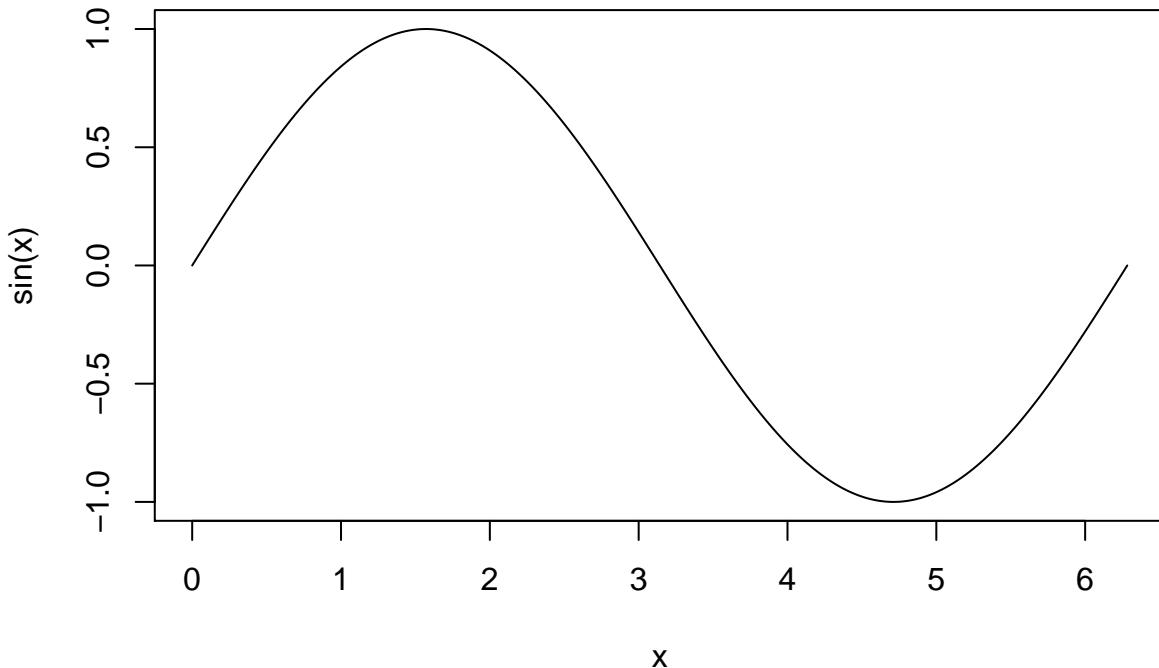


Exercice 1.1 (Premier graphe).

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**.

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

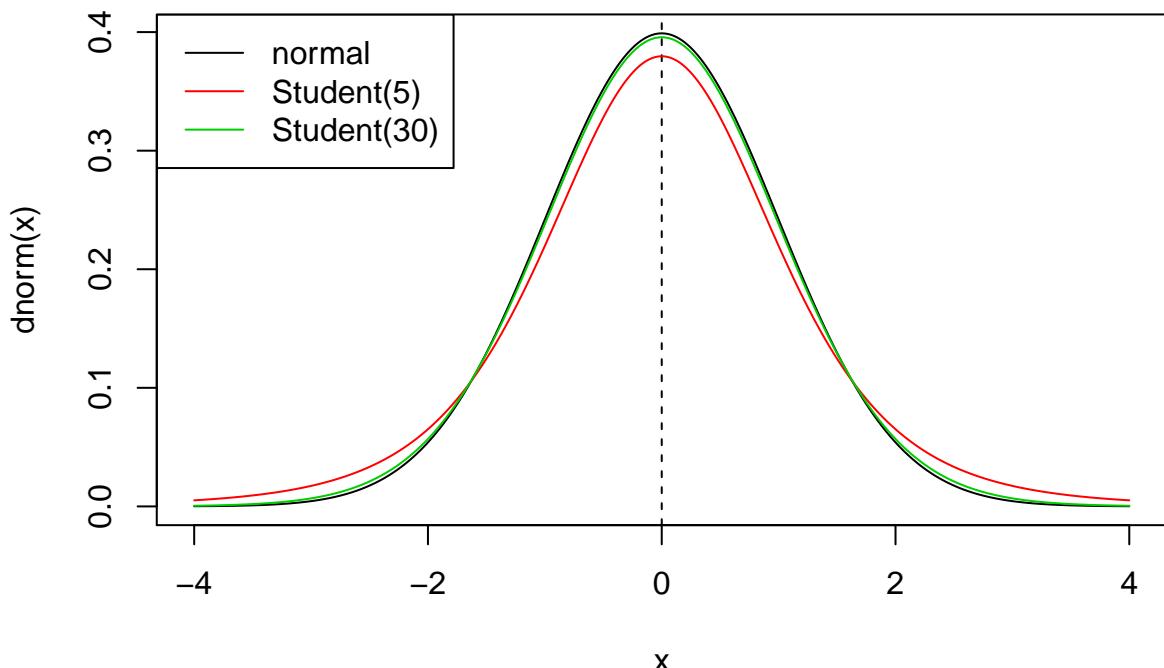
Représentation de la fonction sinus



Exercice 1.2 (Tracé de densités).

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

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



Exercice 1.3 (Tâches solaires).

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

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

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

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

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

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

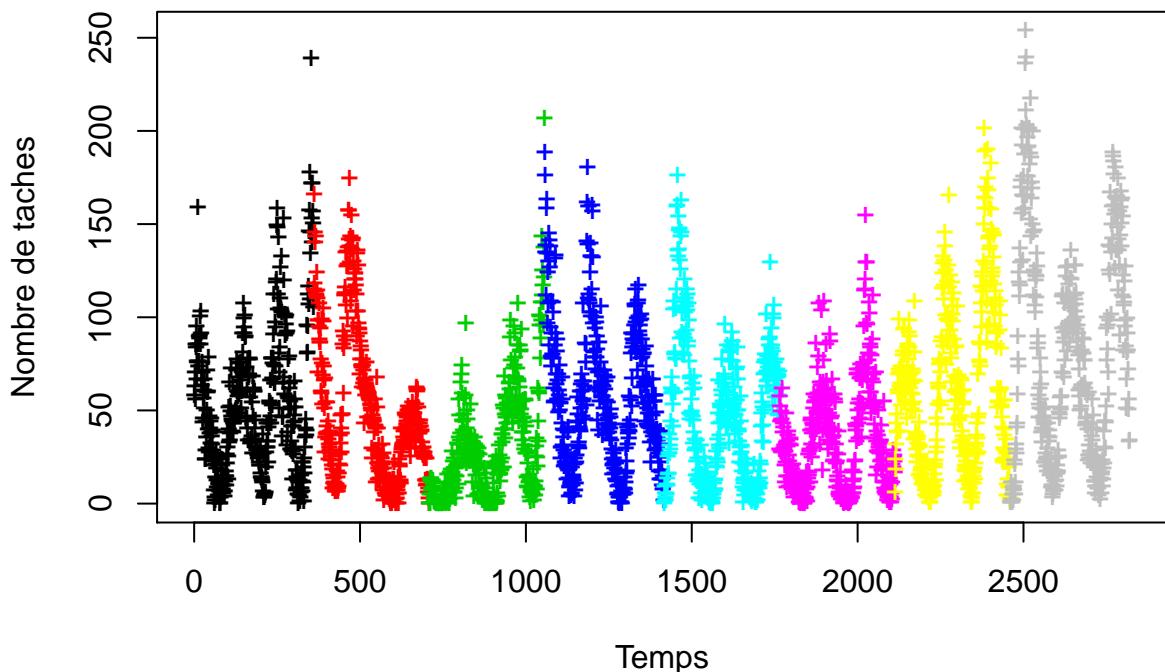
```
> levels(periode) <- couleurs
```

4. Expliquer la sortie de la fonction

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

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

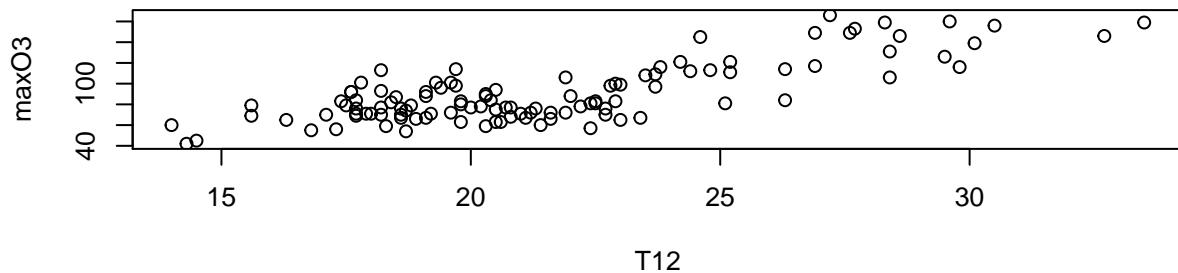
```
> plot(coordx, taches[,1], xlab="Temps", ylab="Nombre de tâches",
+       col=periode, type="p", pch="+")
```



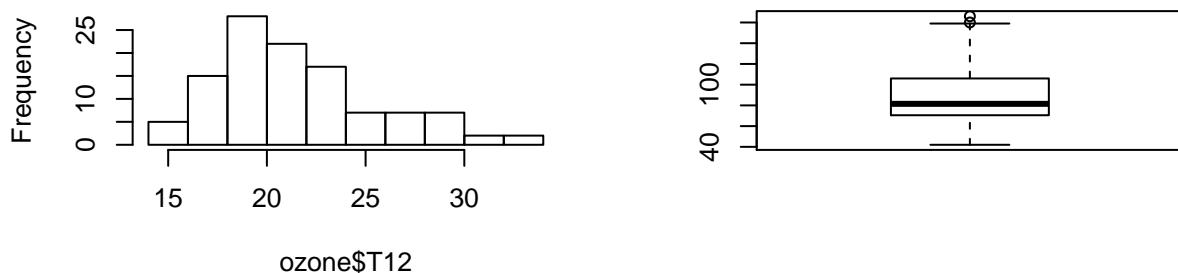
Exercice 1.4 (Layout). On reprend le jeu de données sur l'ozone. A l'aide de la fonction **layout** séparer la fenêtre graphique en deux lignes avec

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**).

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



Histogram of ozone\$T12



1.2 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>.

1.2.1 Premiers graphes ggplot2

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

```
> library(tidyverse)
> set.seed(1234)
> diamonds2 <- diamonds[sample(nrow(diamonds), 5000), ]
> summary(diamonds2)

##      carat          cut       color     clarity      depth
##  Min.   :0.2000   Fair    : 158   D: 640   SI1    :1189   Min.   :43.00
##  1st Qu.:0.4000   Good   : 455   E: 916   VS2    :1157   1st Qu.:61.10
##  Median :0.7000   Very Good:1094   F: 900   SI2    : 876   Median :61.80
##  Mean   :0.7969   Premium :1280   G:1018   VS1    : 738   Mean   :61.76
##  3rd Qu.:1.0400   Ideal   :2013   H: 775   VVS2   : 470   3rd Qu.:62.50
##  Max.   :4.1300                    I: 481   VVS1   : 326   Max.   :71.60
##                               J: 270   (Other): 244
##
##      table          price         x           y
##  Min.   :49.00   Min.   : 365   Min.   : 0.000   Min.   :3.720
##  1st Qu.:56.00   1st Qu.: 945   1st Qu.: 4.720   1st Qu.:4.720
##  Median :57.00   Median :2376   Median : 5.690   Median :5.700
##  Mean   :57.43   Mean   :3917   Mean   : 5.728   Mean   :5.731
##  3rd Qu.:59.00   3rd Qu.:5294   3rd Qu.: 6.530   3rd Qu.:6.520
##  Max.   :95.00   Max.   :18757  Max.   :10.000  Max.   :9.850
##
##      z
##  Min.   :0.000
##  1st Qu.:2.920
##  Median :3.520
##  Mean   :3.538
##  3rd Qu.:4.030
##  Max.   :6.430
##
> help(diamonds)
```

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

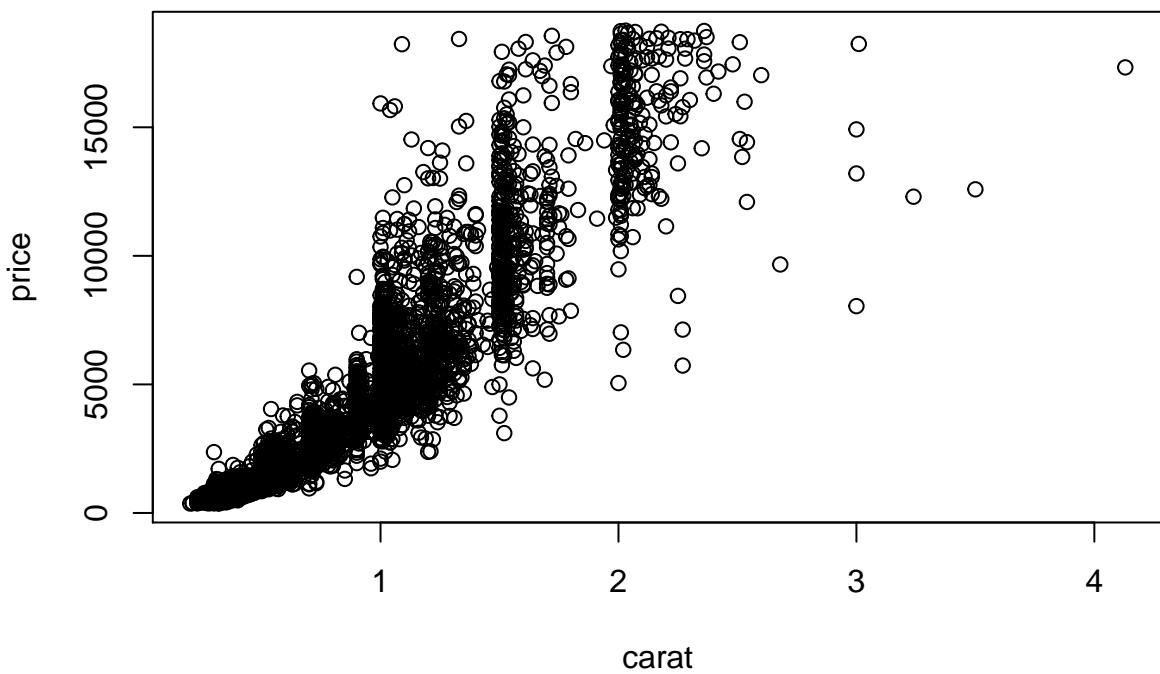
- les données
- les variables que l'on souhaite représenter
- le type de représentation (nuage de points, boxplot...).

Il existe un verbe pour définir chacune de ces couches :

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

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

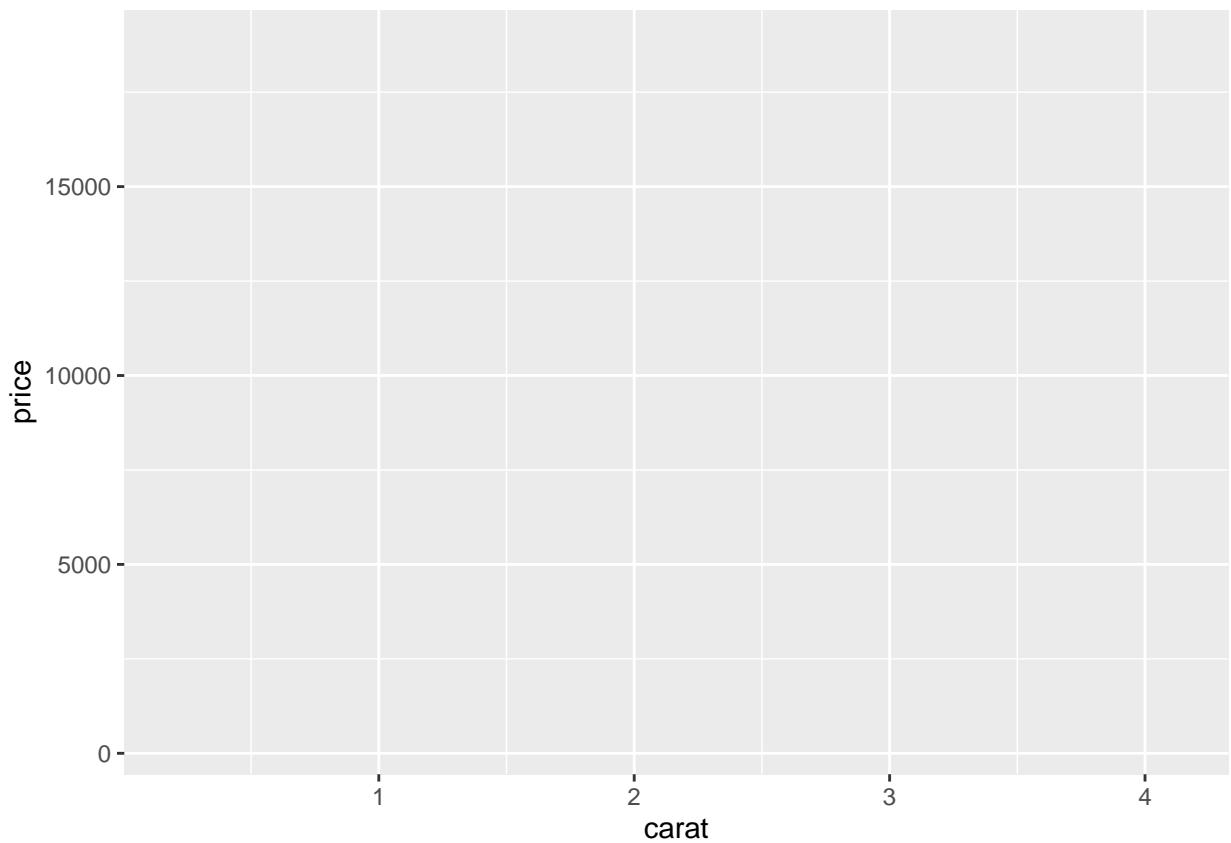
```
> plot(price~carat,data=diamonds2)
```



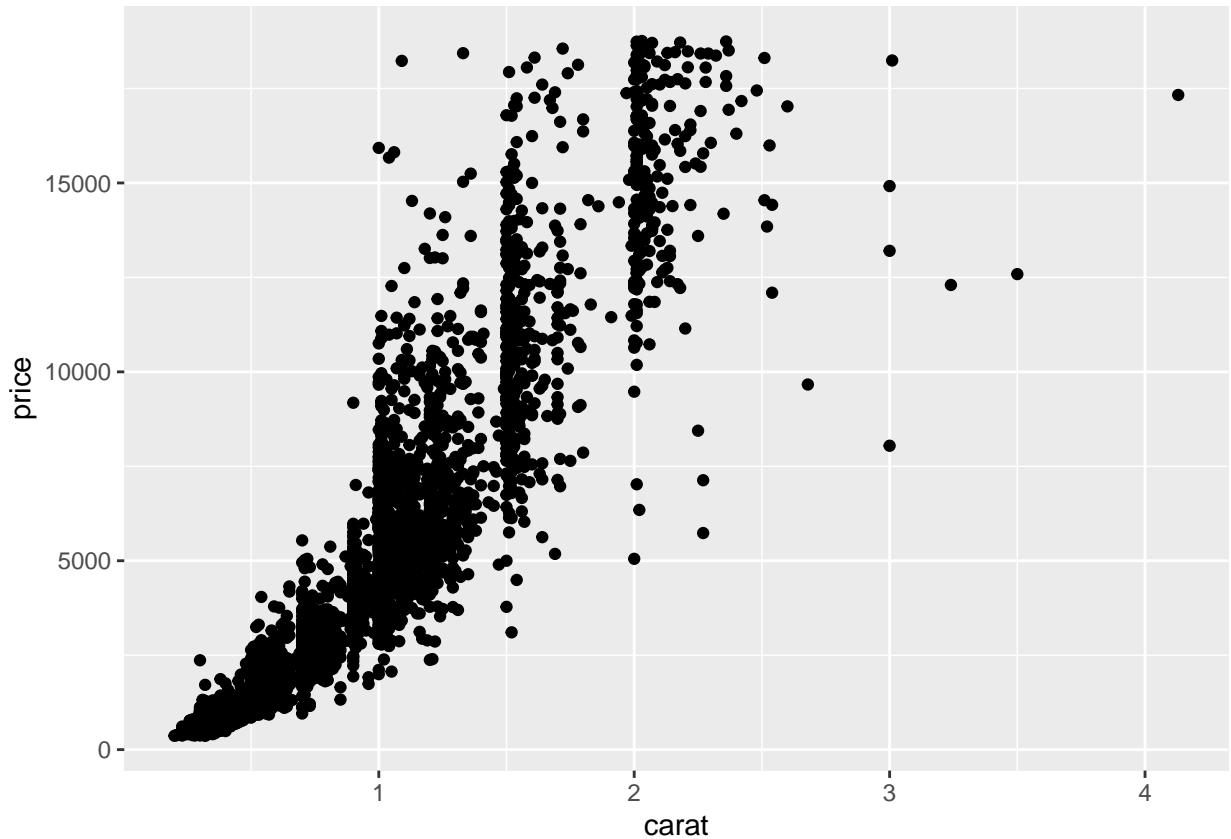
Avec **ggplot**, on va faire

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

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



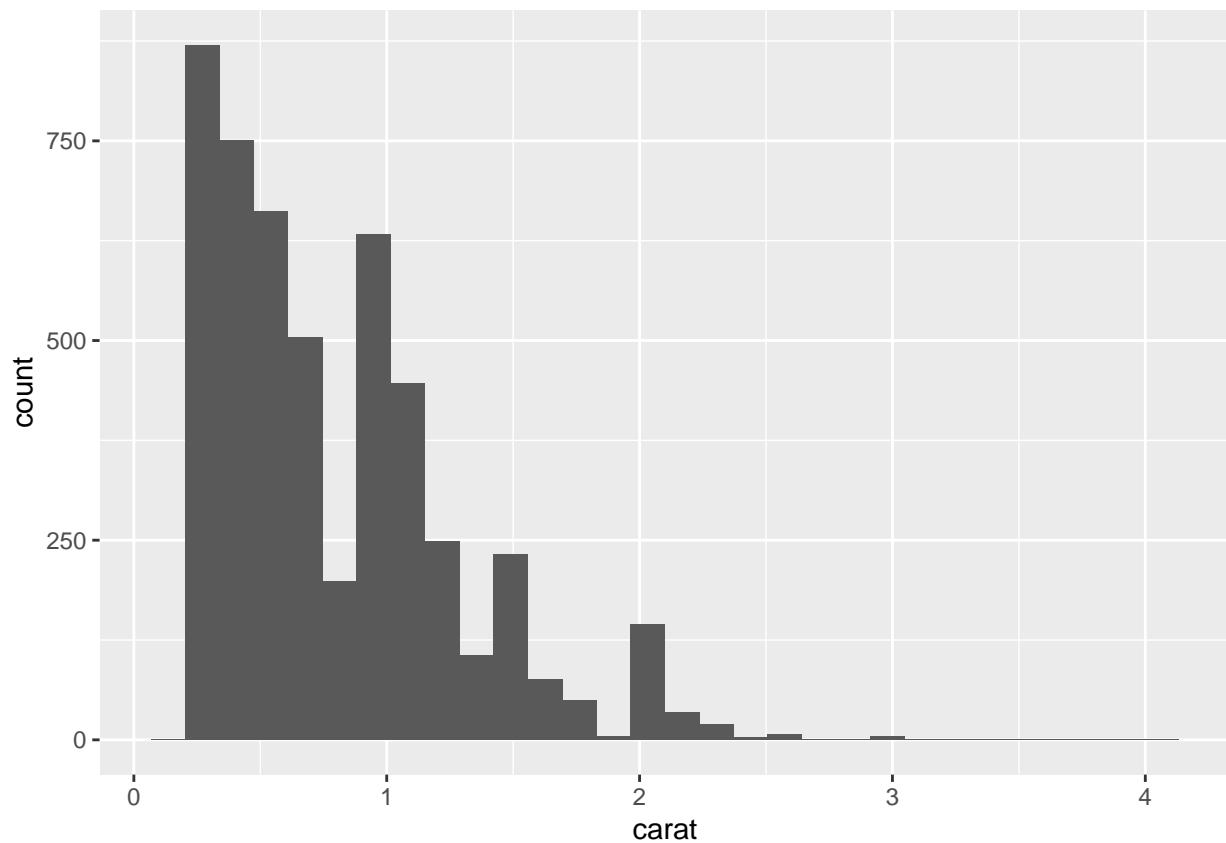
```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point() #bon
```



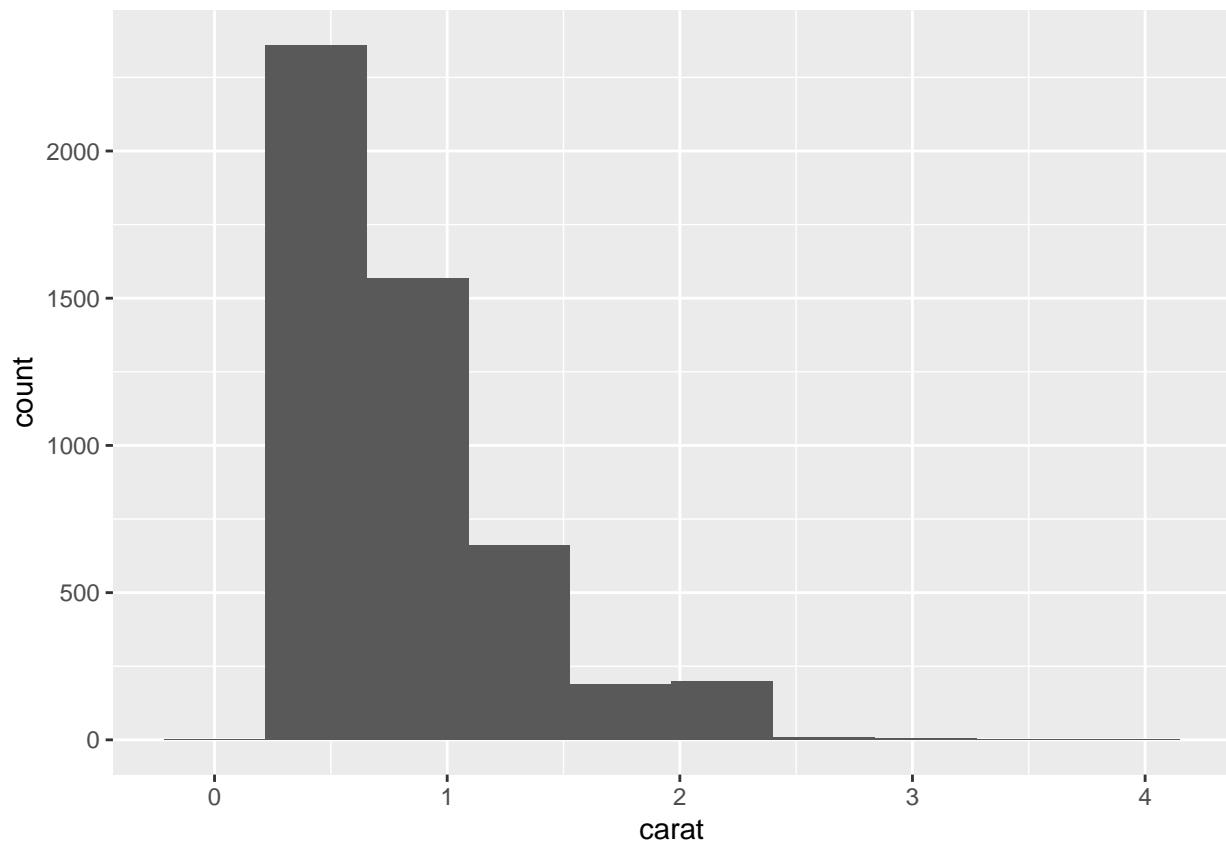
Exercice 1.5 (Premiers graphes ggplot).

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

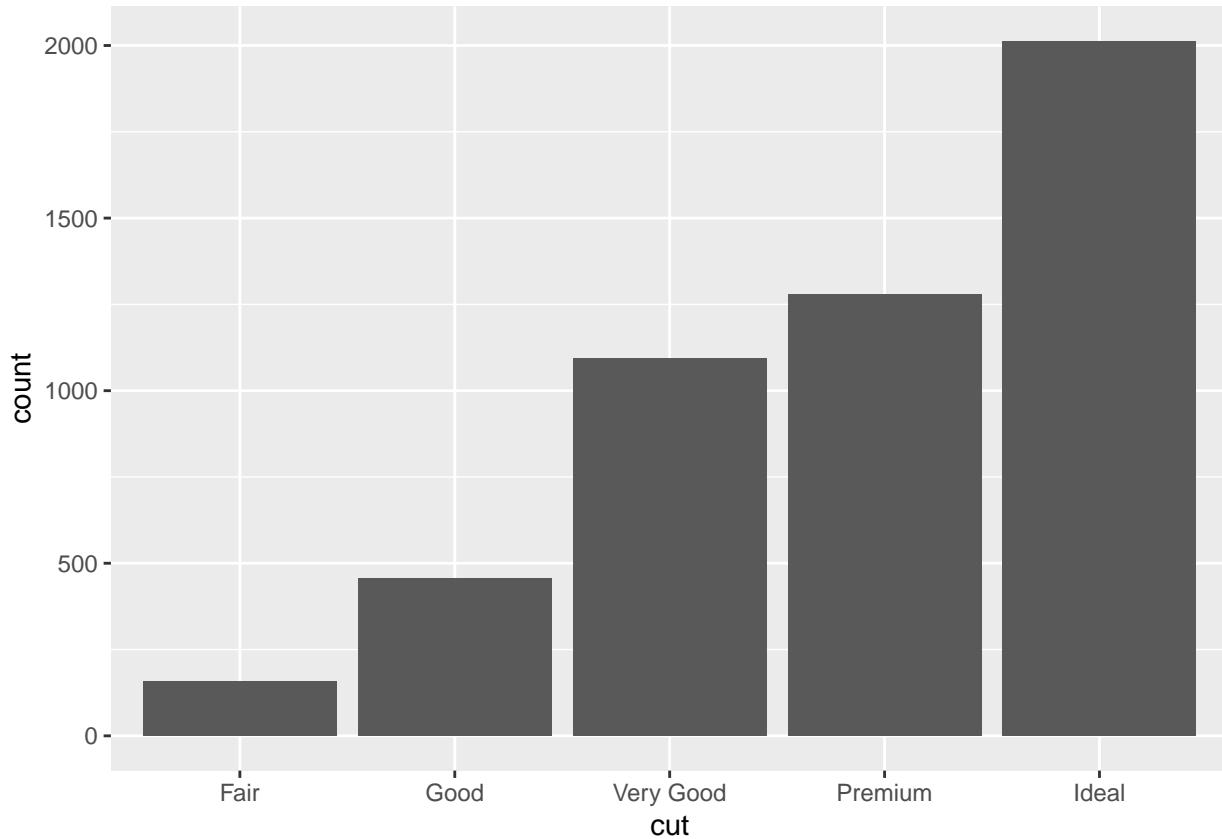
```
> ggplot(diamonds2)+aes(x=carat)+geom_histogram()
```



```
> ggplot(diamonds2)+aes(x=carat)+geom_histogram(bins=10)
```



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



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

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

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

1.2.2 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)
```

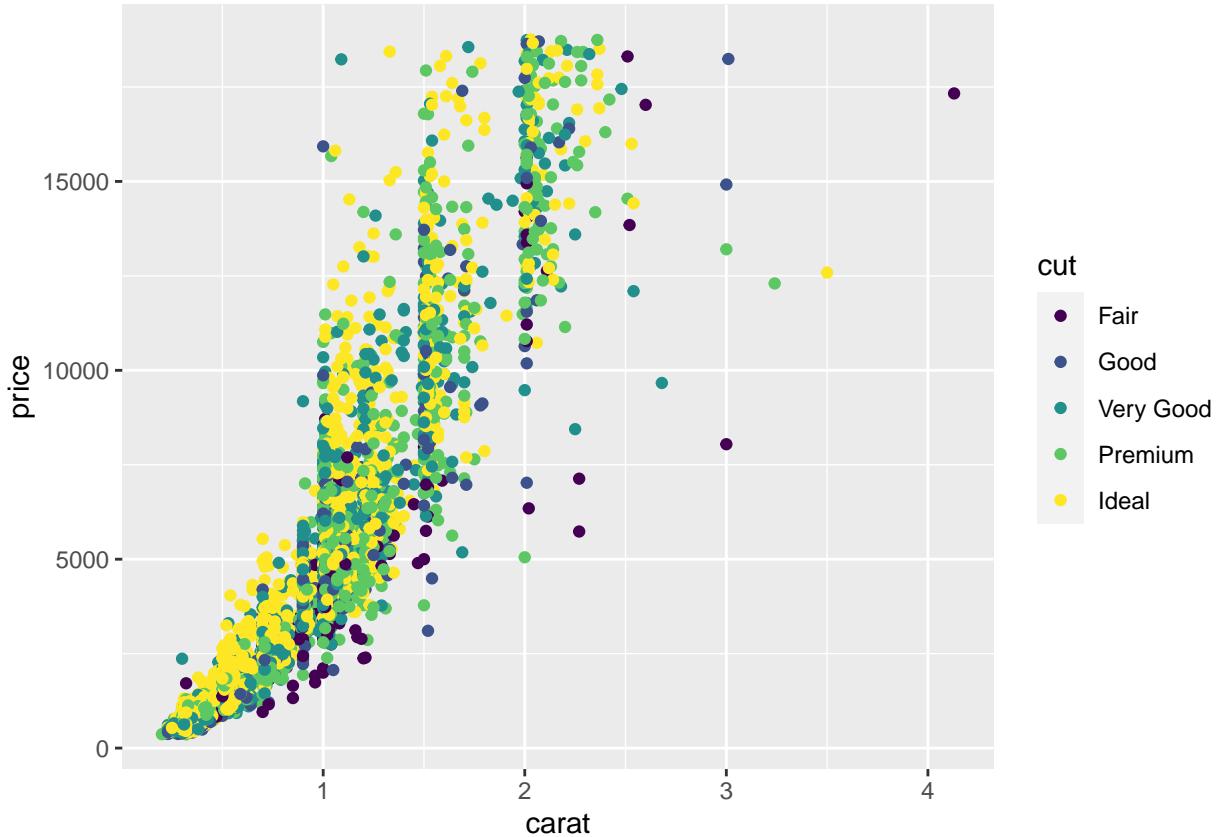
Les variables peuvent également être utilisées pour colorier des points ou des barres, définir des tailles... Dans ce cas on pourra renseigner les arguments **color**, **size**, **fill** dans la fonction **aes**. Par exemple

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

1.2.3 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` :

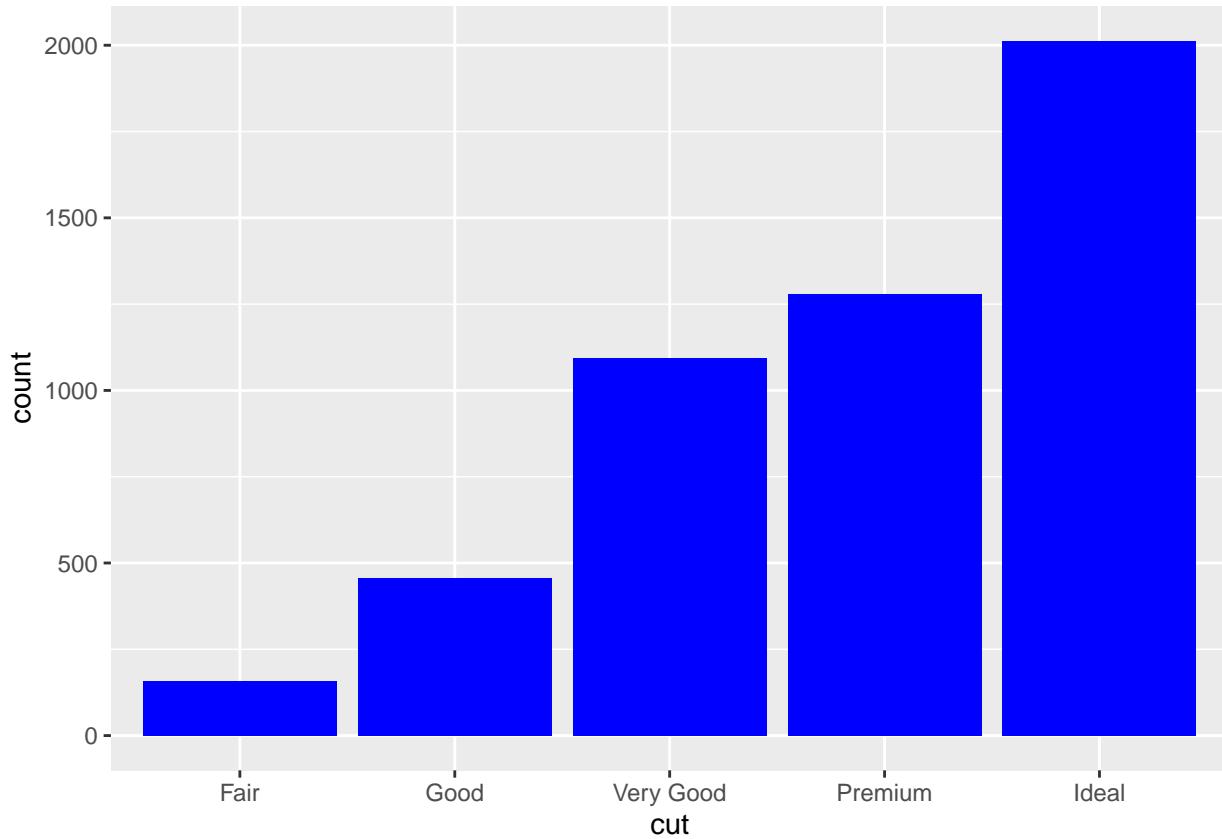
Table 1: Principaux geometrics

Geom	Description	Aesthetics
<code>geom_point()</code>	nuage de points	x, y, shape, fill
<code>geom_line()</code>	Ligne (ordonnée selon x)	x, y, linetype
<code>geom_abline()</code>	Ligne	slope, intercept
<code>geom_path()</code>	Ligne (ordonnée par l'index)	x, y, linetype
<code>geom_text()</code>	Texte	x, y, label, hjust, vjust
<code>geom_rect()</code>	Rectangle	xmin, xmax, ymin, ymax, fill, linetype
<code>geom_polygon()</code>	Polygone	x, y, fill, linetype
<code>geom_segment()</code>	Segment	x, y, xend, yend, fill, linetype
<code>geom_bar()</code>	Diagramme en barres	x, fill, linetype, weight
<code>geom_histogram()</code>	Histogramme	x, fill, linetype, weight
<code>geom_boxplot()</code>	Boxplot	x, fill, weight
<code>geom_density()</code>	Densité	x, y, fill, linetype
<code>geom_contour()</code>	Lignes de contour	x, y, fill, linetype
<code>geom_smooth()</code>	Lisseur (linéaire ou non linéaire)	x, y, fill, linetype
Tous		color, size, group

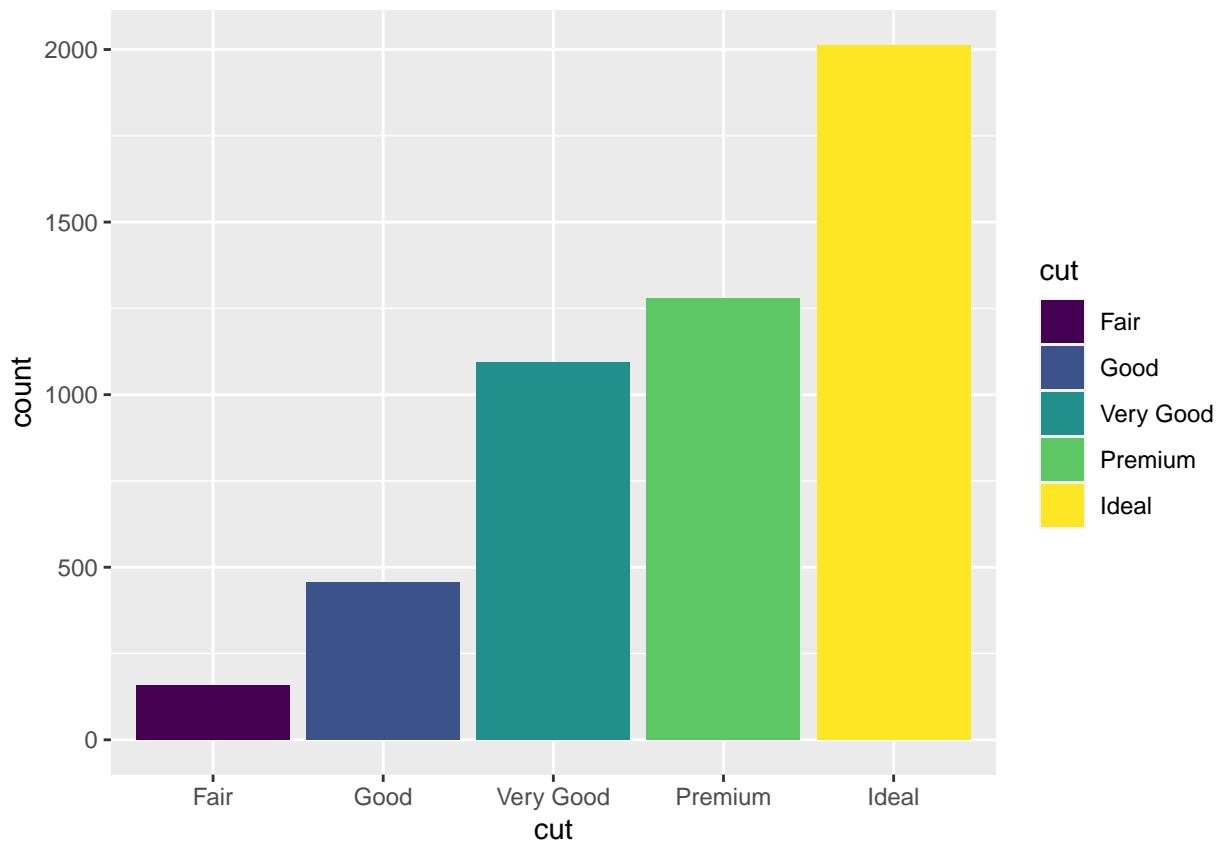
Exercice 1.6 (Diagrammes en barres).

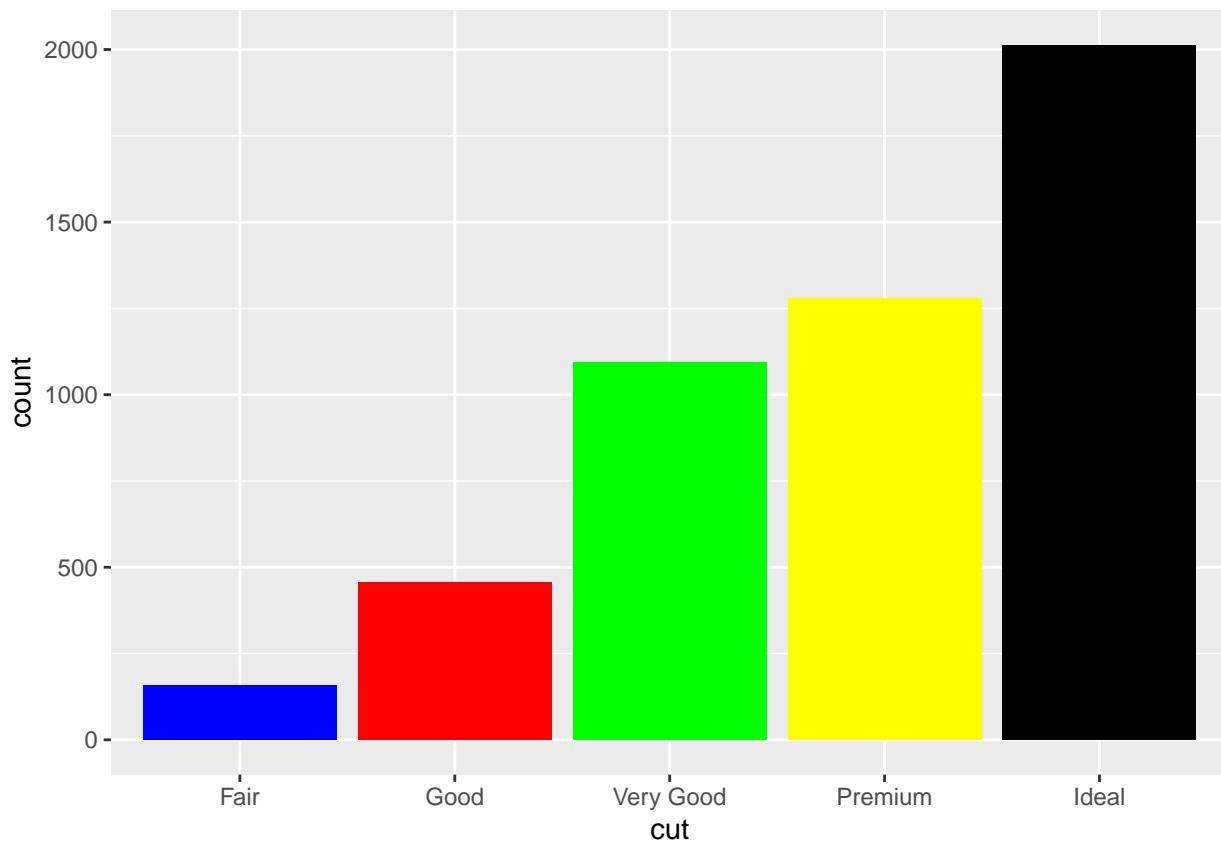
1. Tracer le diagramme en barres de la variable **cut** avec des barres bleues.
2. Tracer le diagramme en barres de la variable **cut** avec une couleur pour chaque modalité de **cut** ainsi qu'une légende qui permet de repérer la couleur.
3. Tracer le diagramme en barres de la variable **cut** avec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

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



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

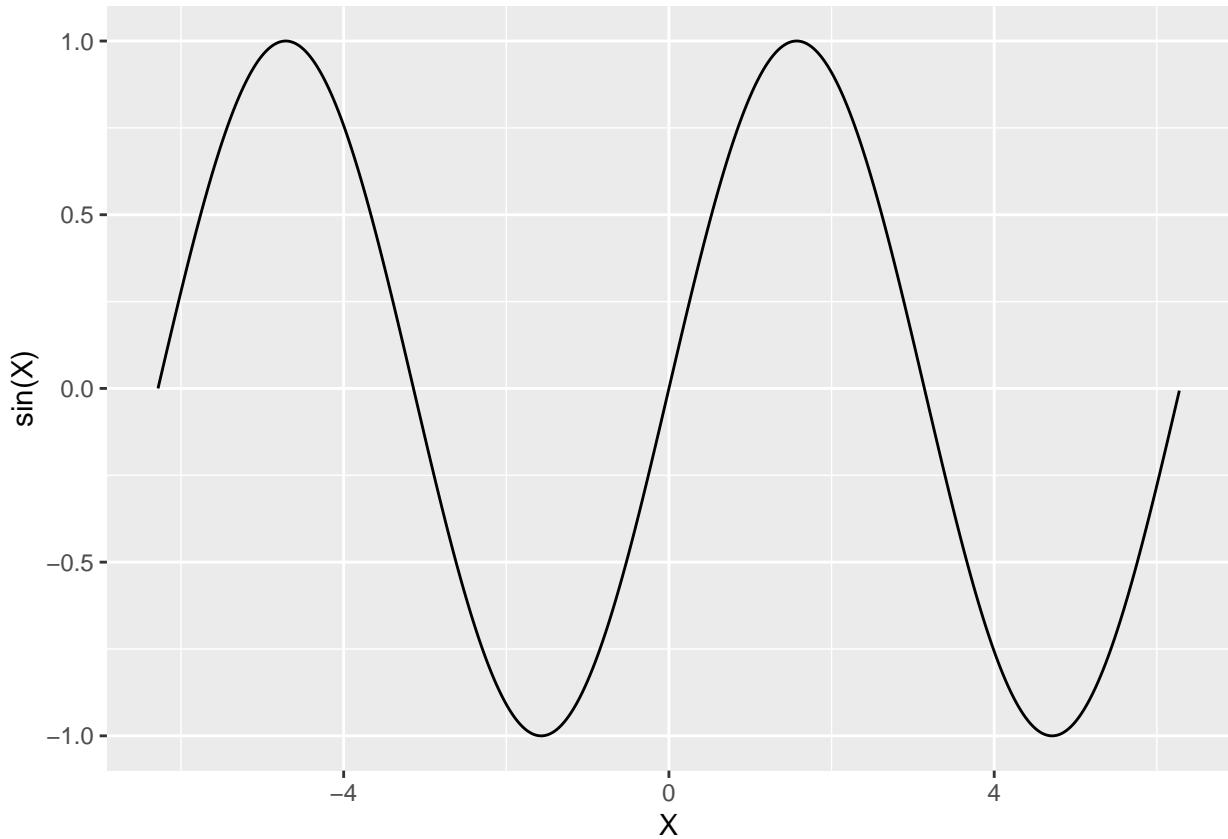




1.2.4 Statistics

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()
```

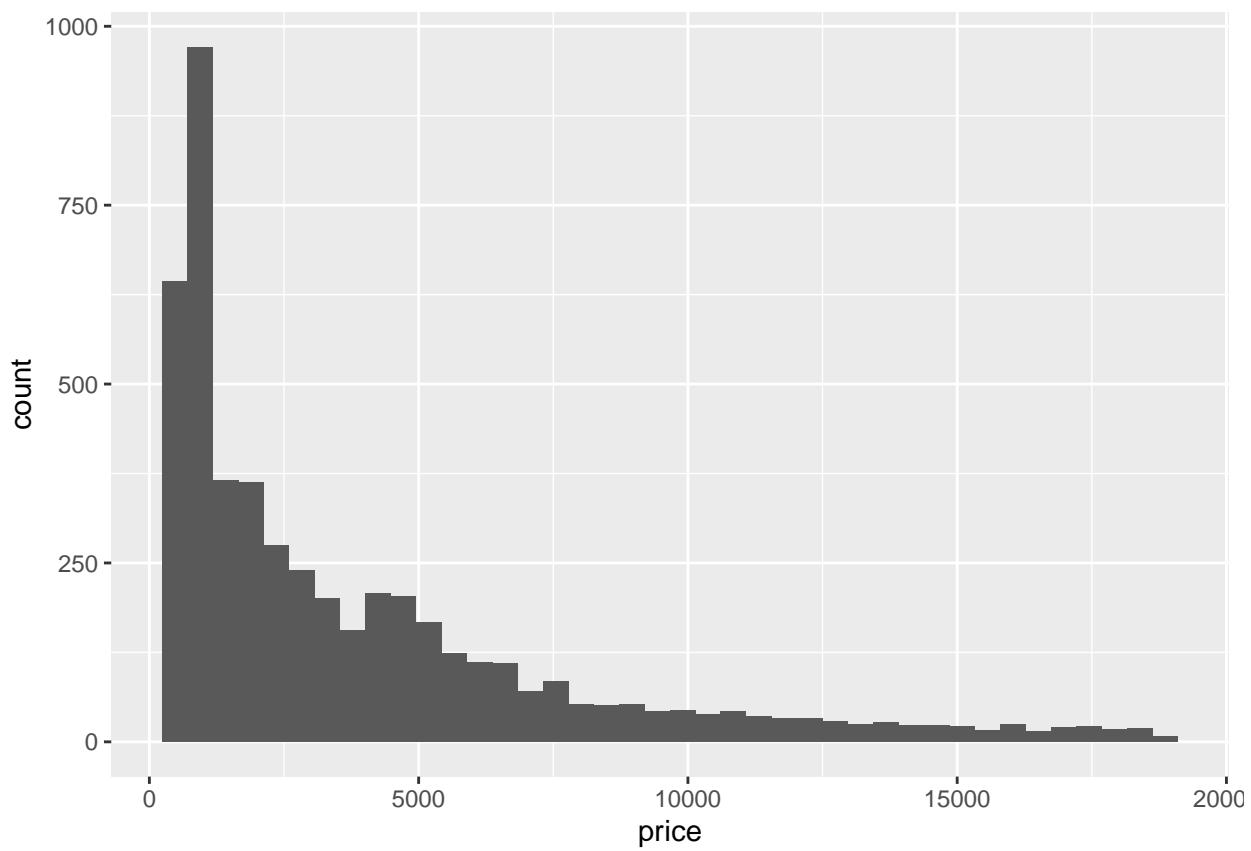


La transformation est spécifiée dans la fonction **aes**. Pour des transformations plus complexes, nous devons utiliser des **statistics**. Une fonction **stat** permet de définir des nouvelles variables à partir du jeu de données initial, il est ensuite possible de représenter ces nouvelles variables.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

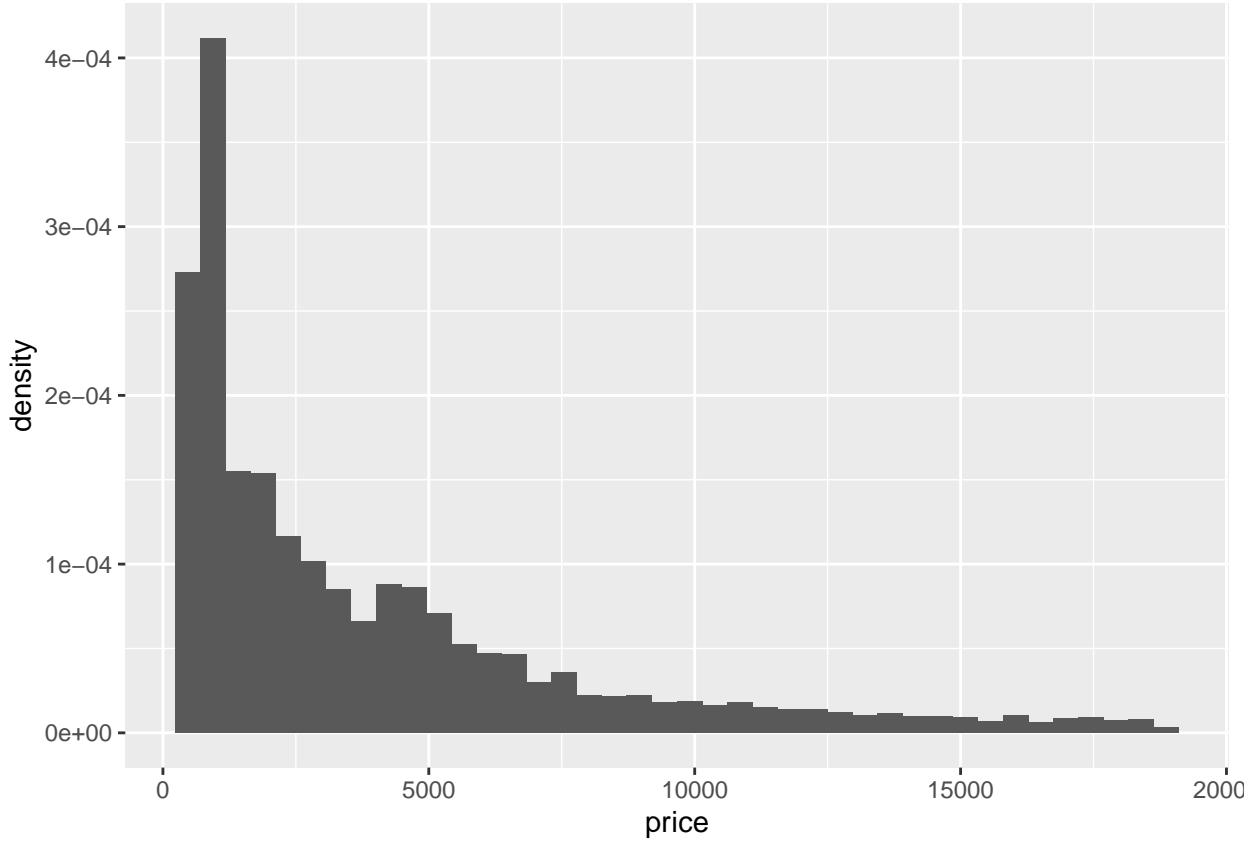
Par défaut *geom_histogram* fait appel à cette fonction **stat_bin** et représente sur l'axe *y* le nombre d'observations dans chaque classe (la variable **count**).

```
> 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 2: 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 1.7 (Diagramme en barres "très simple"....).

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

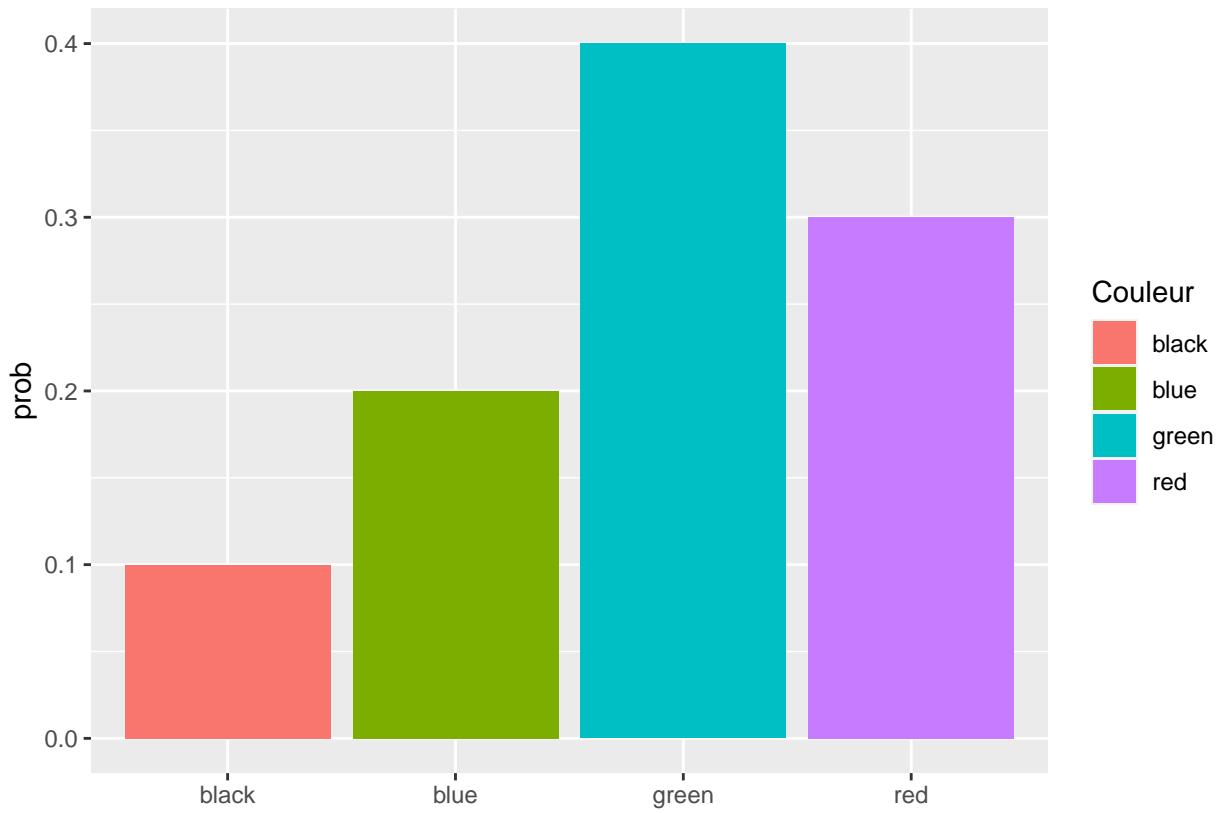
$$P(X = red) = 0.3, \quad P(X = blue) = 0.2, \quad P(X = green) = 0.4, \quad P(X = black) = 0.1$$

Représenter cette distribution de probabilité avec un diagramme en barres.

```

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

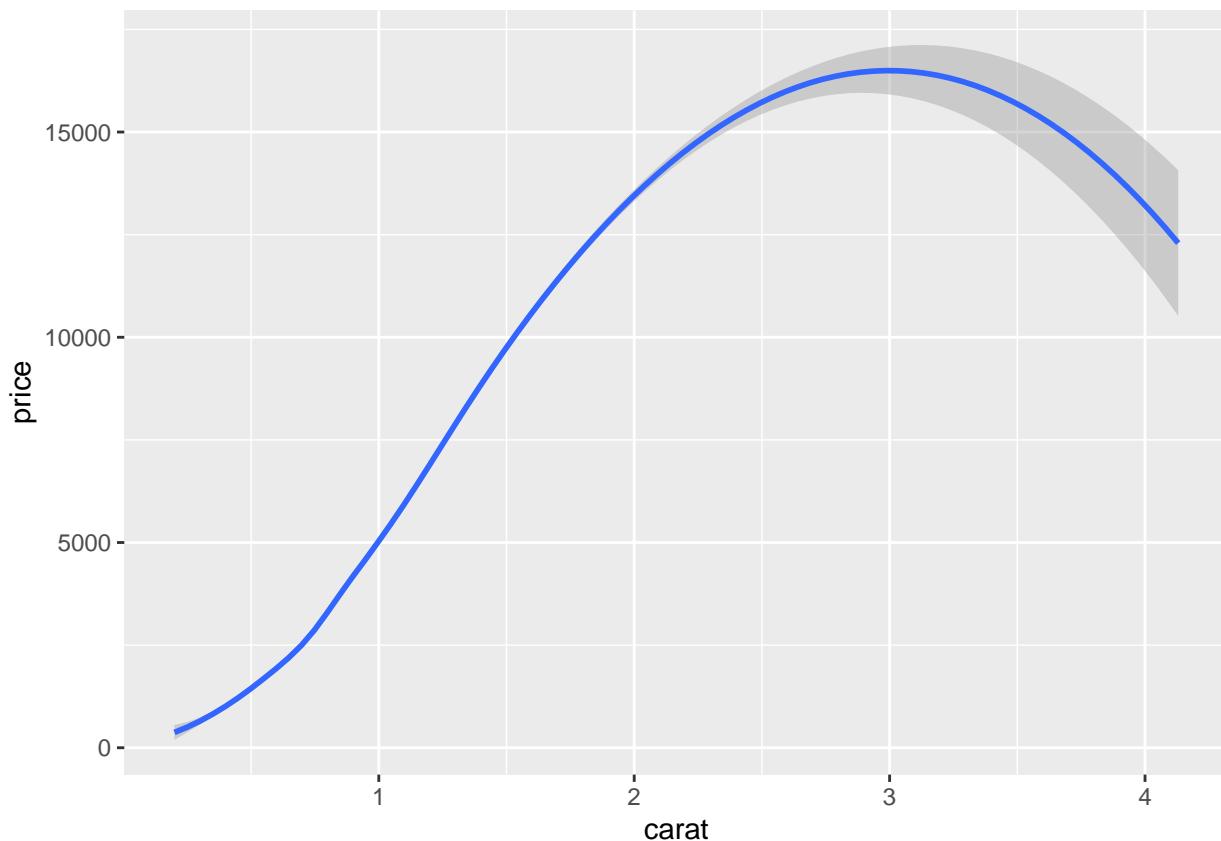
```

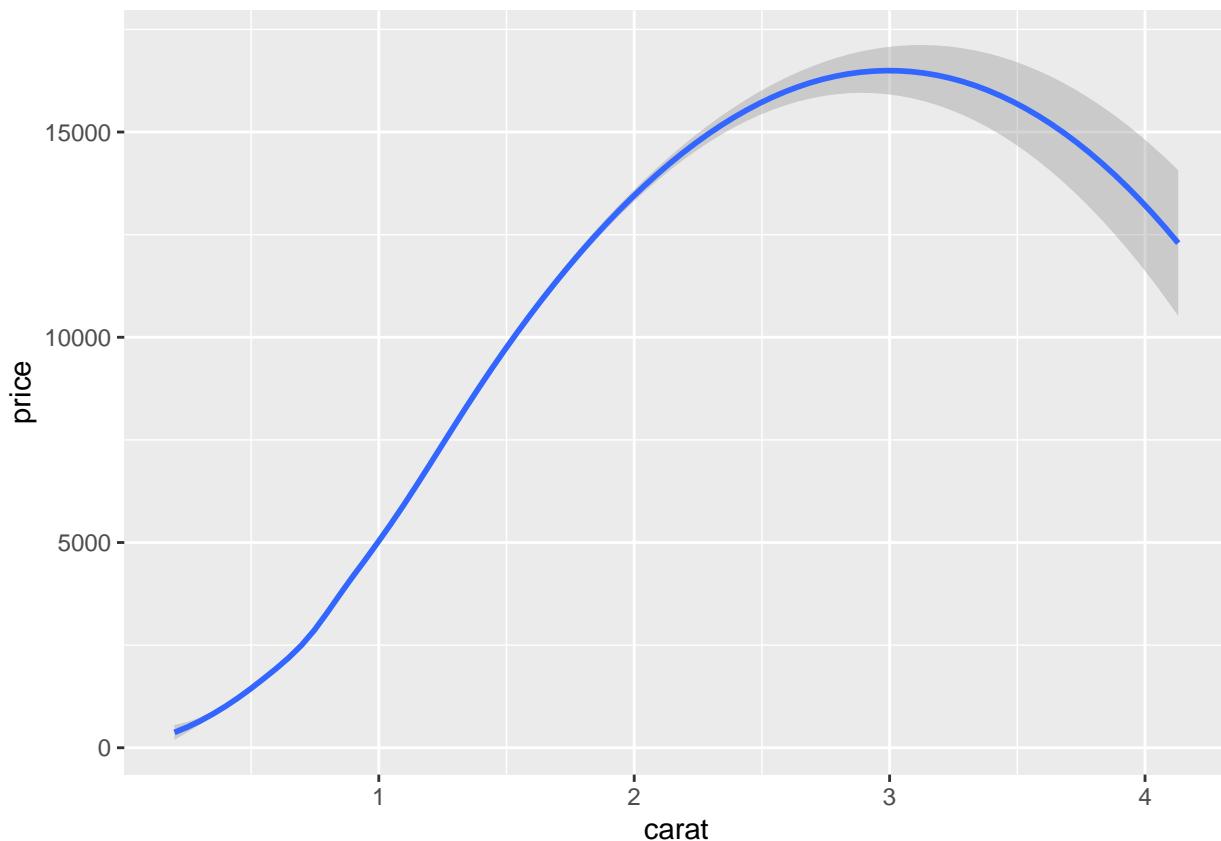


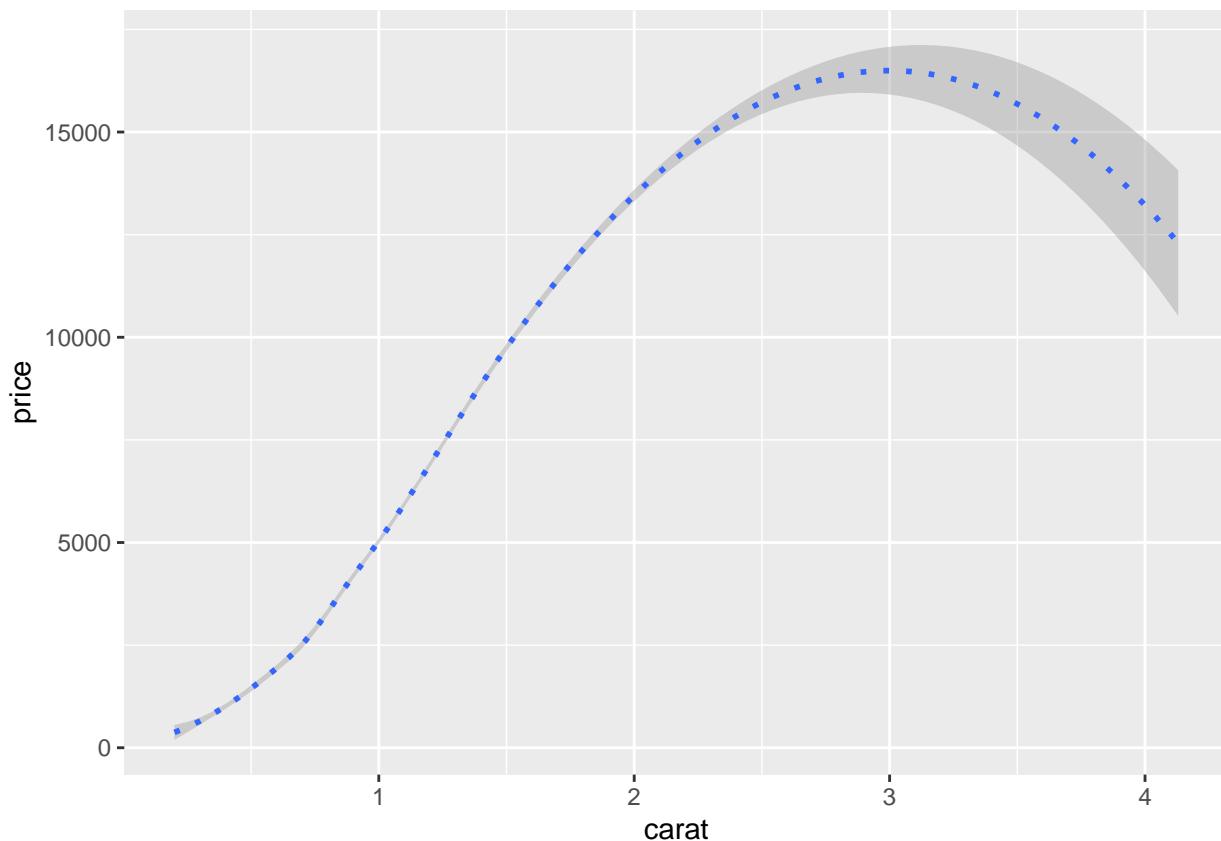
Exercice 1.8 (Lissage).

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.

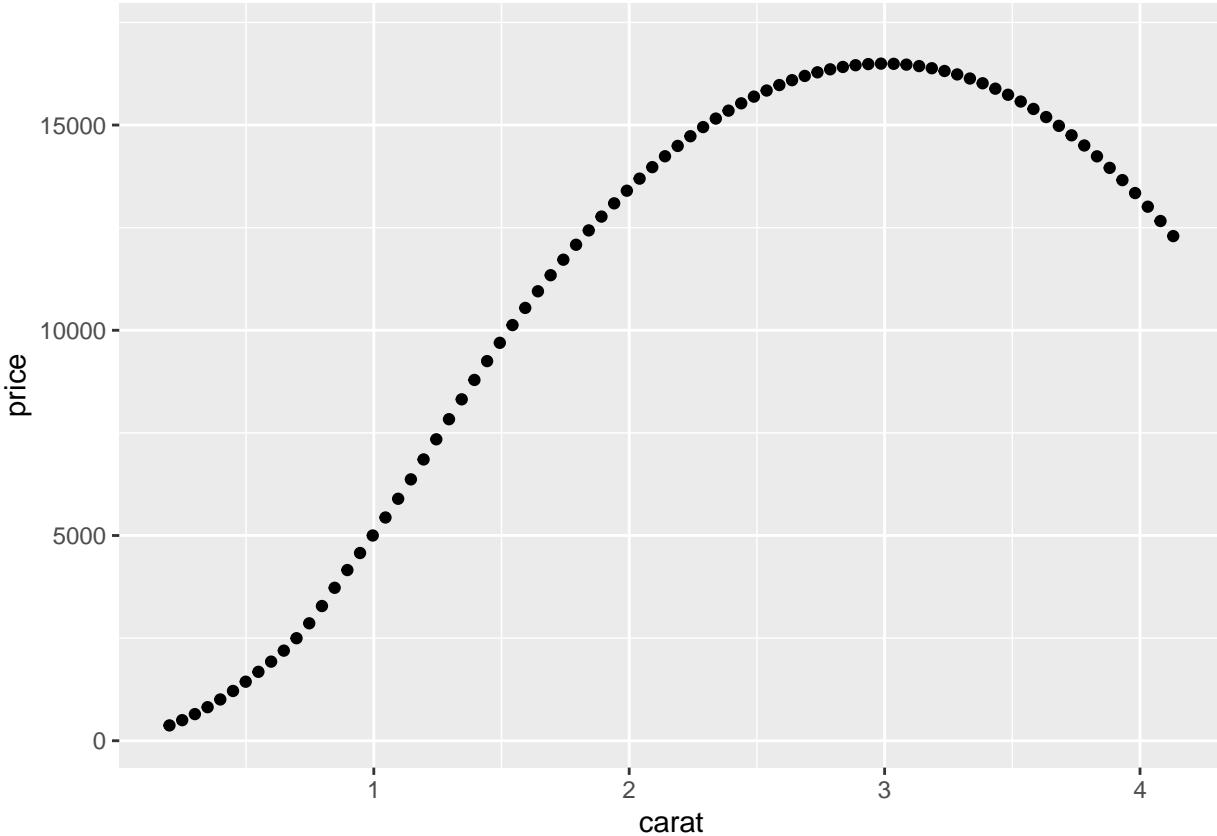
```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_smooth(method="loess")
```







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



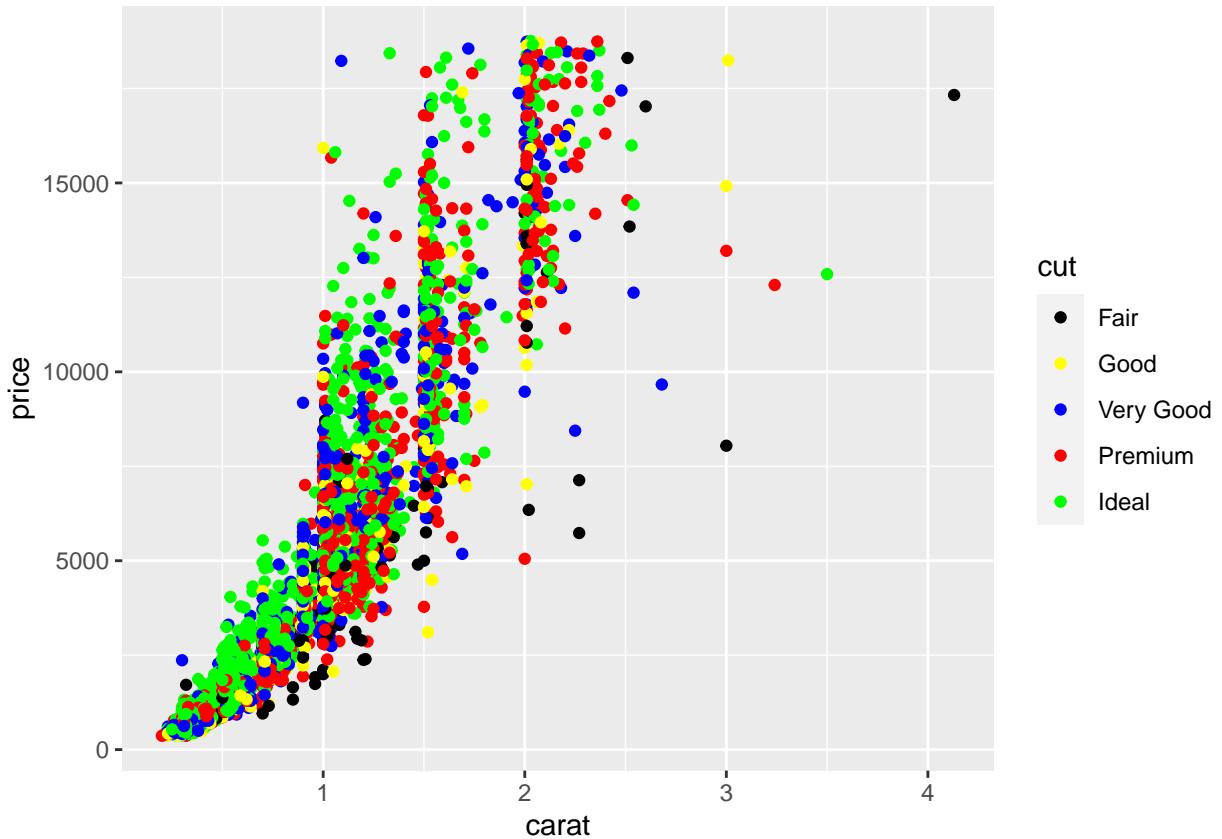
1.2.5 Scales

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 :

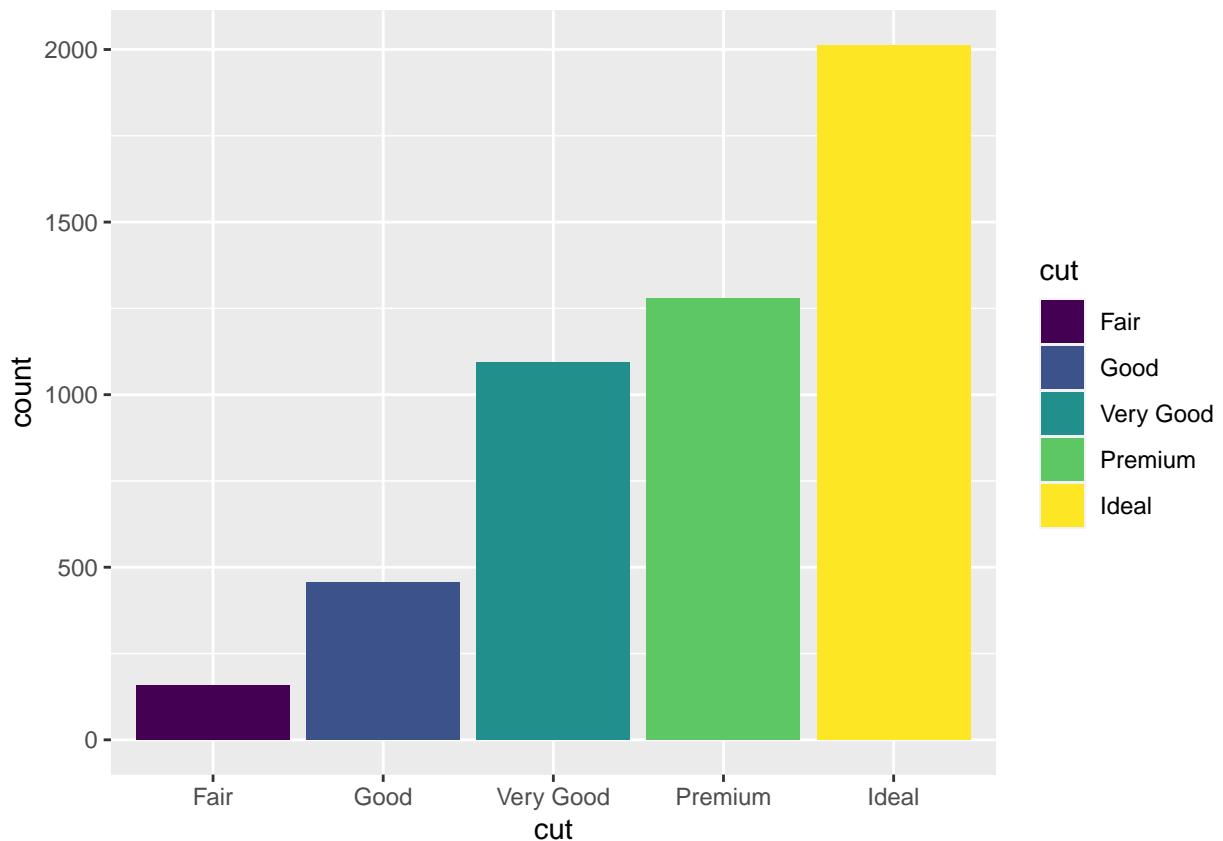
Table 3: Exemples d'échelles

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

Nous présentons quelques exemples d'utilisation des échelles :

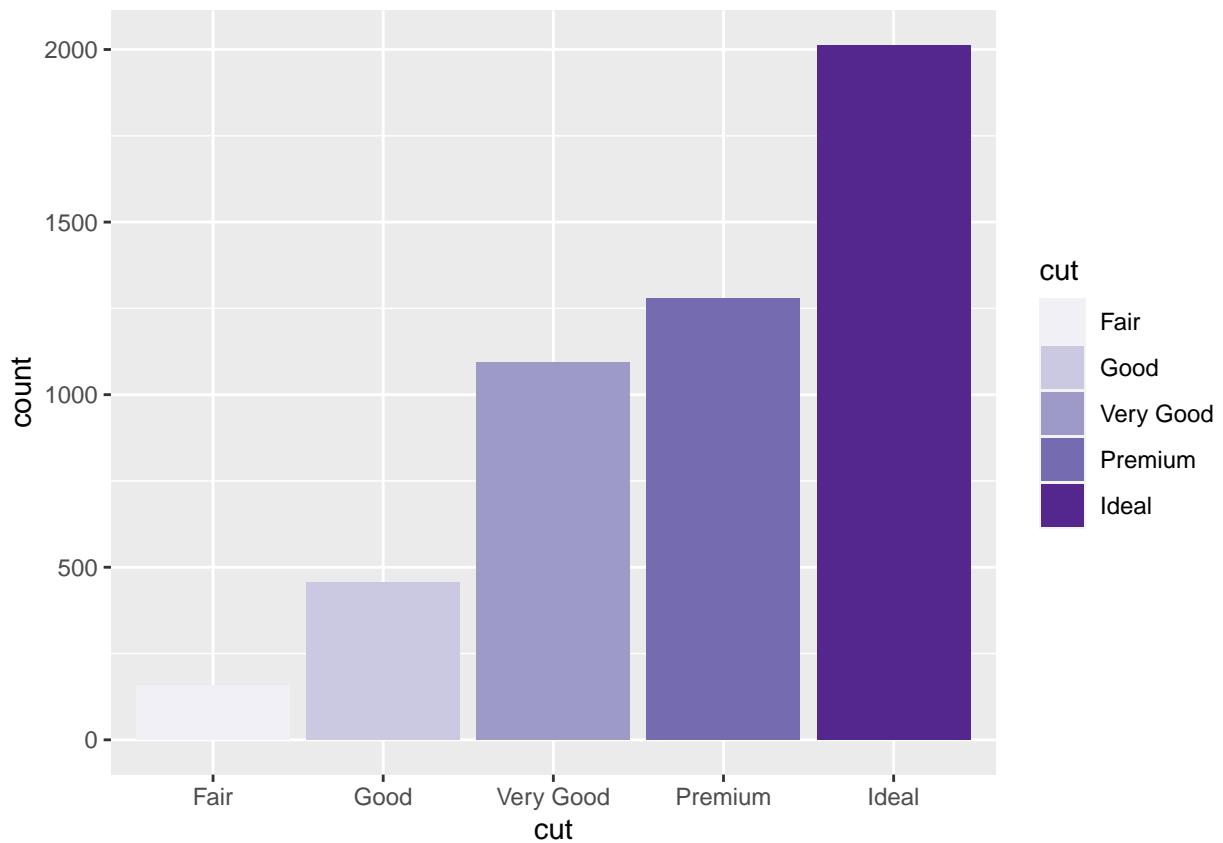
- Couleur dans un diagramme en barres

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



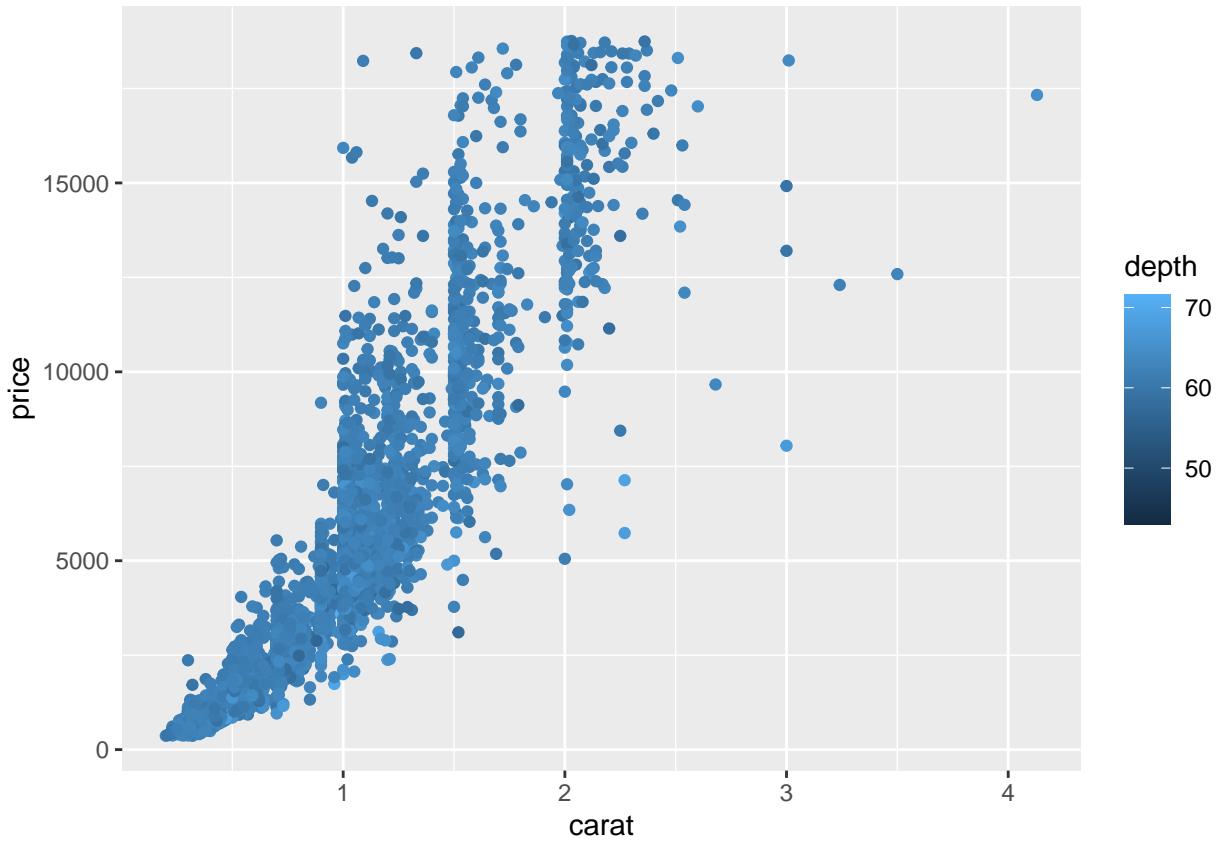
On change la couleur en utilisant la palette **Purples** :

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



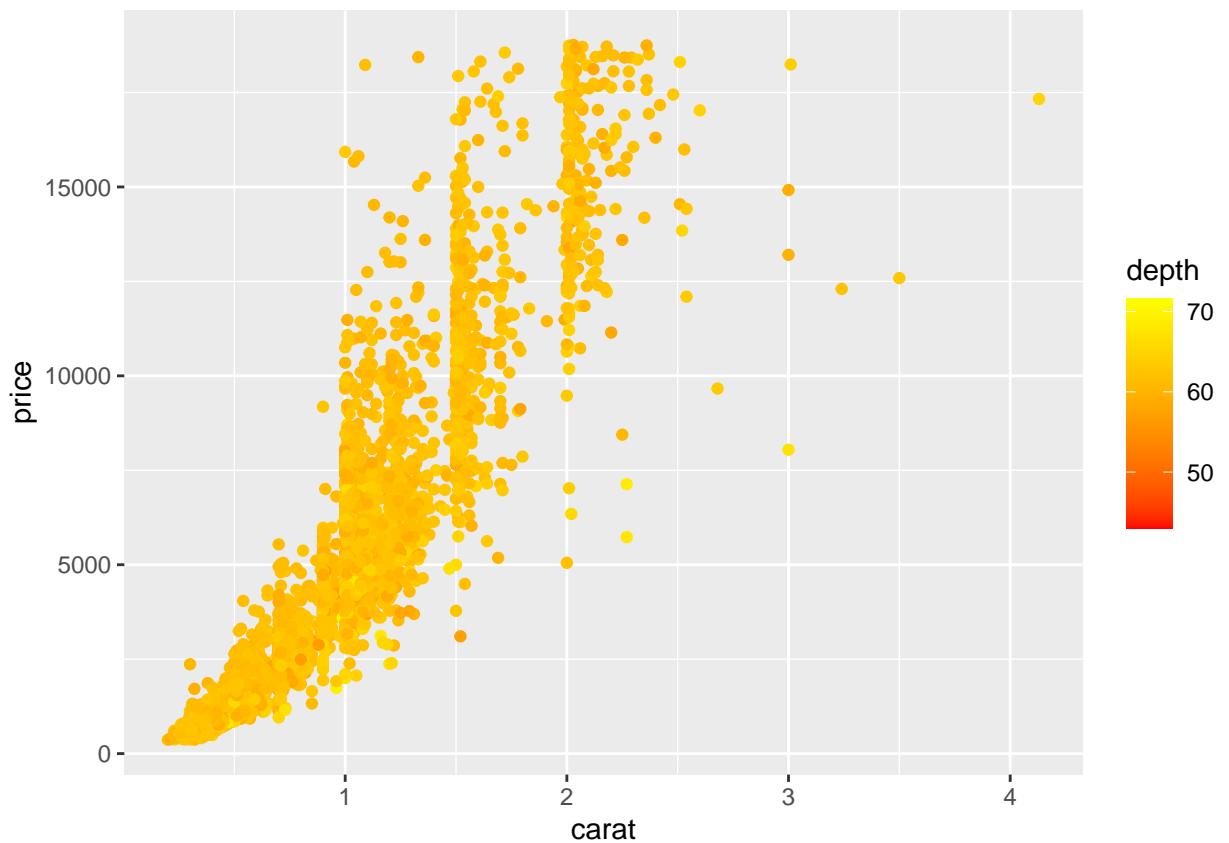
- Gradient de couleurs pour un nuage de points :

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



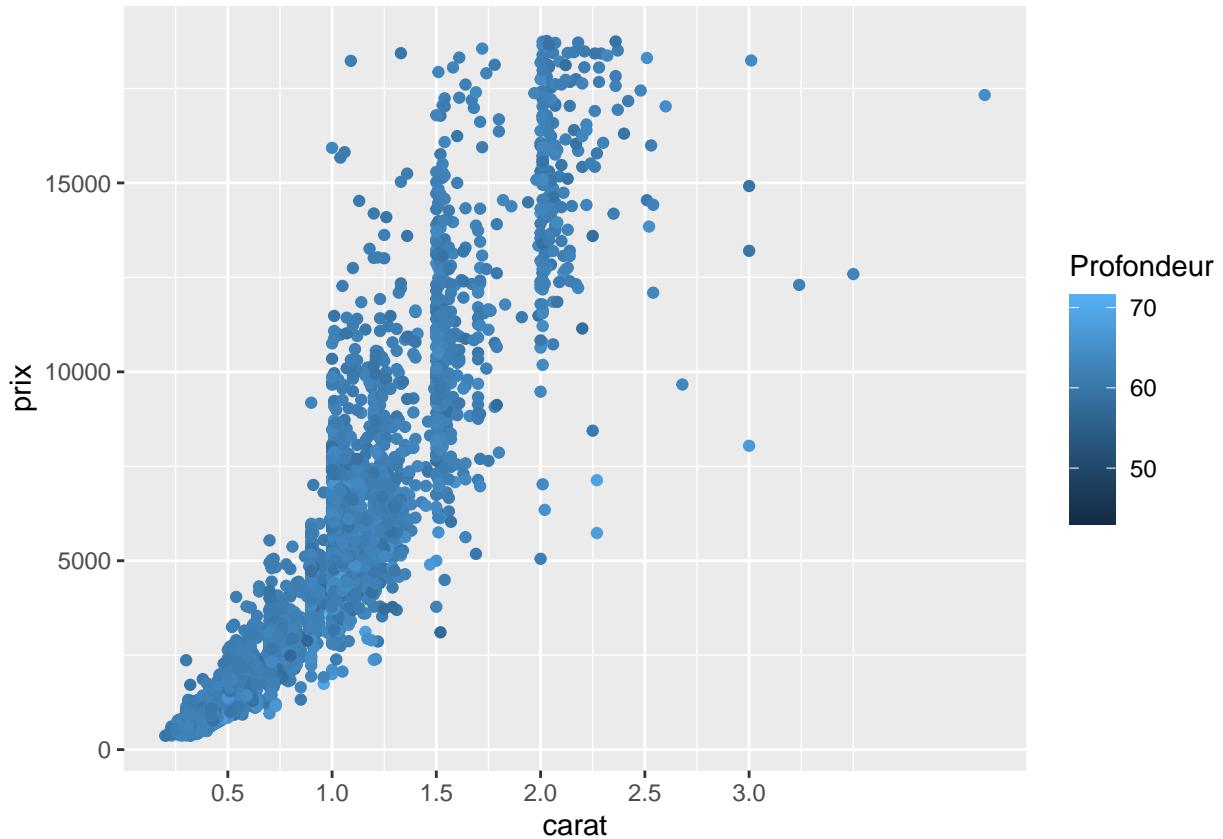
On change le gradient de couleur

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



- Modification sur les axes

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



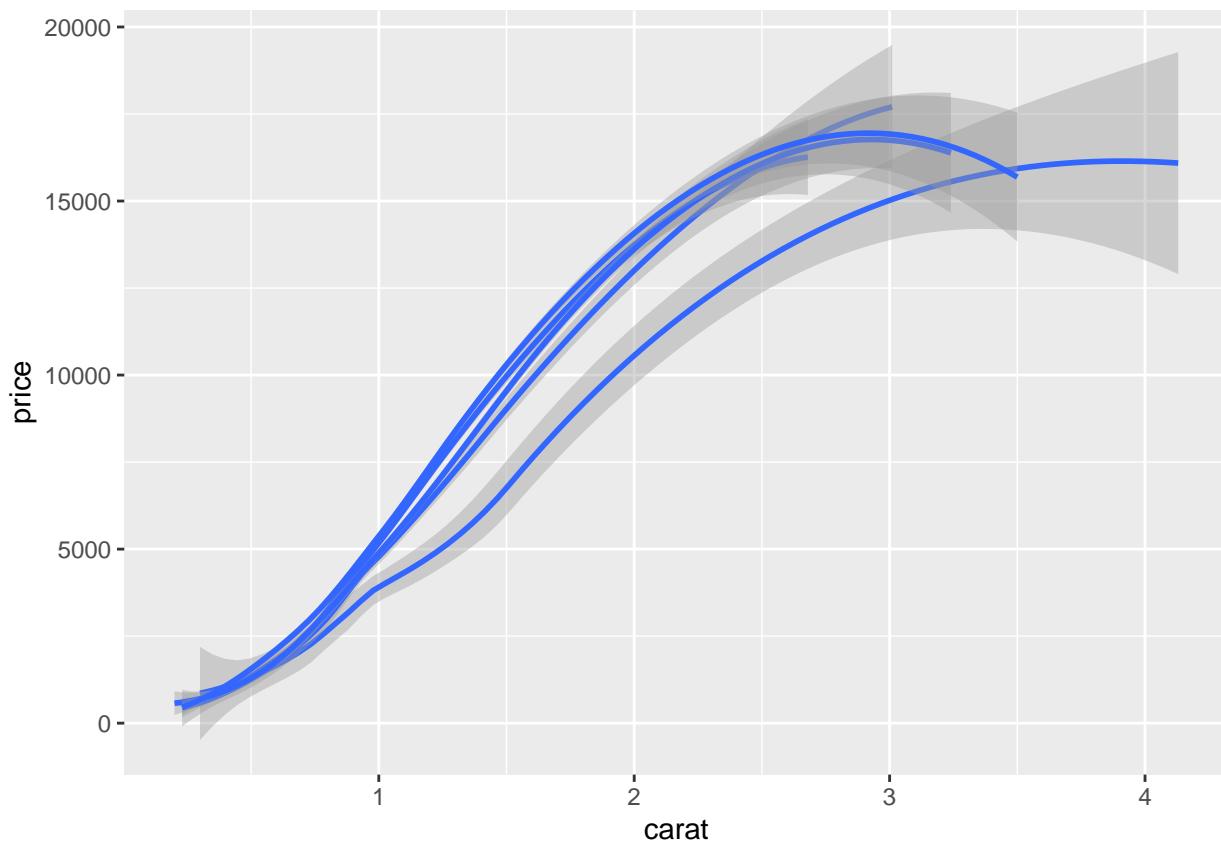
1.2.6 Group et facets

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**.

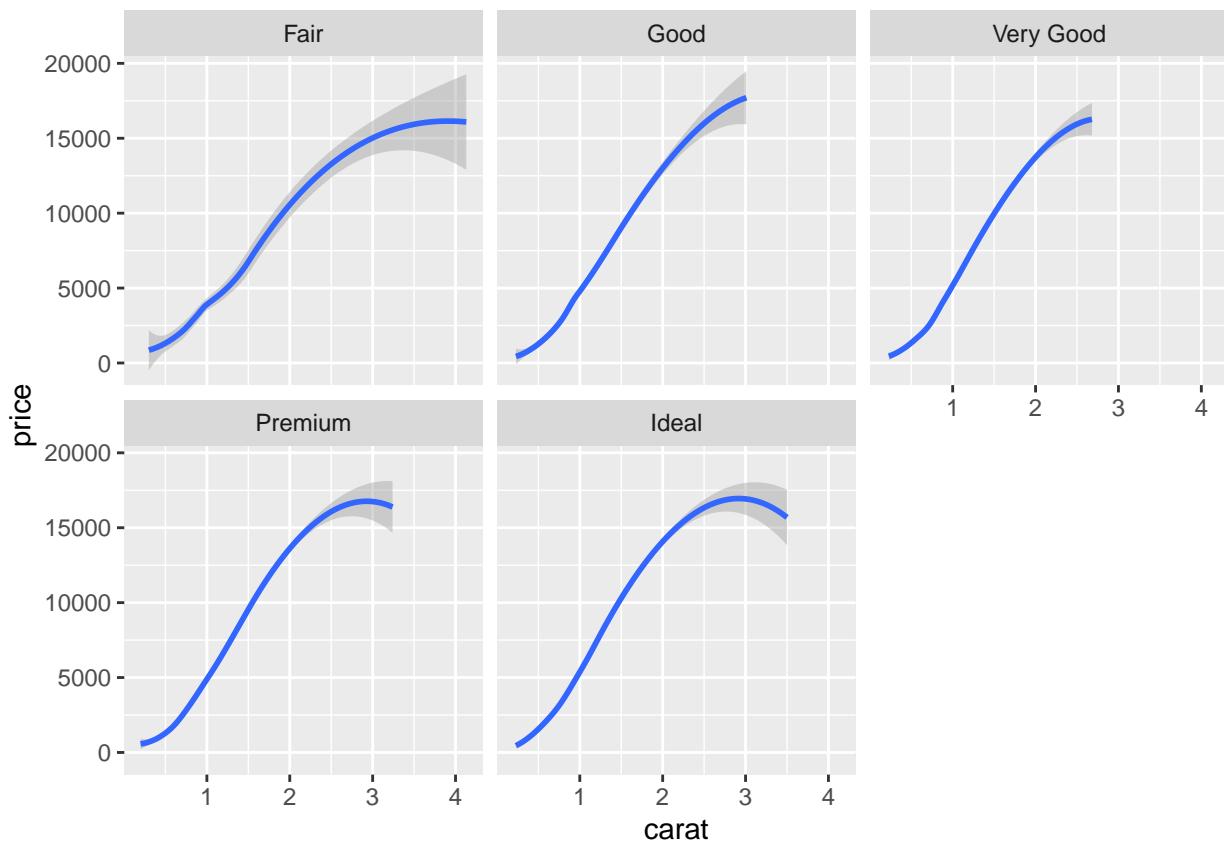
Représentons ici (sur le même graphe) le lisser **price vs carat** pour chaque modalité de *cut*

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

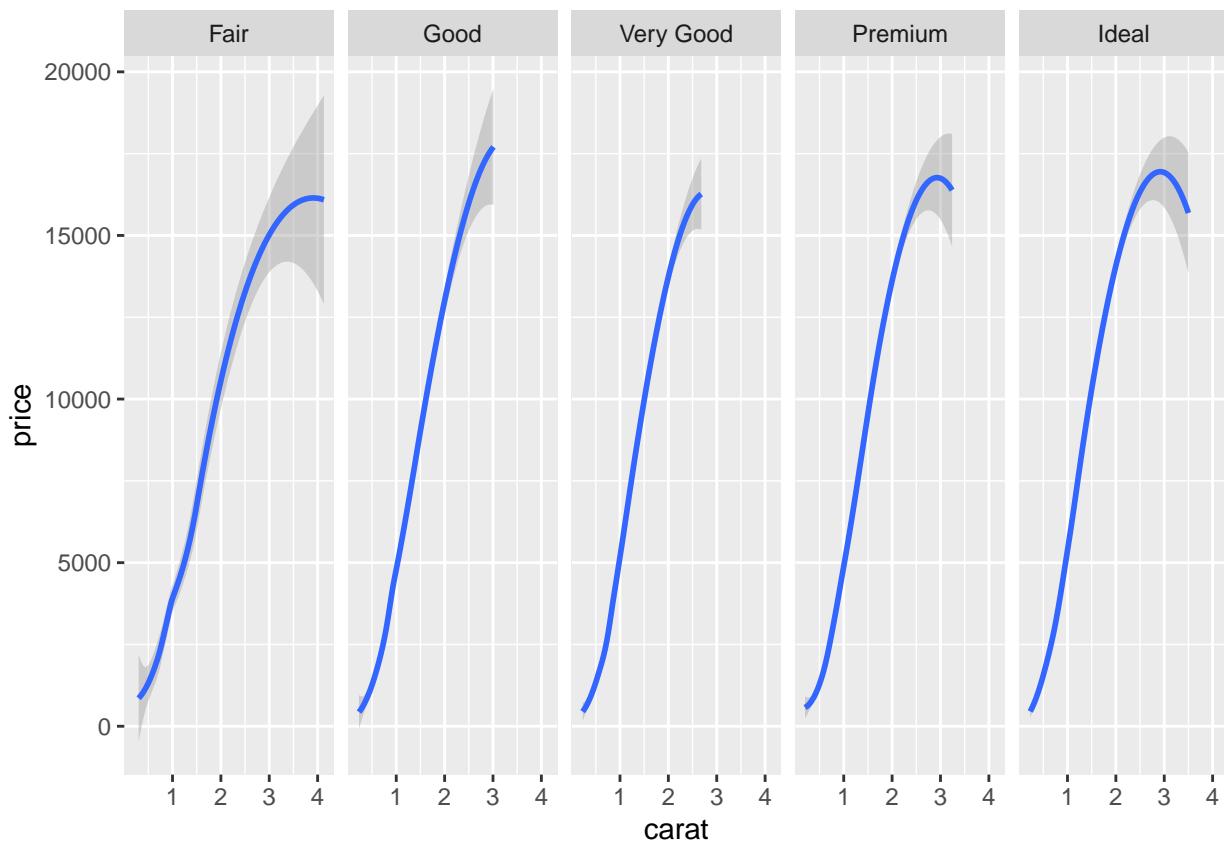


Pour obtenir cette représentation sur plusieurs fenêtres, on utilise

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

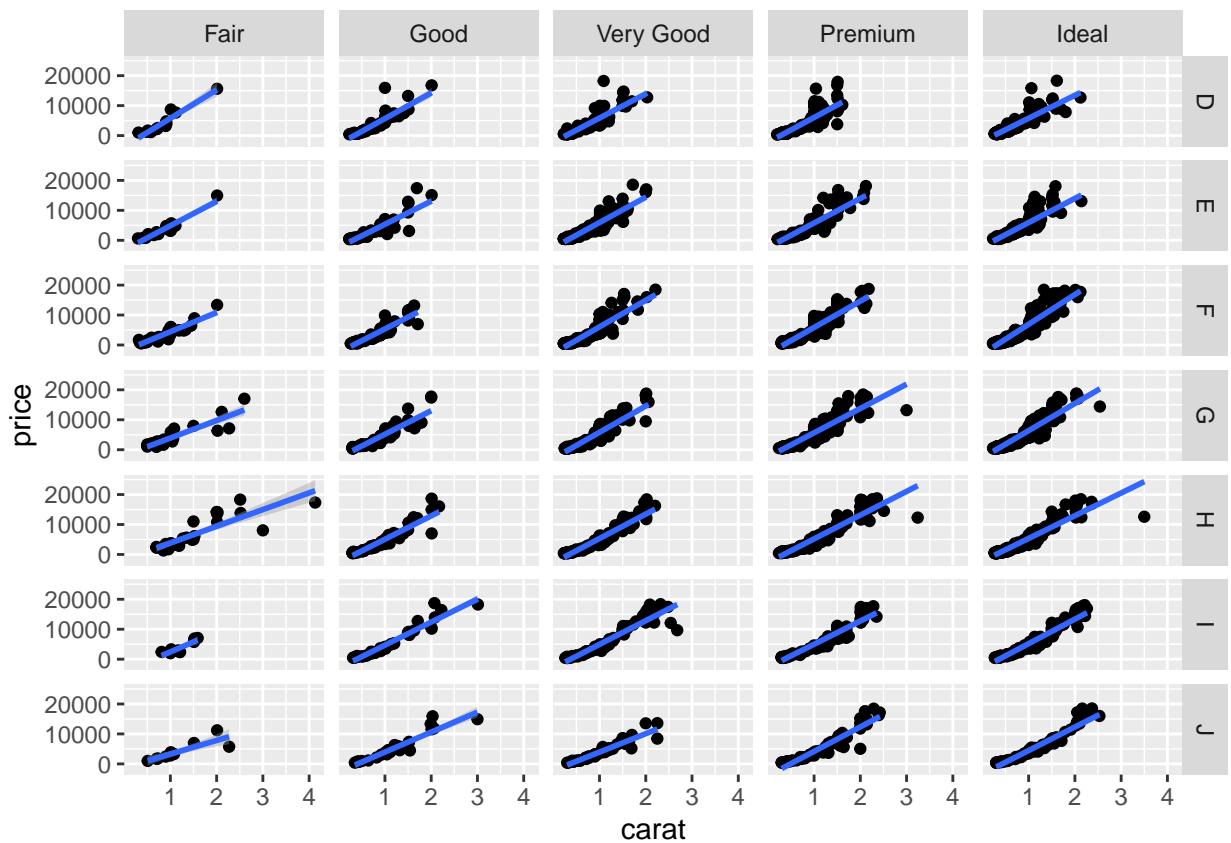


```
> ggplot(diamonds2)+aes(x=carat,y=price)+  
+   geom_smooth(method="loess")+facet_wrap(~cut,nrow=1)
```

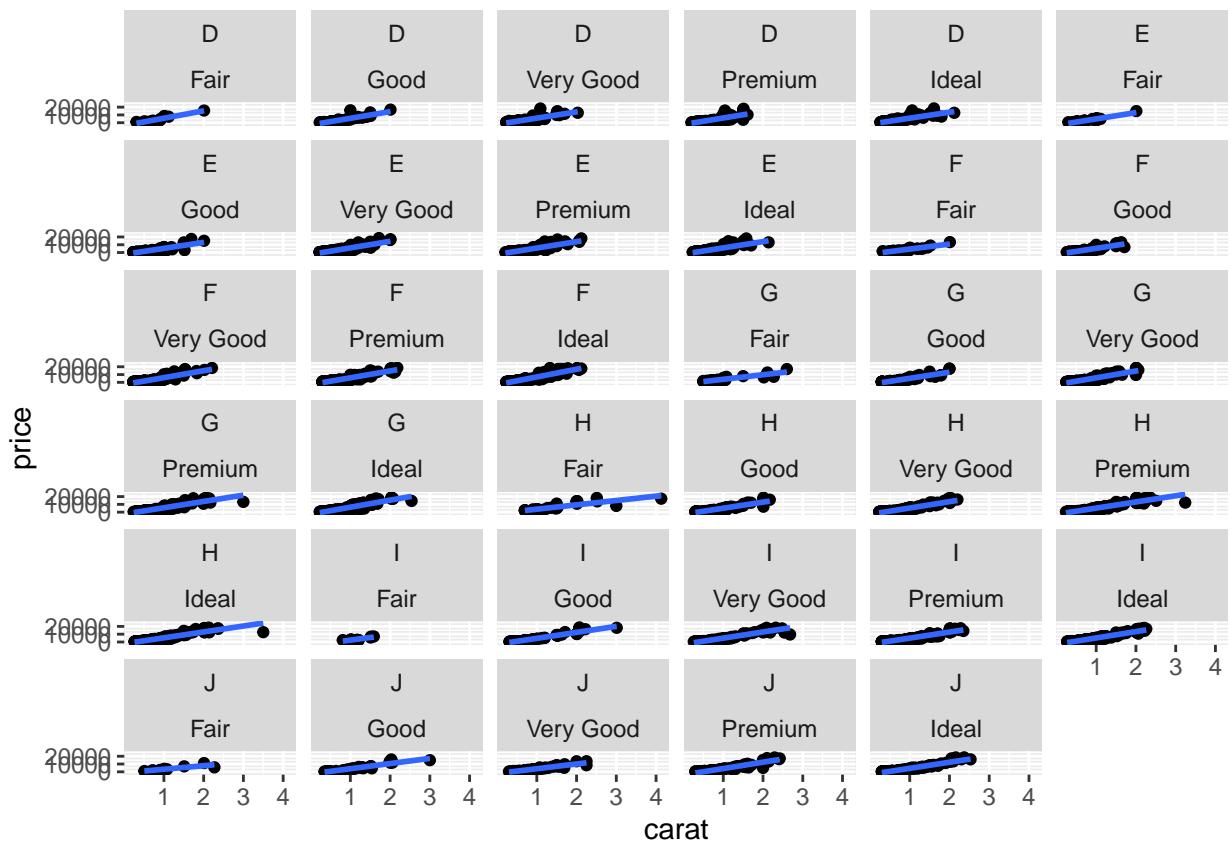


facet_grid et *facet_wrap* font des choses proches mais divisent la fenêtre de façon différente :

```
> 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)
```



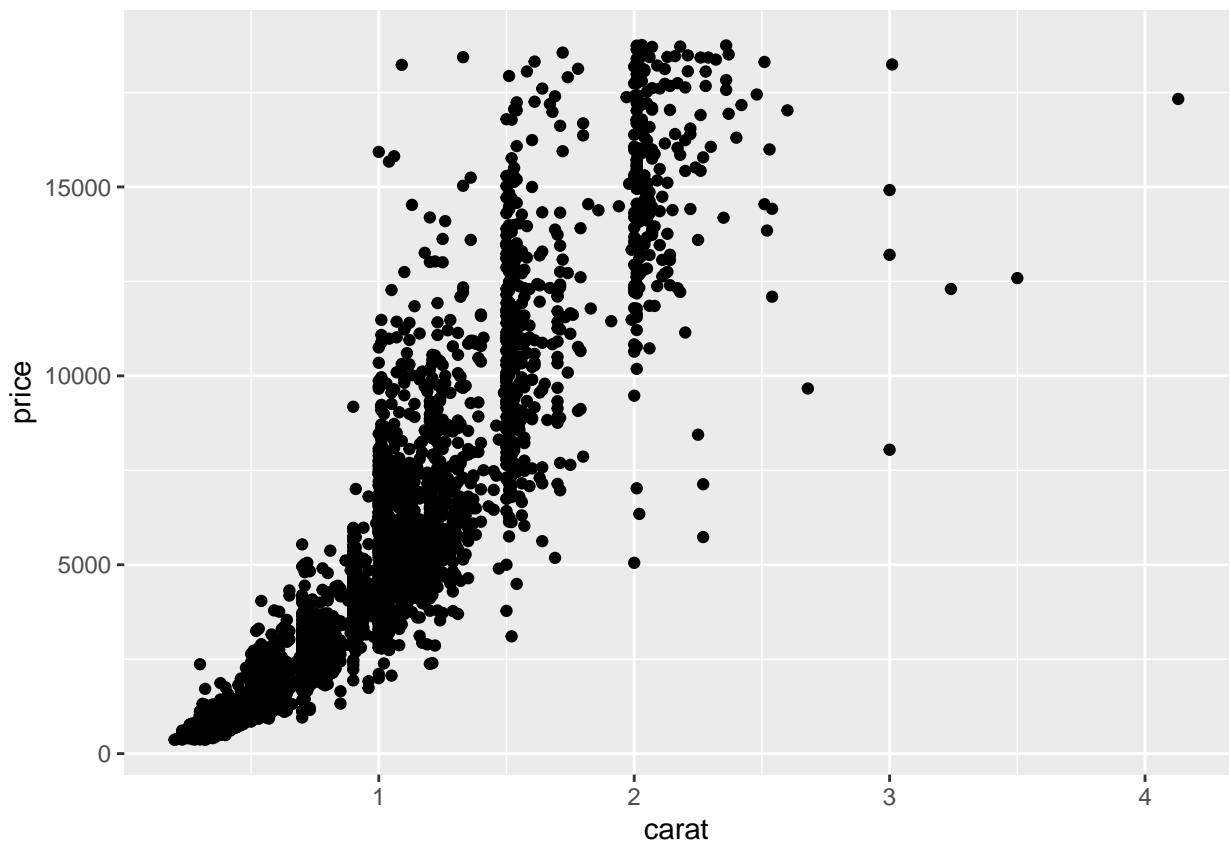
1.3 Compléments

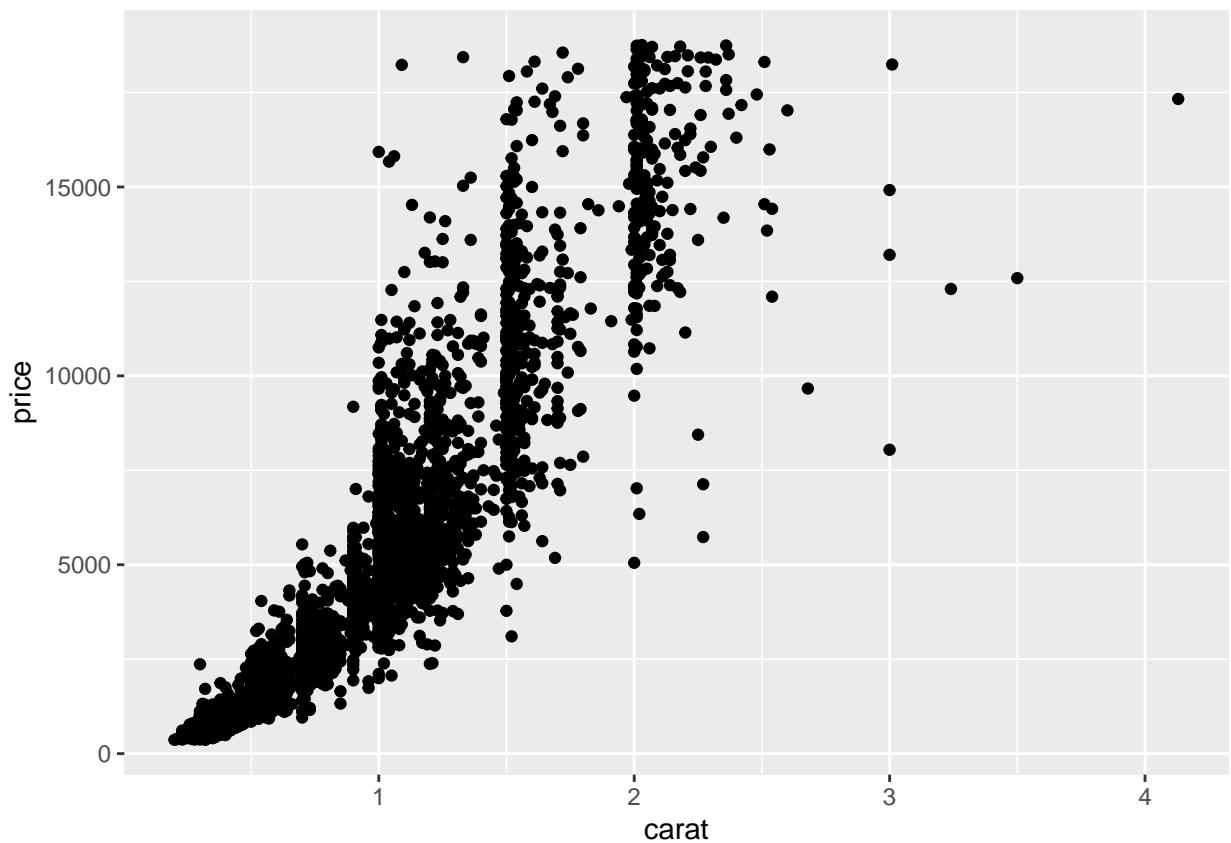
La syntaxe `ggplot` est définie selon le schéma :

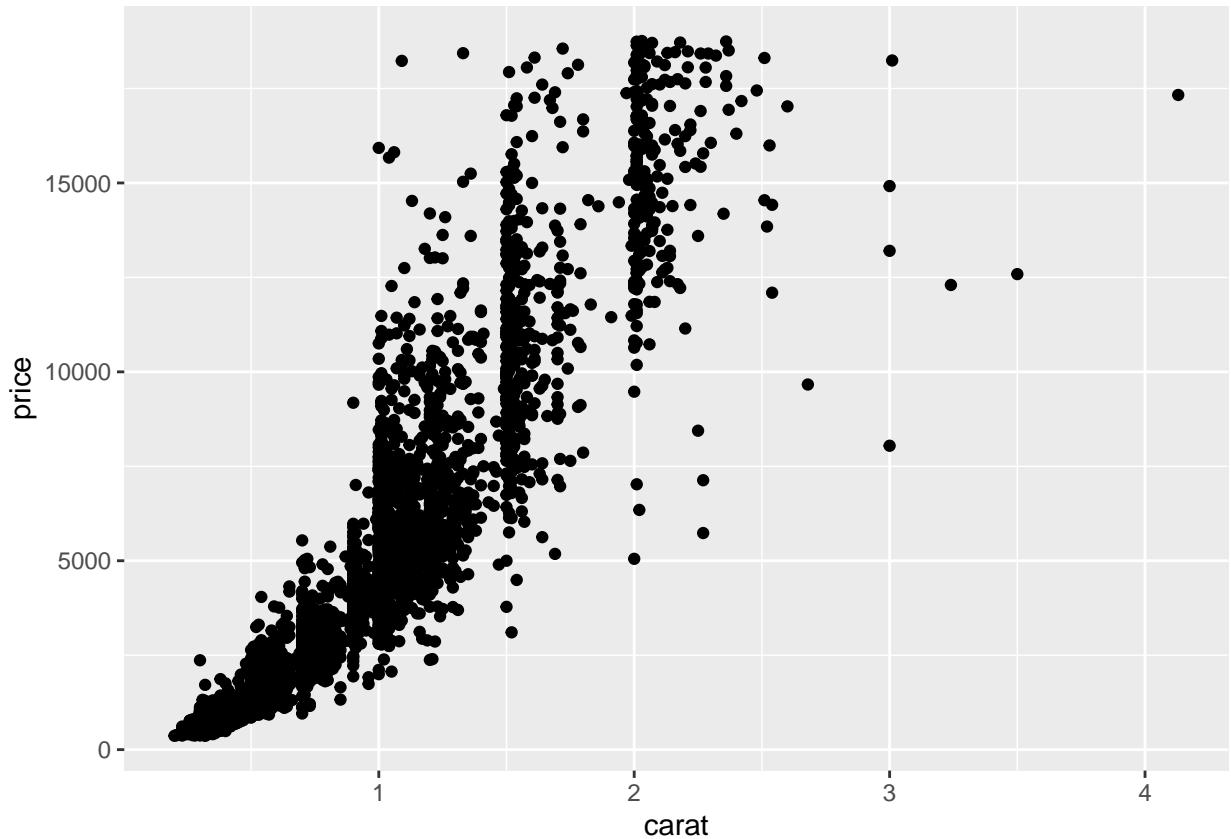
```
> ggplot() + aes() + geom_() + scale_()
```

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()
```



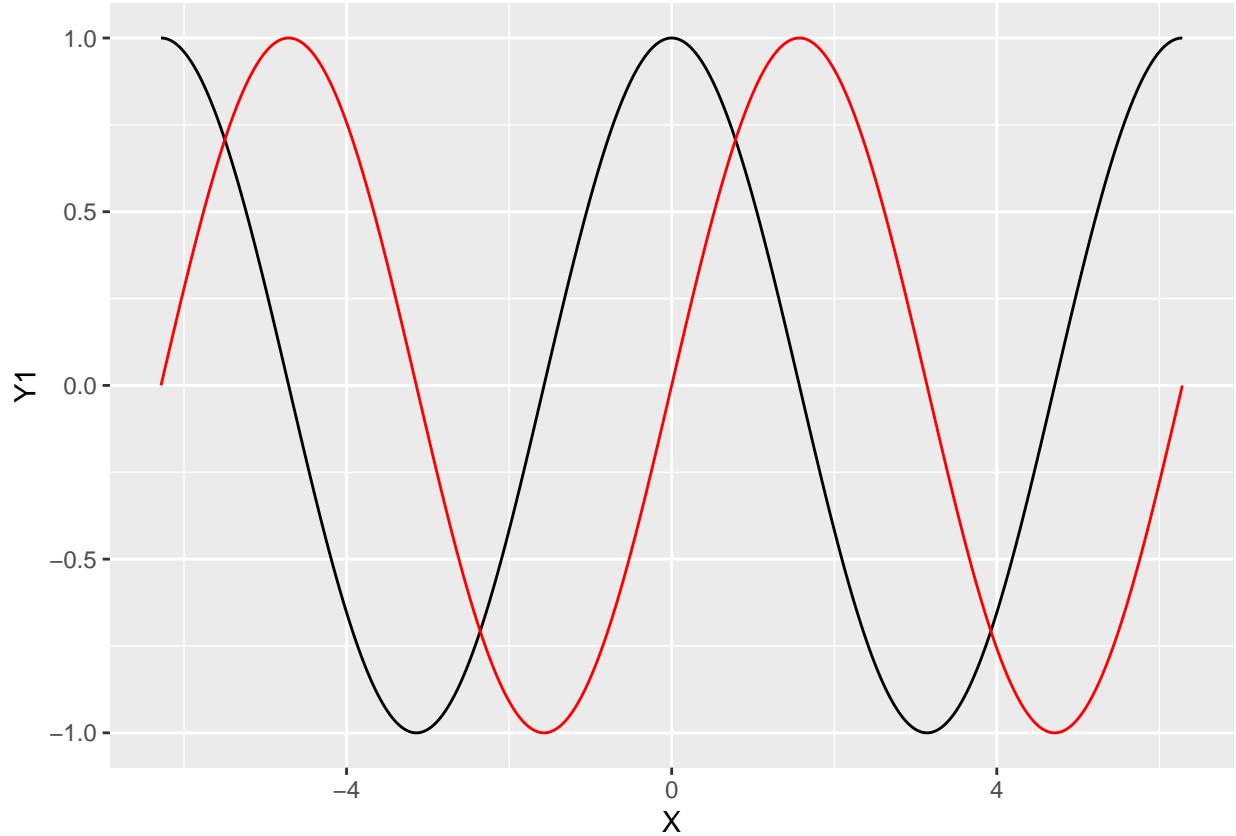




Ceci peut se révéler très utile lorsqu'on utilise des **aes** différents dans les **geom_**.

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

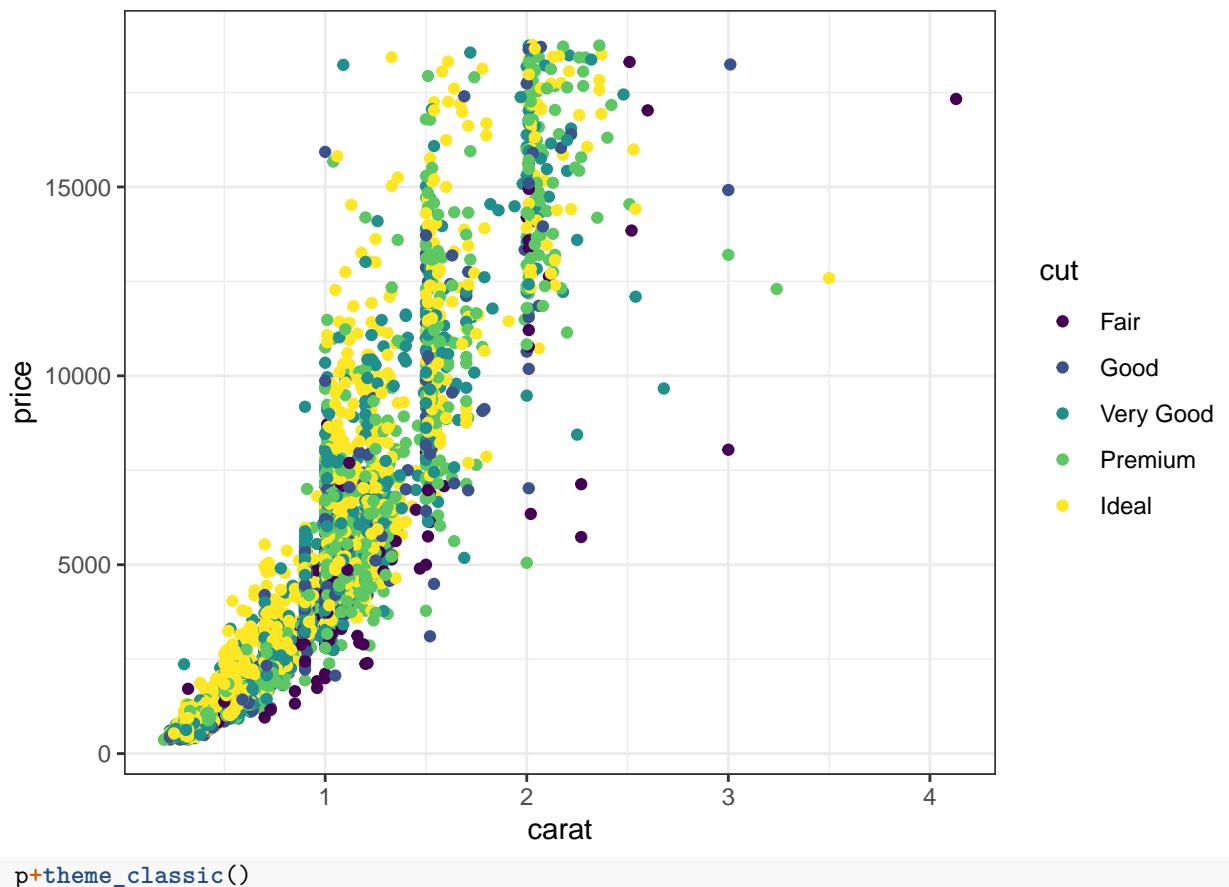
```
> X <- seq(-2*pi,2*pi,by=0.001)
> Y1 <- cos(X)
> Y2 <- sin(X)
> donnees1 <- data.frame(X,Y1)
> donnees2 <- data.frame(X,Y2)
> ggplot(donnees1)+geom_line(aes(x=X,y=Y1))+ 
+   geom_line(data=donnees2,aes(x=X,y=Y2),color="red")
```

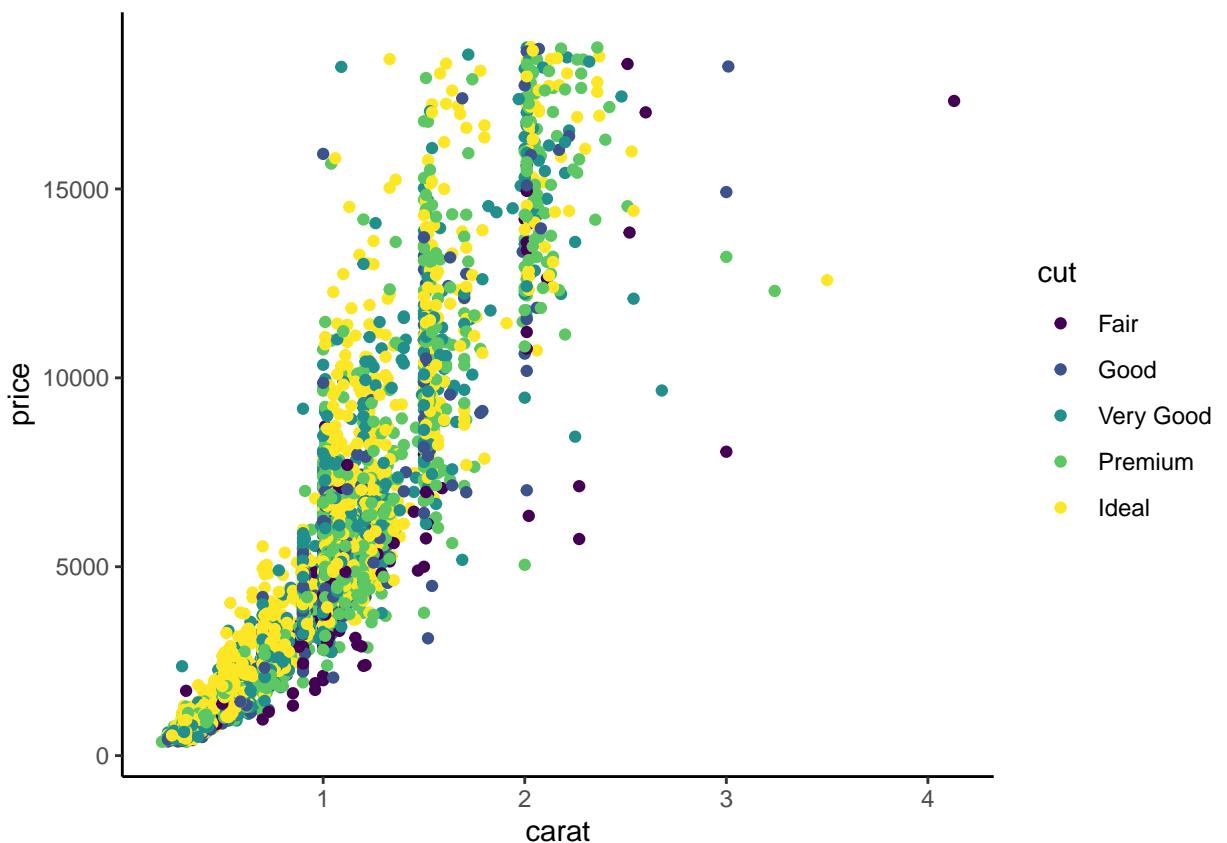


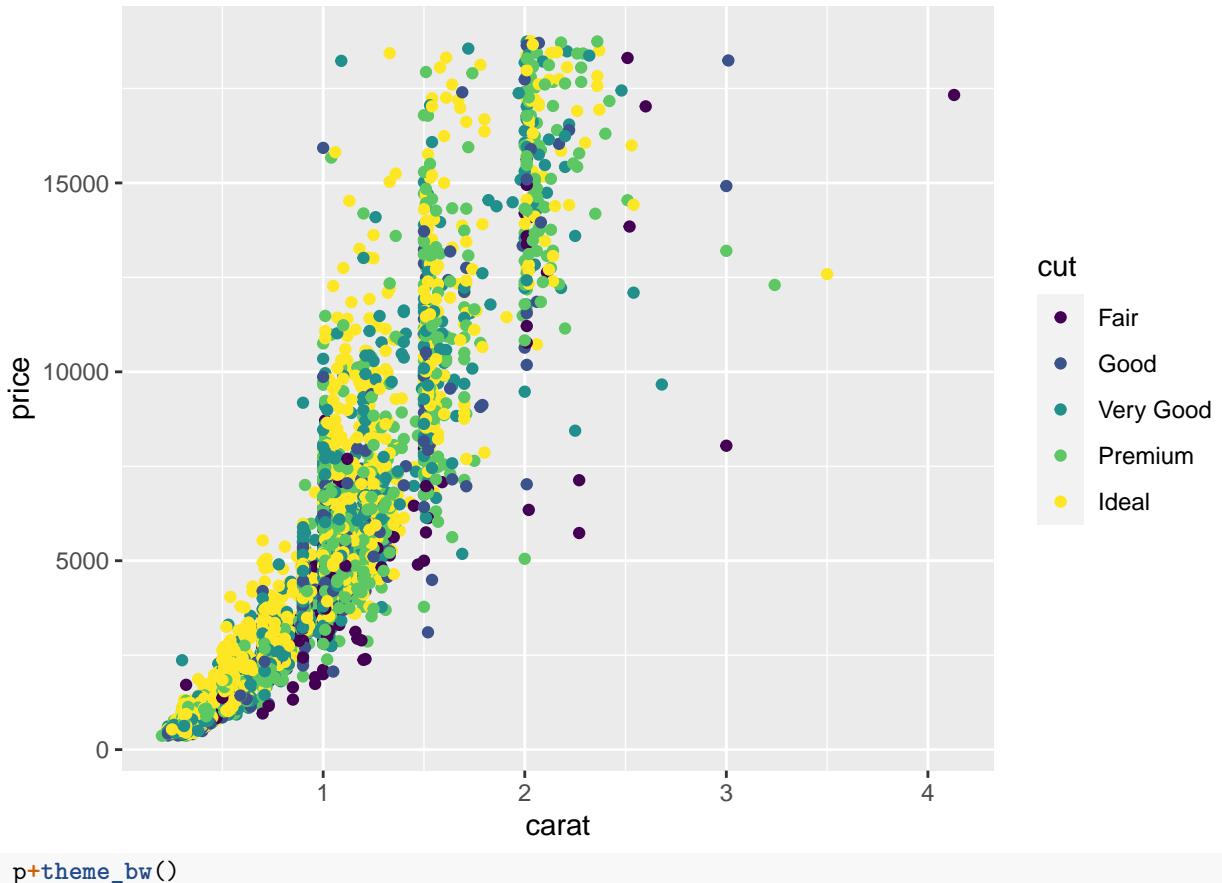
Il existe d'autres fonctions **ggplot** :

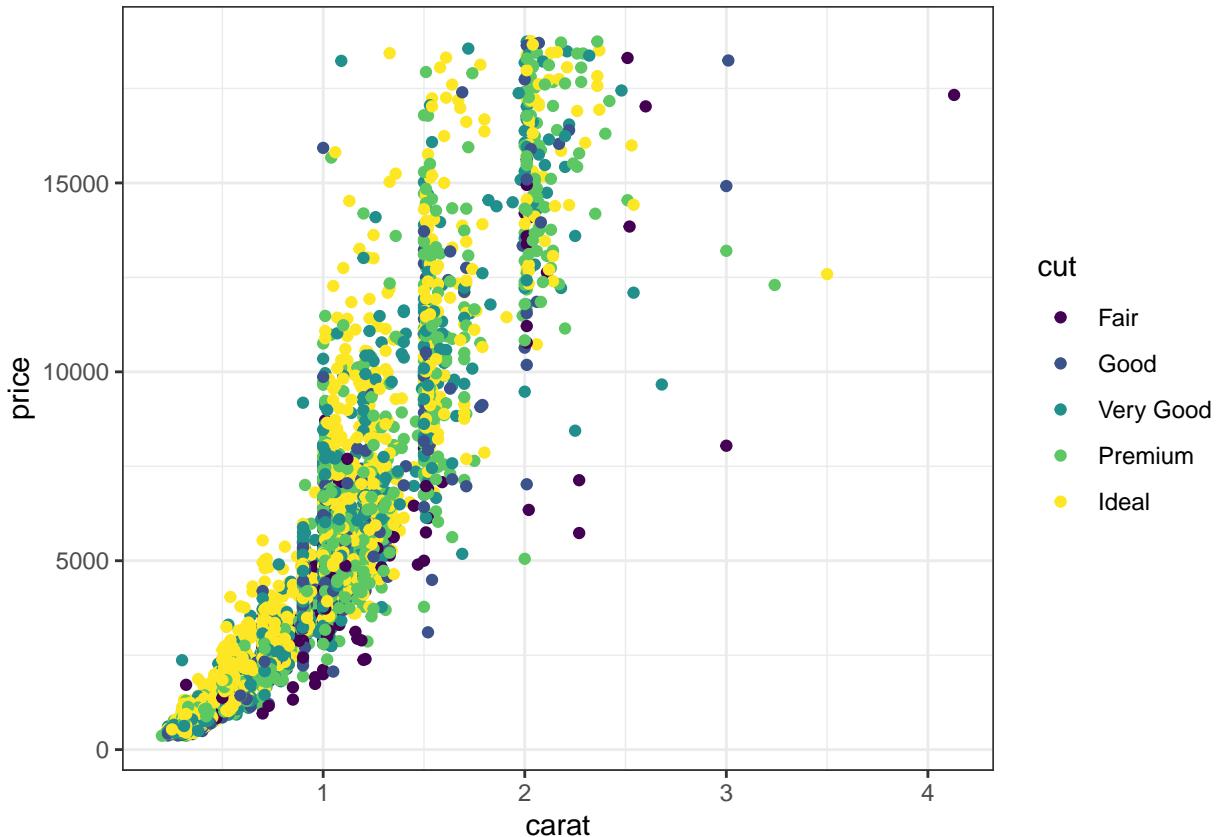
- **ggttitle** pour ajouter un titre.
- **ggsave** pour sauver un graphe.
- **theme_** pour changer le theme du graphe.

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









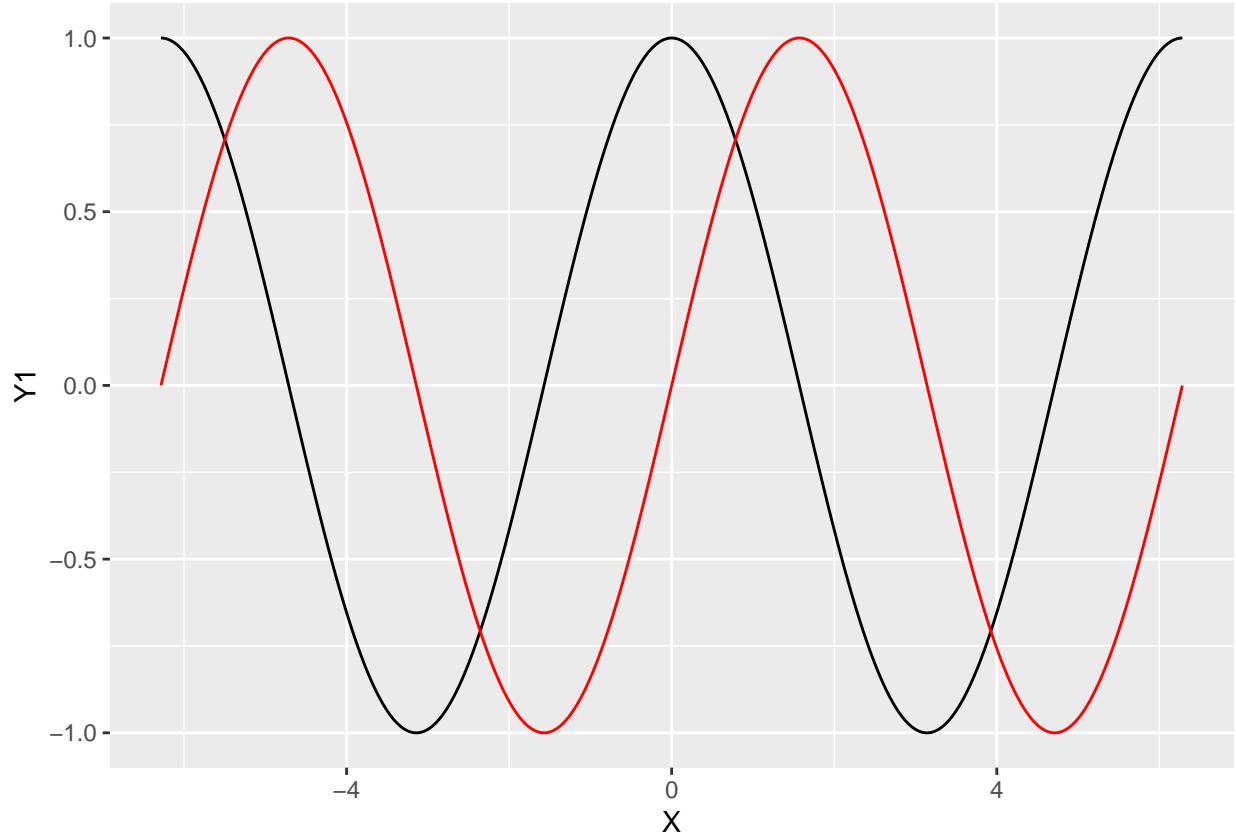
D'autres thèmes sont disponibles dans le package **ggtheme**. On pourra également parler de la fonction **set_theme** qui permet de préciser modifier le thème par défaut pour un document **Markdown**.

1.4 Quelques exercices supplémentaires

Exercice 1.9 (Fonctions cosinus et sinus).

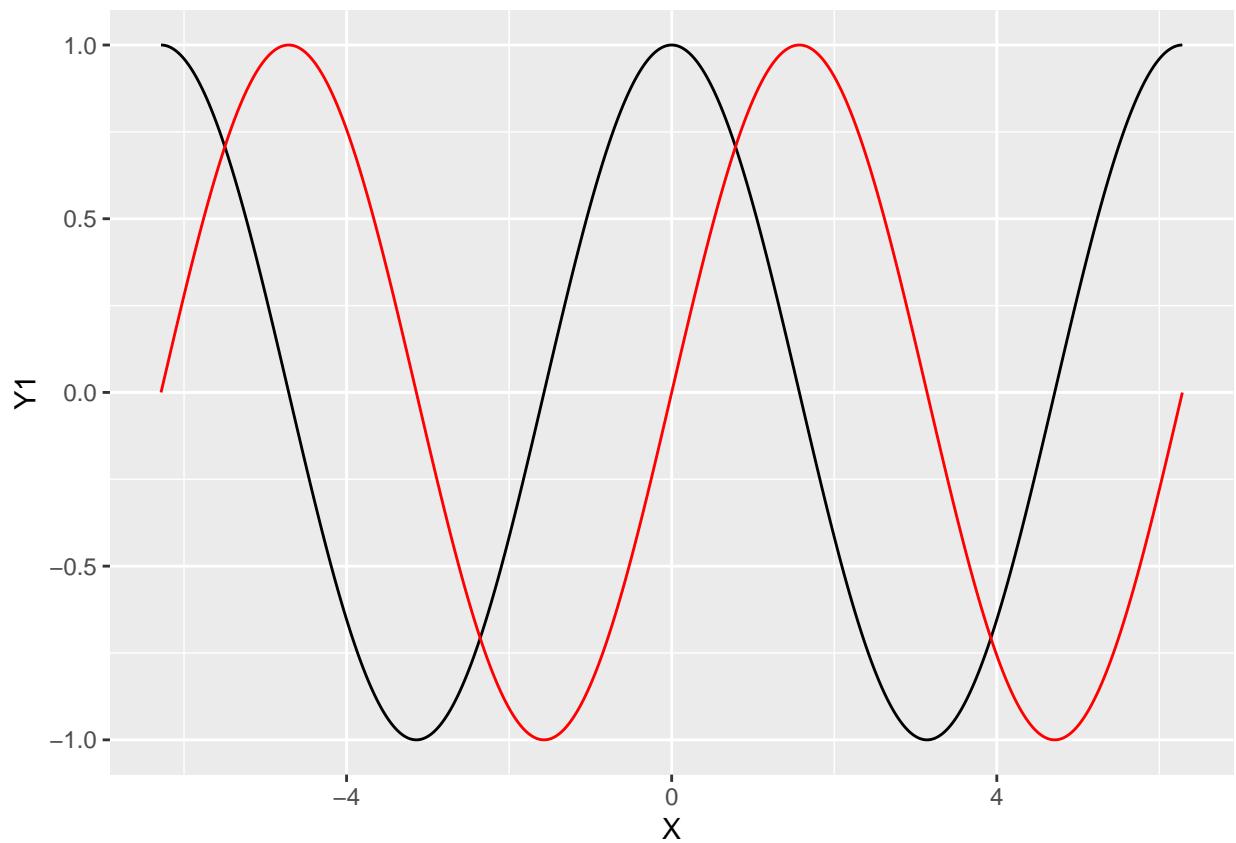
- Tracer les fonctions sinus et cosinus. On utilisera tout d'abord deux jeux de données : un pour le sinus, l'autre pour le cosinus.

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

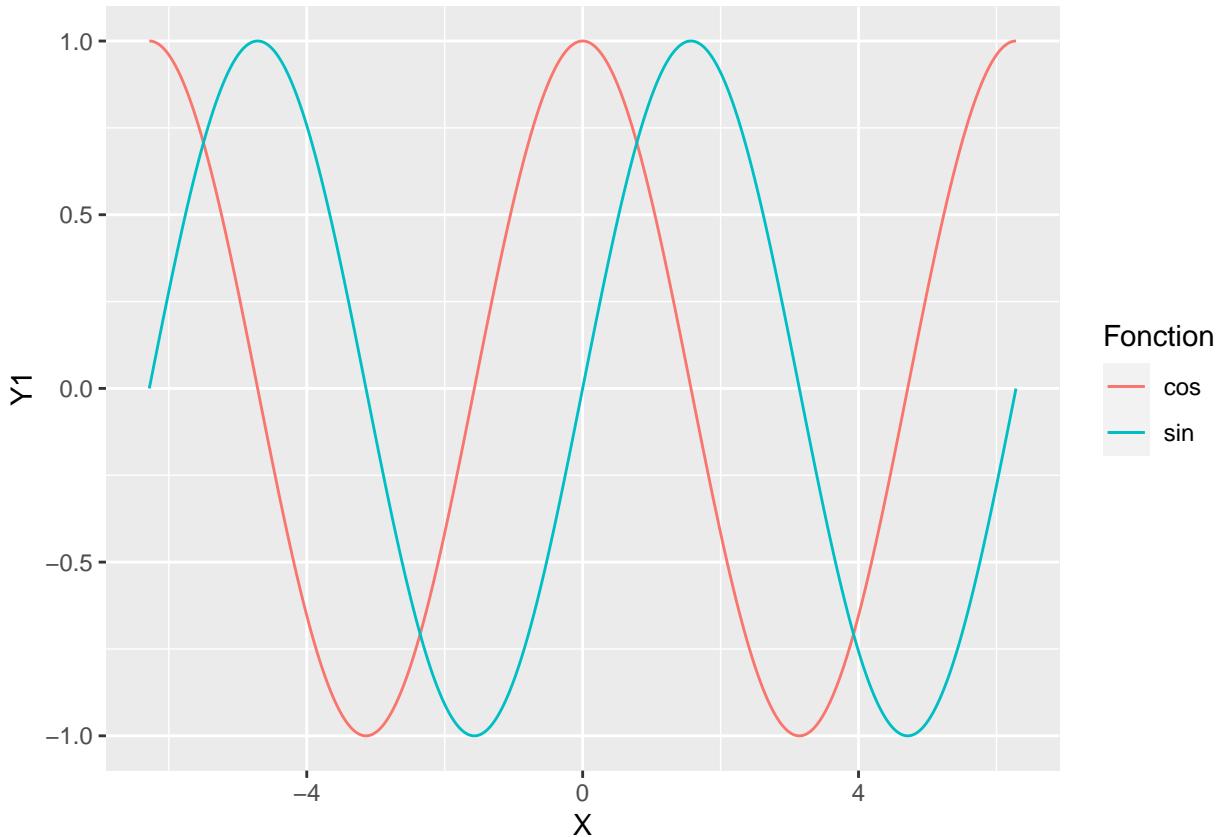


2. Faire la même chose avec un jeu de données et deux appels à la fonction `geom_line`. On pourra ajouter une légende.

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

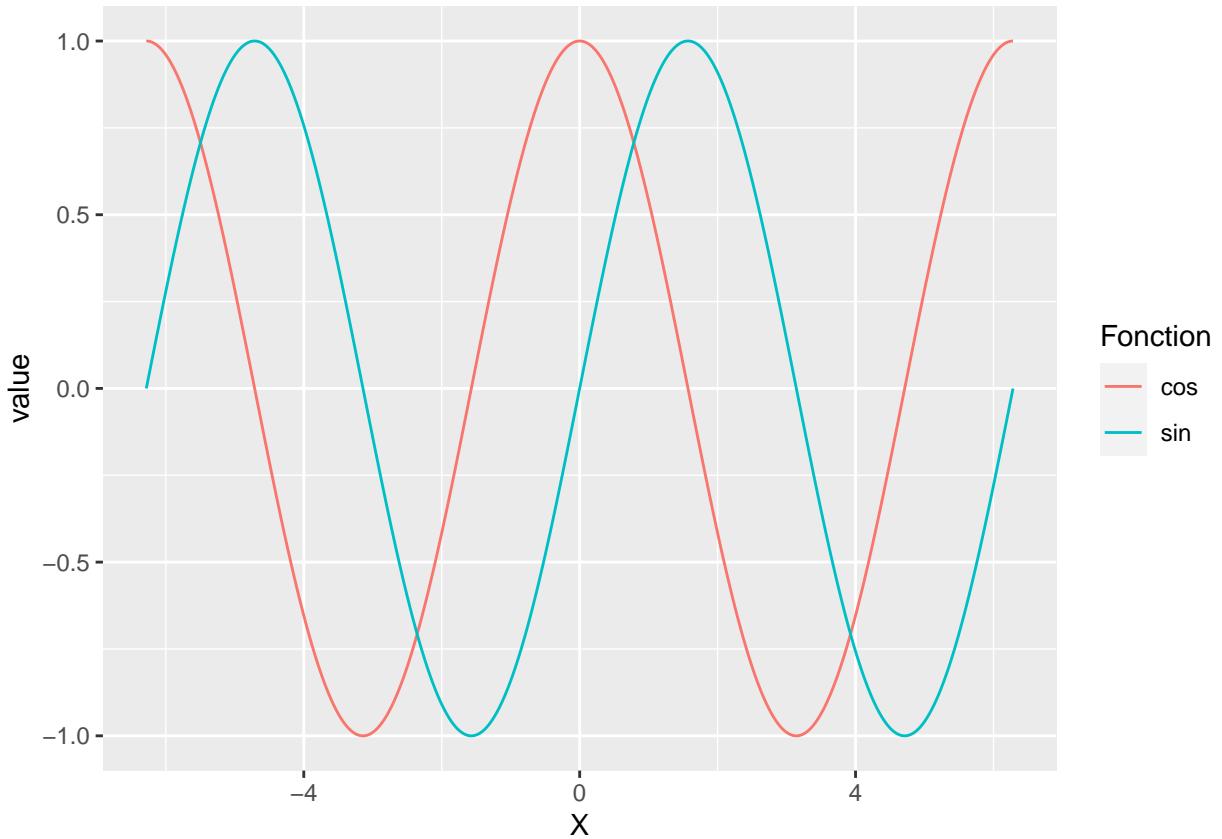


```
> #ou pour la légende  
> ggplot(donnees)+aes(x=x,y=Y1)+geom_line(aes(color="cos"))+  
+   geom_line(aes(y=Y2,color="sin"))+labs(color="Fonction")
```



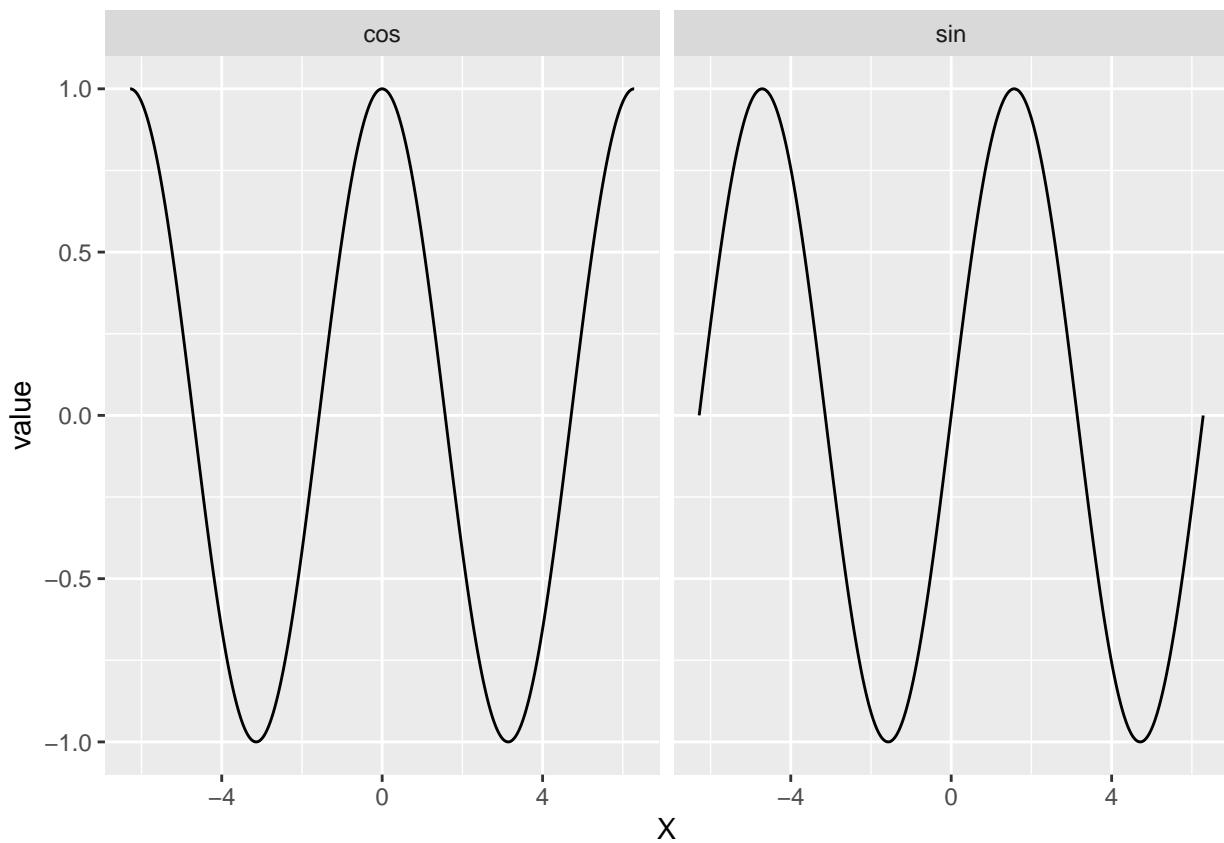
3. Faire la même chose avec un jeu de données et un seul appel à `geom_line`. On pourra utiliser la fonction `gather` du `tidyverse`.

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



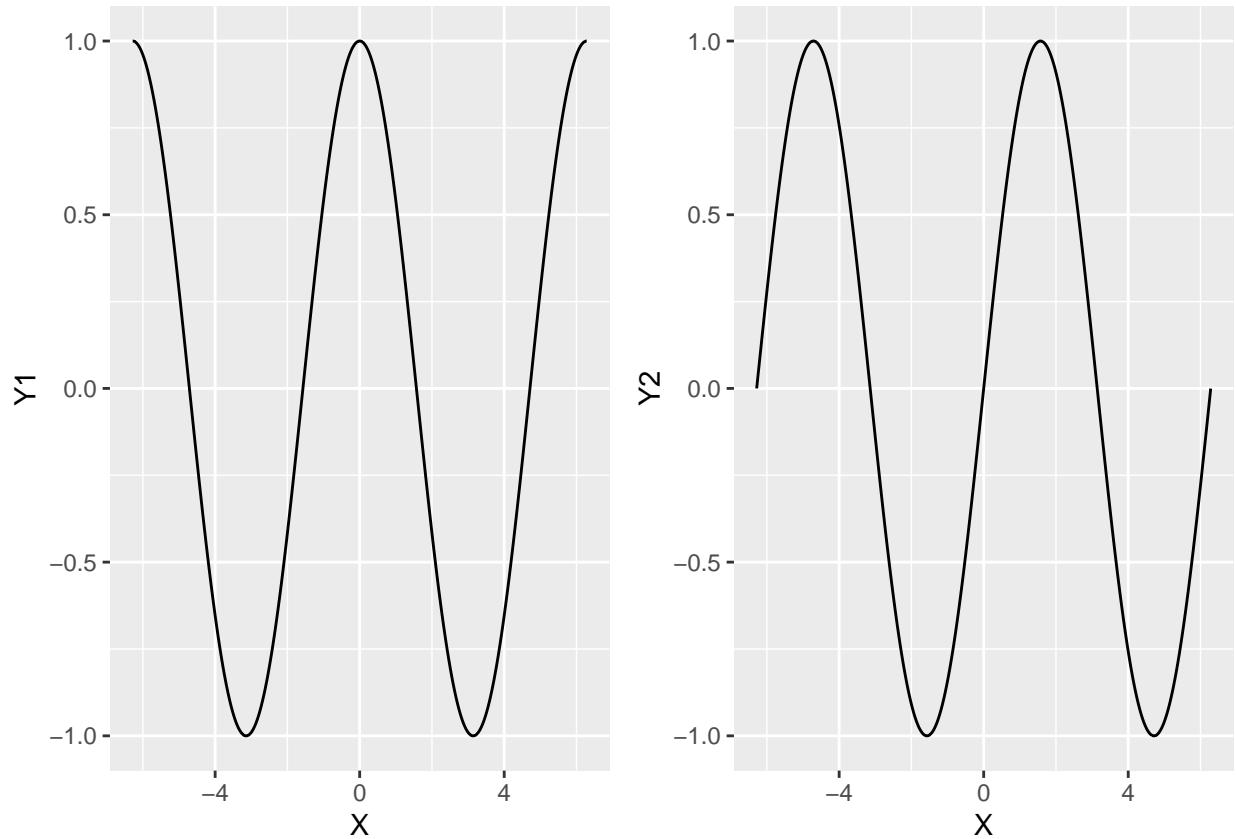
4. Tracer les deux fonctions sur deux fenêtres graphiques (utiliser `facet_wrap`).

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



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

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



Exercice 1.10 (Différents graphes).

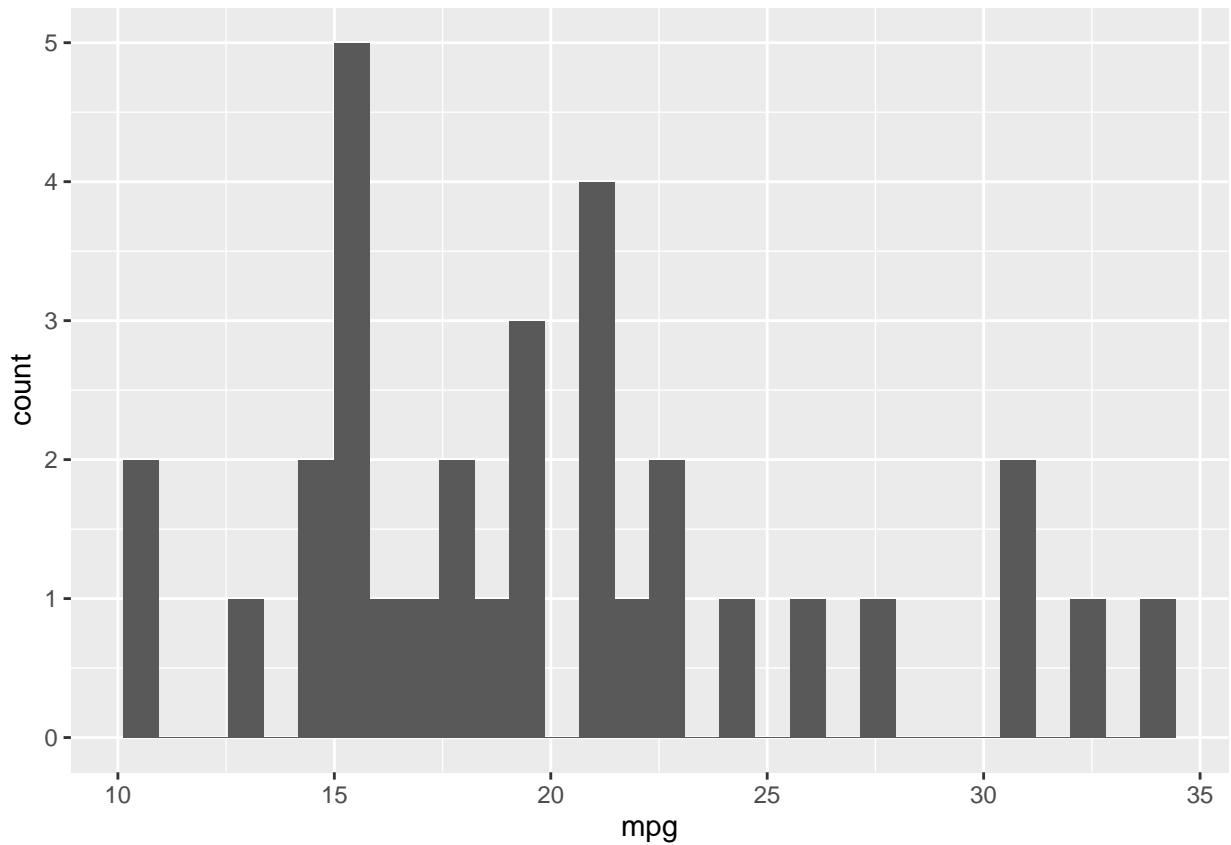
On considère les données **mtcars**

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

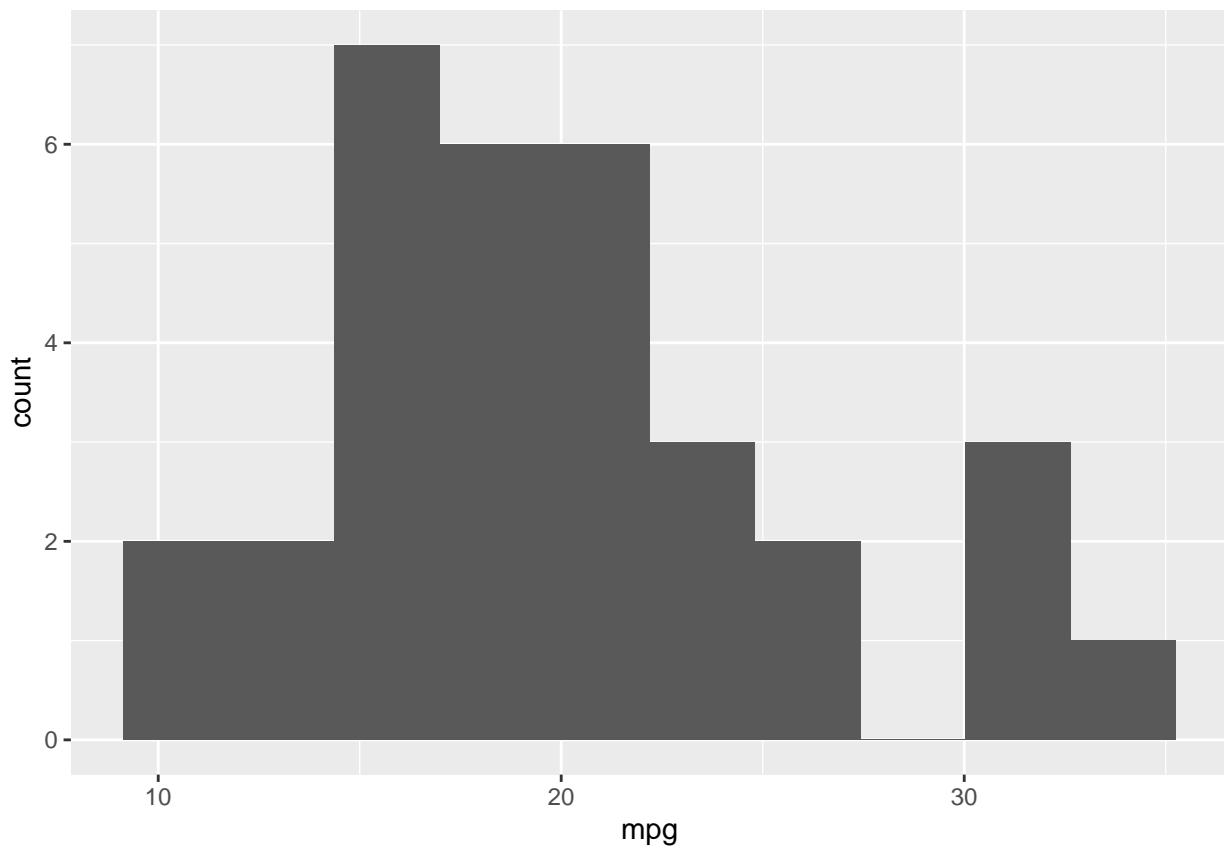
##          mpg              cyl             disp            hp
##  Min.   :10.40   Min.   :4.000   Min.   :71.1   Min.   :52.0
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8  1st Qu.:96.5
##  Median :19.20   Median :6.000   Median :196.3  Median :123.0
##  Mean    :20.09   Mean    :6.188   Mean    :230.7  Mean    :146.7
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0  3rd Qu.:180.0
##  Max.    :33.90   Max.    :8.000   Max.    :472.0  Max.    :335.0
##          drat              wt             qsec            vs
##  Min.   :2.760   Min.   :1.513   Min.   :14.50  Min.   :0.0000
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89  1st Qu.:0.0000
##  Median :3.695   Median :3.325   Median :17.71  Median :0.0000
##  Mean    :3.597   Mean    :3.217   Mean    :17.85  Mean    :0.4375
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90  3rd Qu.:1.0000
##  Max.    :4.930   Max.    :5.424   Max.    :22.90  Max.    :1.0000
##          am               gear            carb
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
##  Median :0.0000   Median :4.000   Median :2.000
##  Mean    :0.4062   Mean    :3.688   Mean    :2.812
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
##  Max.    :1.0000   Max.    :5.000   Max.    :8.000
```

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

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

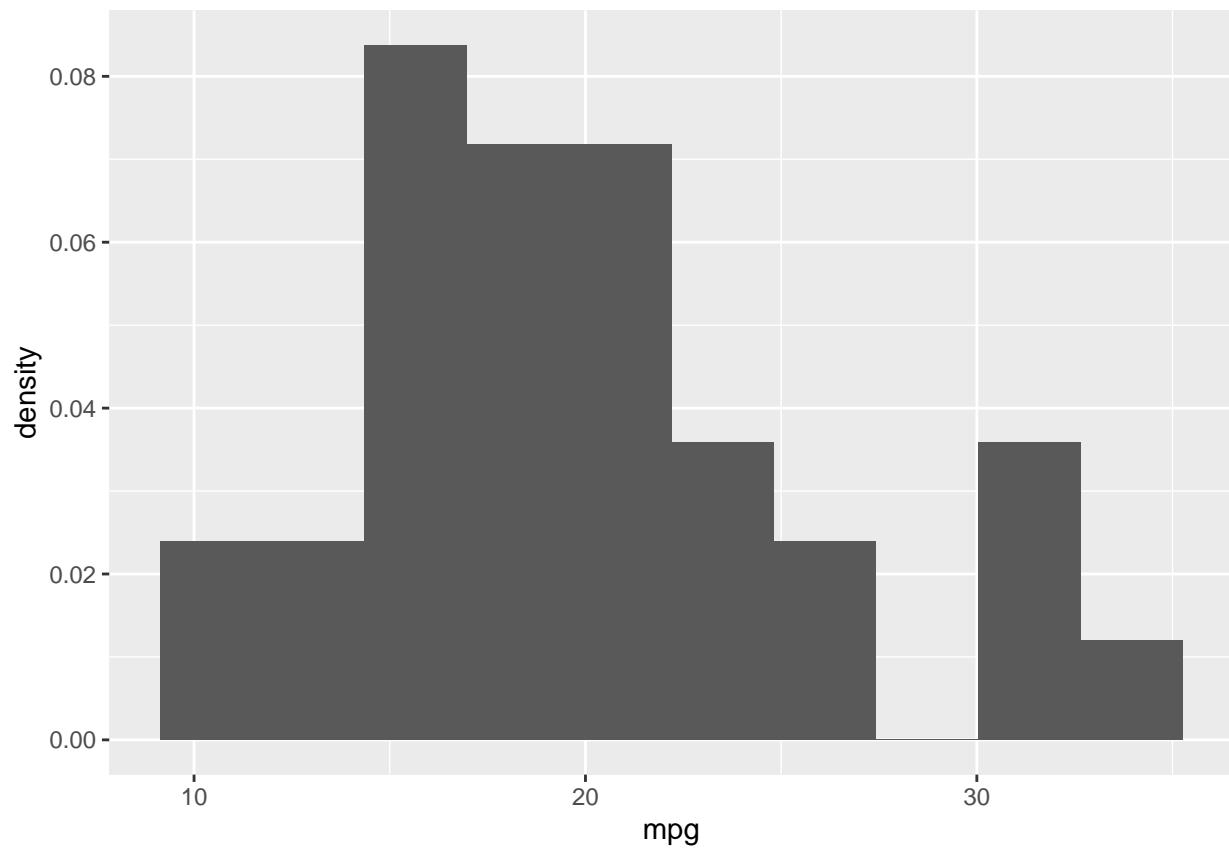


```
> ggplot(mtcars)+aes(x=mpg)+geom_histogram(bins=10)
```



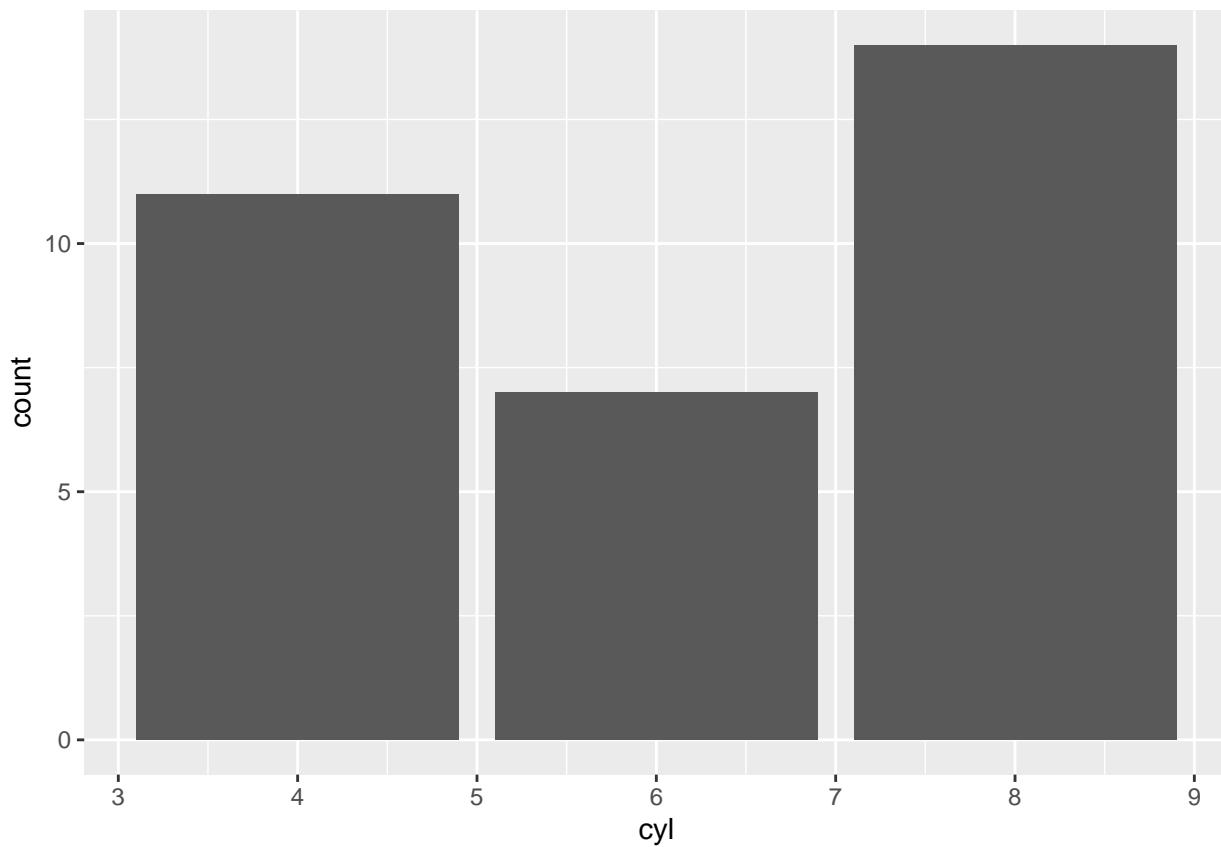
2.Tracer l'histogramme de la densité.

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



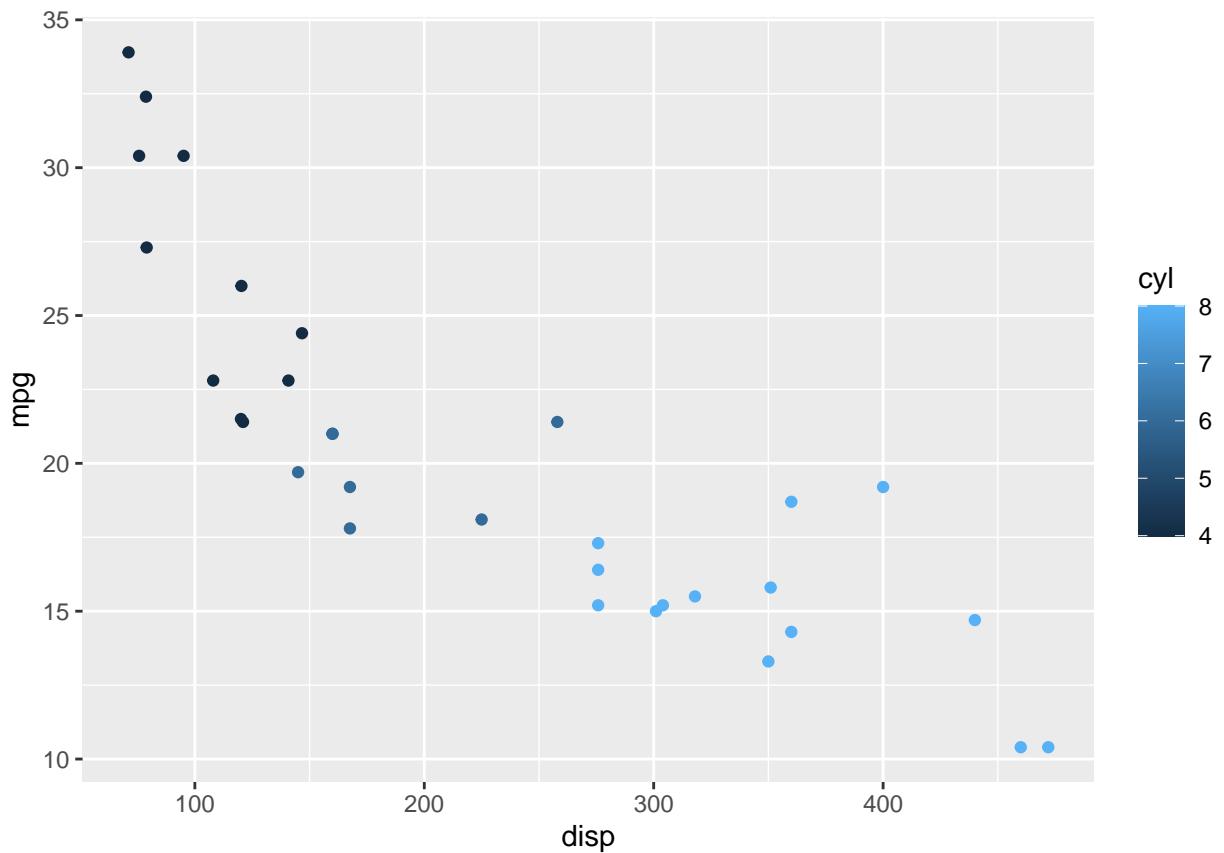
3. Tracer le diagramme en barres de cyl.

```
> ggplot(mtcars)+aes(x=cyl)+geom_bar()
```

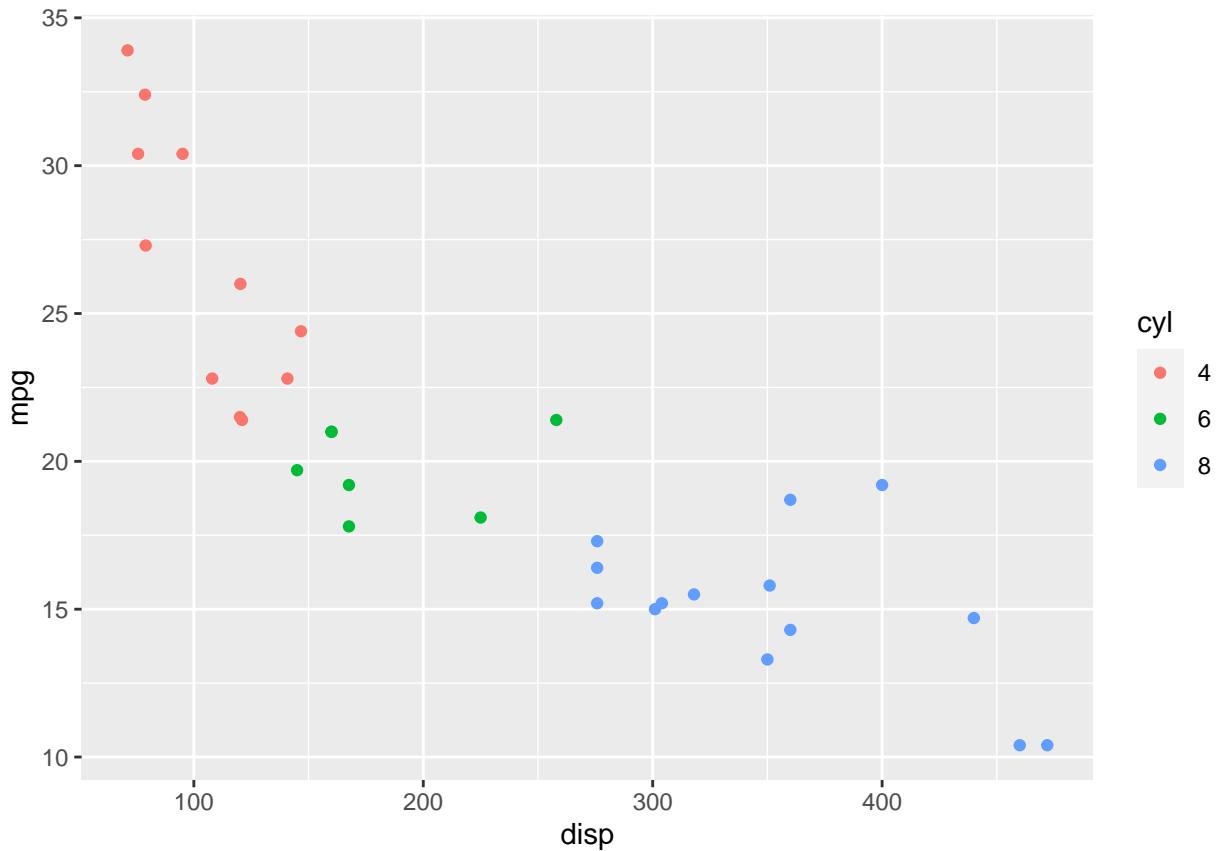


4. Tracer le nuage de points **disp** vs **mpg** en utilisant une couleur différente pour chaque valeur de **cyl**.

```
> ggplot(mtcars)+aes(x=disp,y=mpg,color=cyl)+geom_point()
```

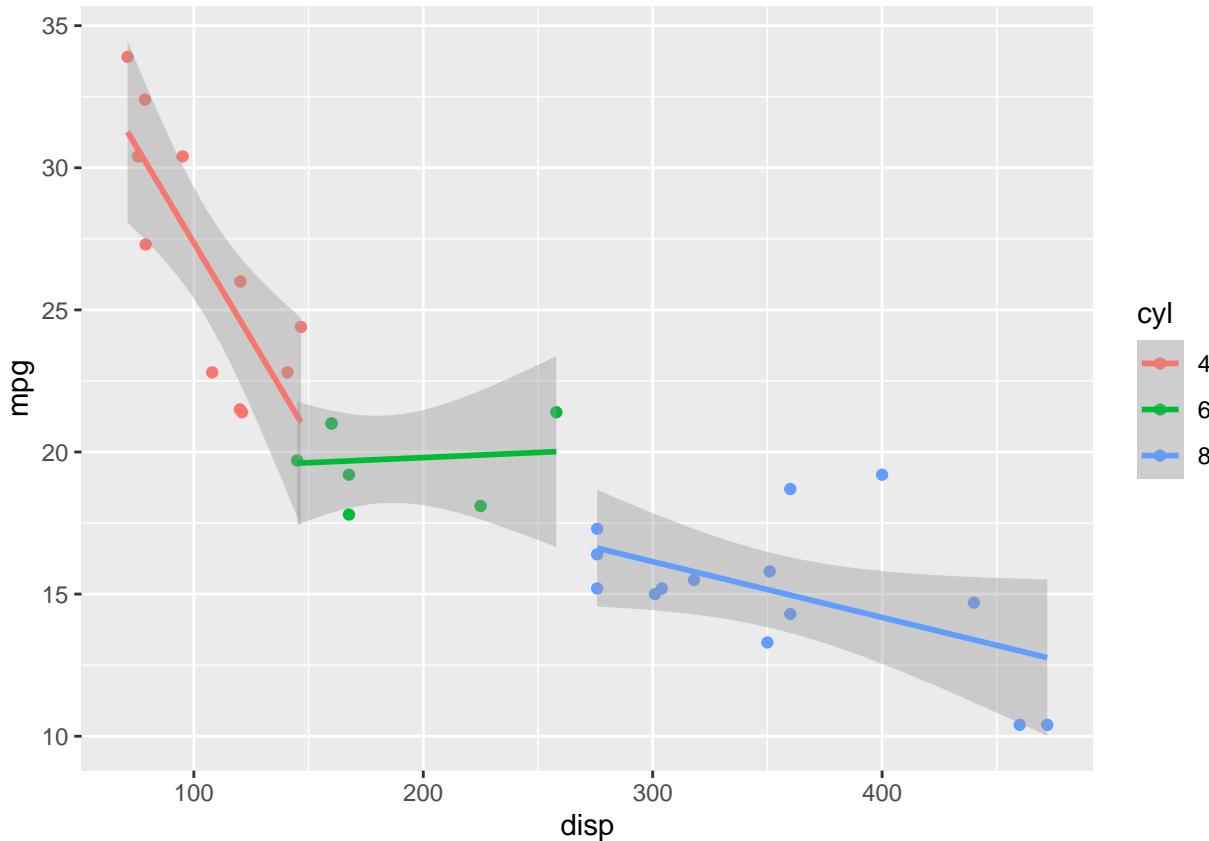


```
> ggplot(mtcars)+aes(x=disp,y=mpg,color=as.factor(cyl))+geom_point()+labs(color="cyl")
```



5. Ajouter le lisseur linéaire sur le graphe.

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



Exercice 1.11 (Résidus pour régression simple).

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

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

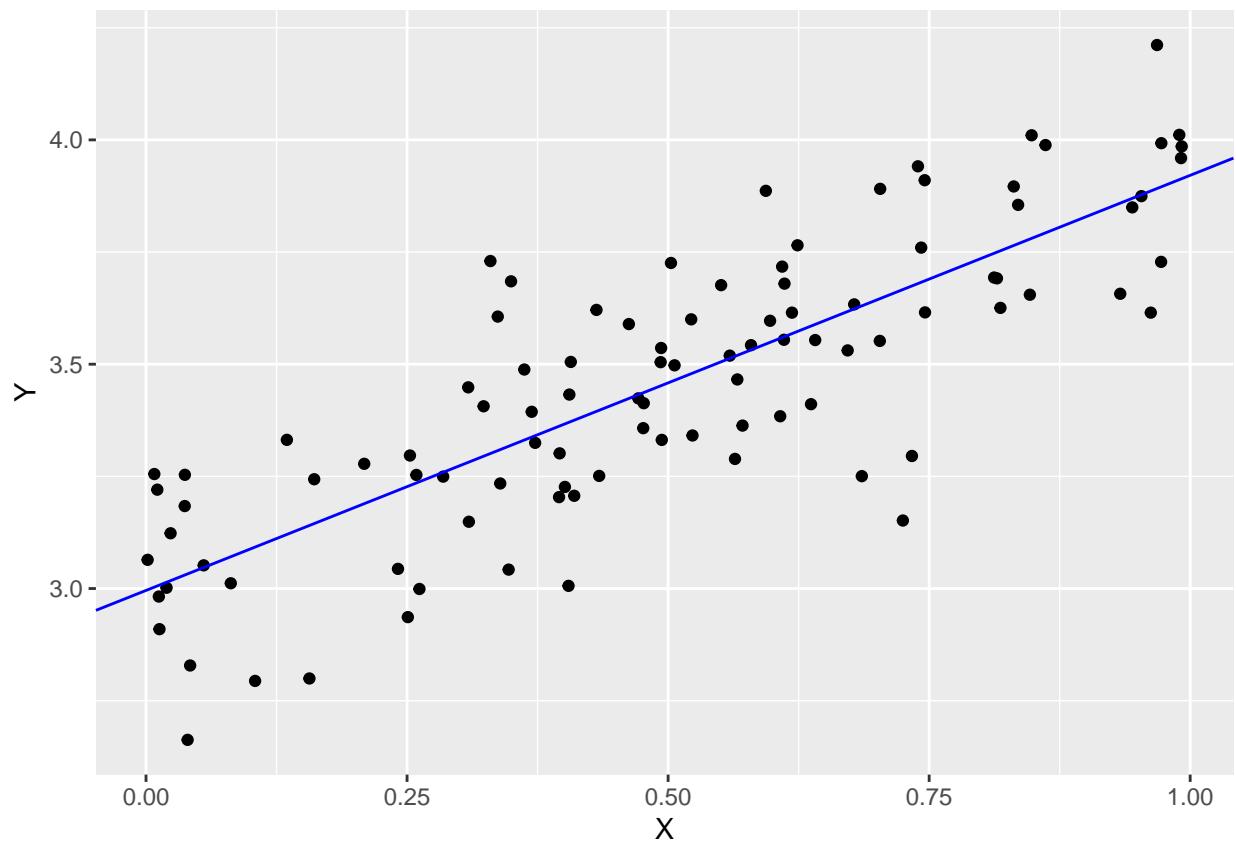
où X_i sont i.i.d. de loi uniforme sur $[0, 1]$ et ε_i sont i.i.d. de loi gaussienne $N(0, 0.2^2)$ (utiliser **runif** et **rnorm**).

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

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

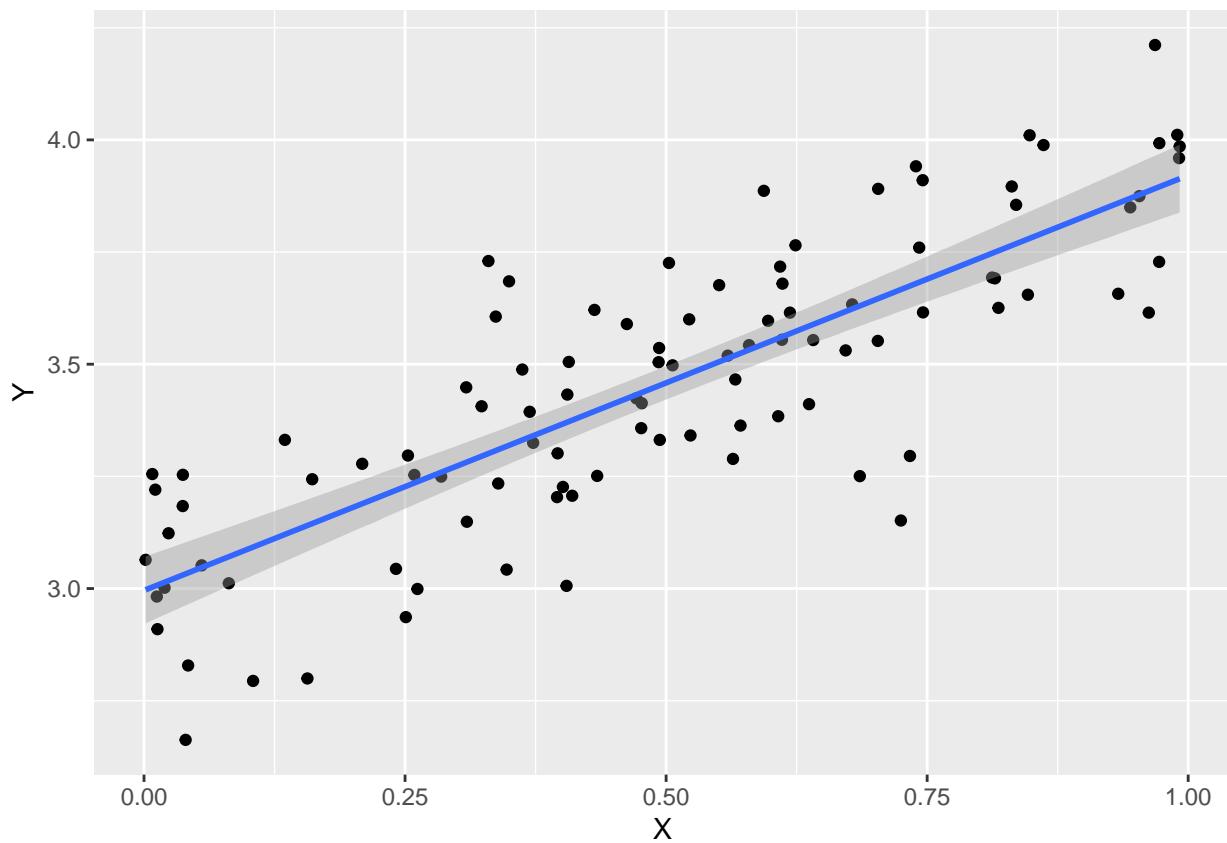
On le fait d'abord “à la main” en calculant l'équation de la droite de régression.

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



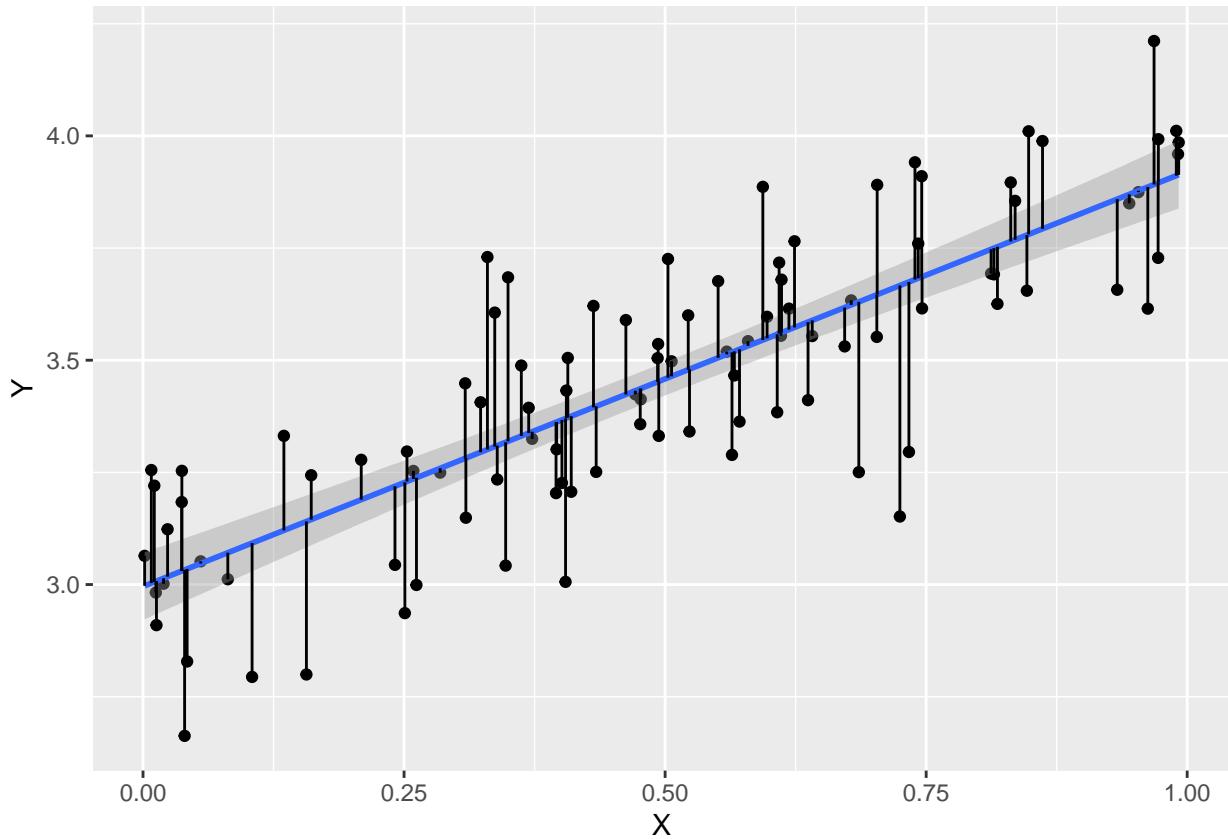
On peut avoir le tracé directement avec geom_smooth.

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



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

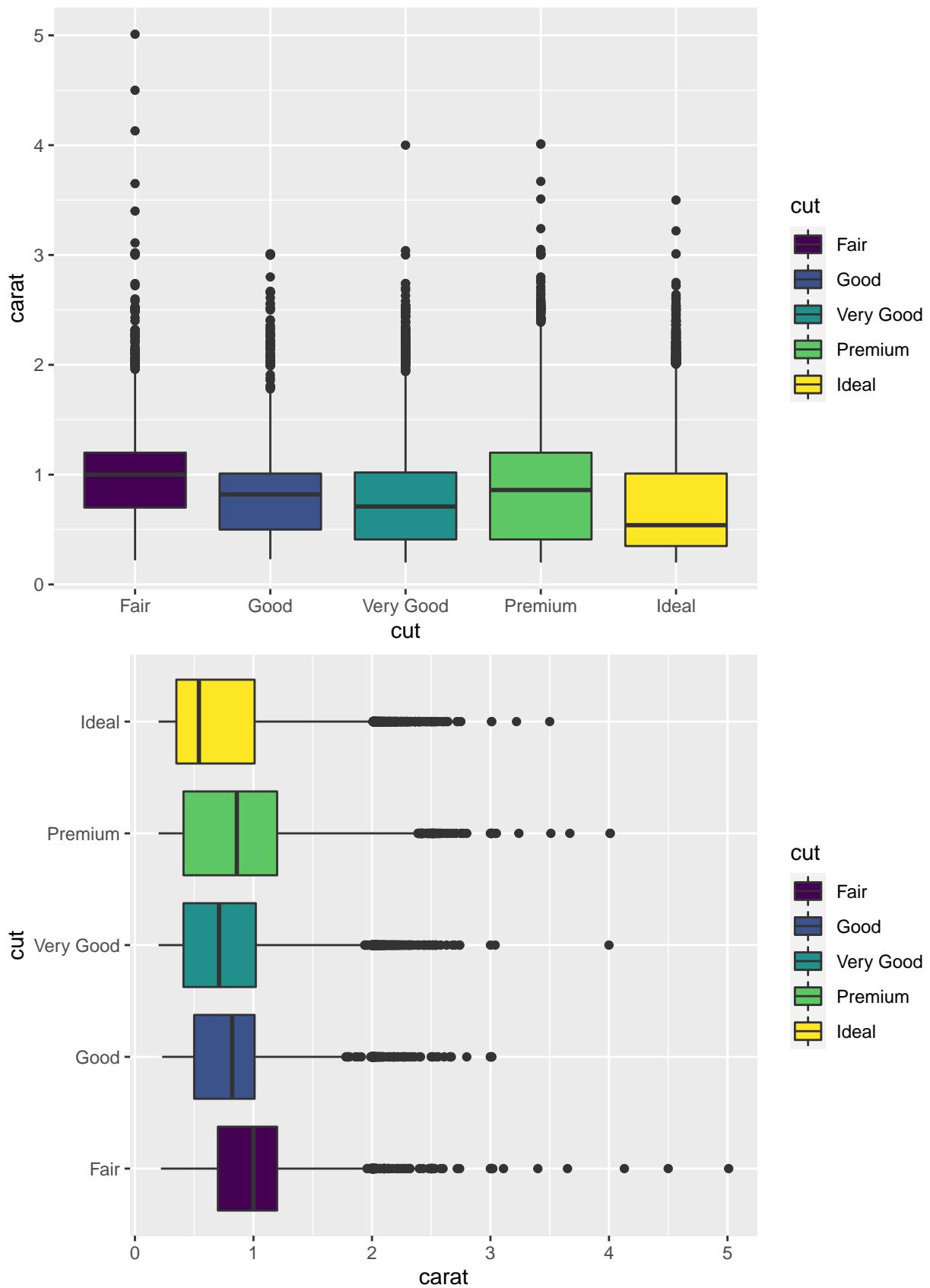
```
> ggplot(D)+aes(x=X,y=Y)+geom_point() +geom_smooth(method="lm")+
+  geom_segment(aes(xend=X,yend=fit))
```

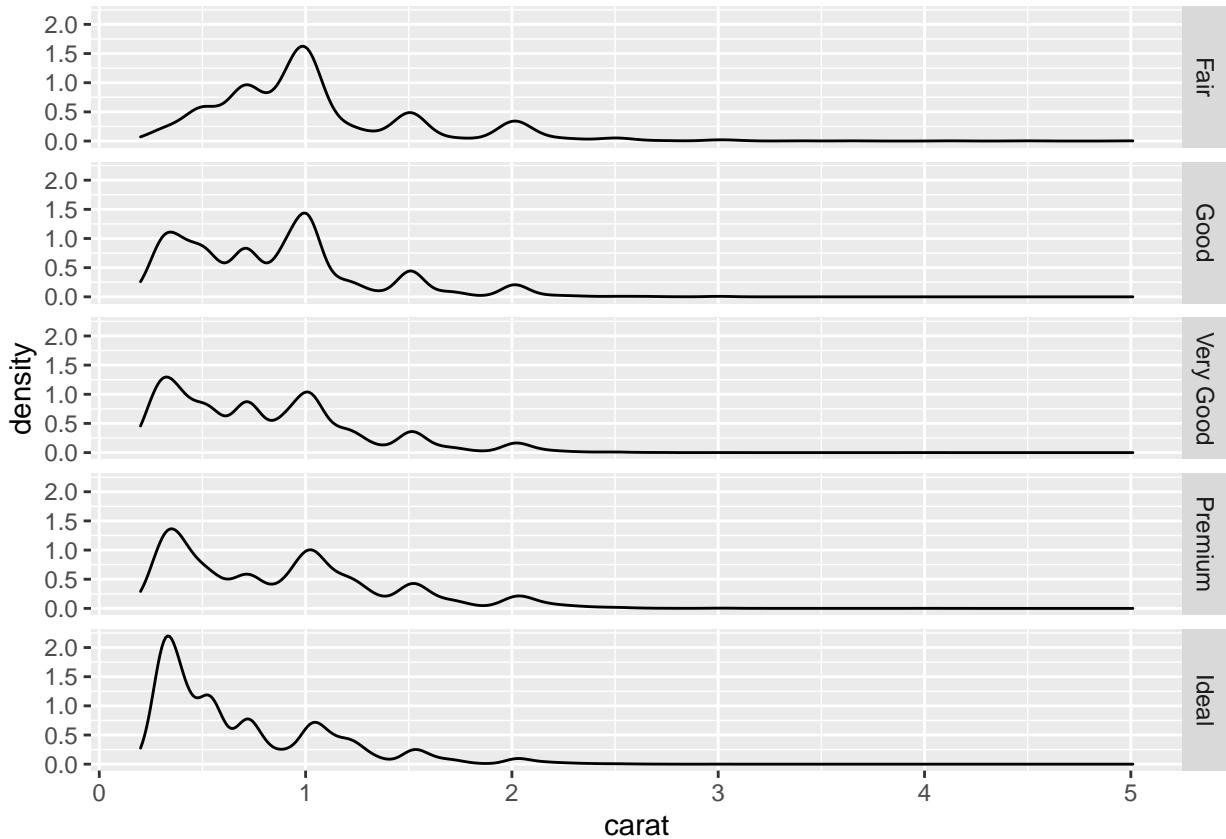


Exercice 1.12 (Challenge). Refaire la carte des températures du premier challenge (voir section 2.2.1) en utilisant `leaflet`. On utilisera la table construite dans le challenge 1 et la fonction `addPolygons`. On pourra également ajouter un popup qui permet de visualiser le nom du département ainsi que la température prévue lorsqu'on clique dessus.

On considère les données **diamonds**.

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

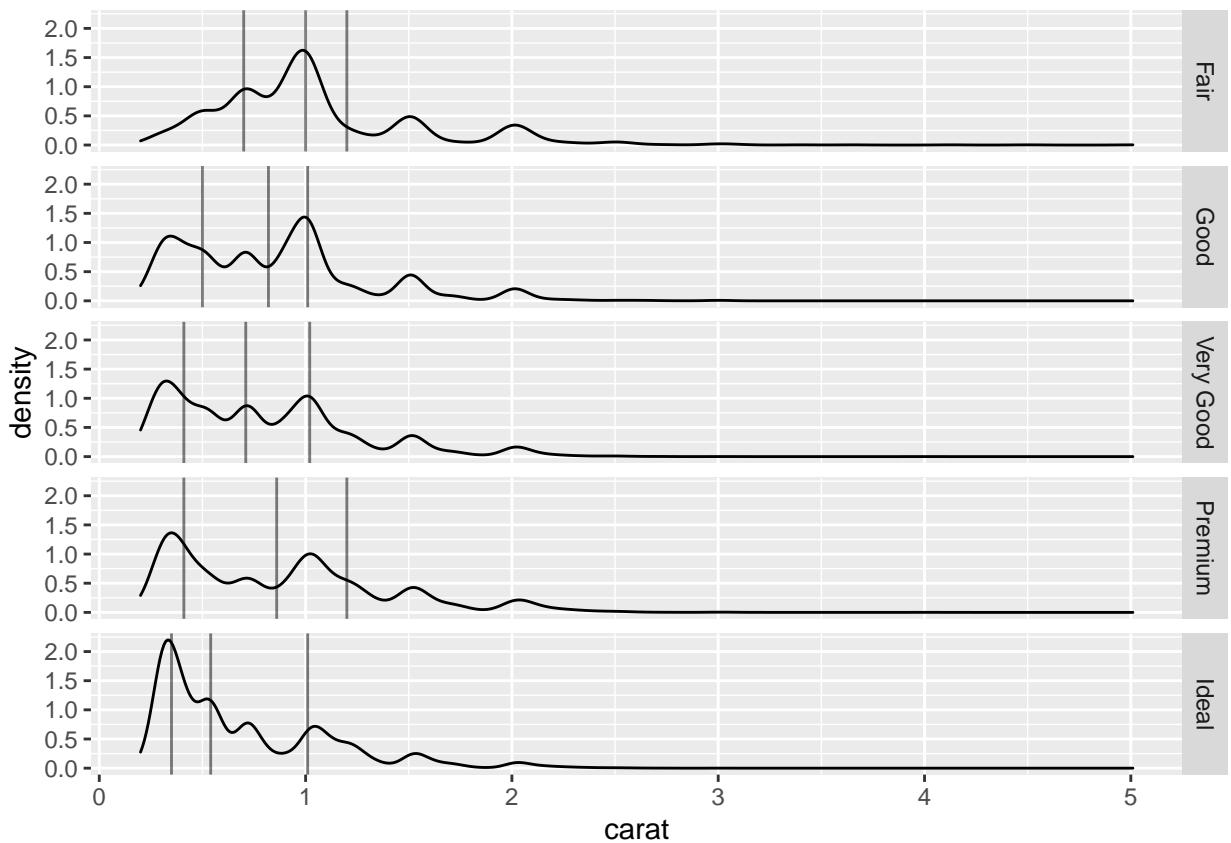




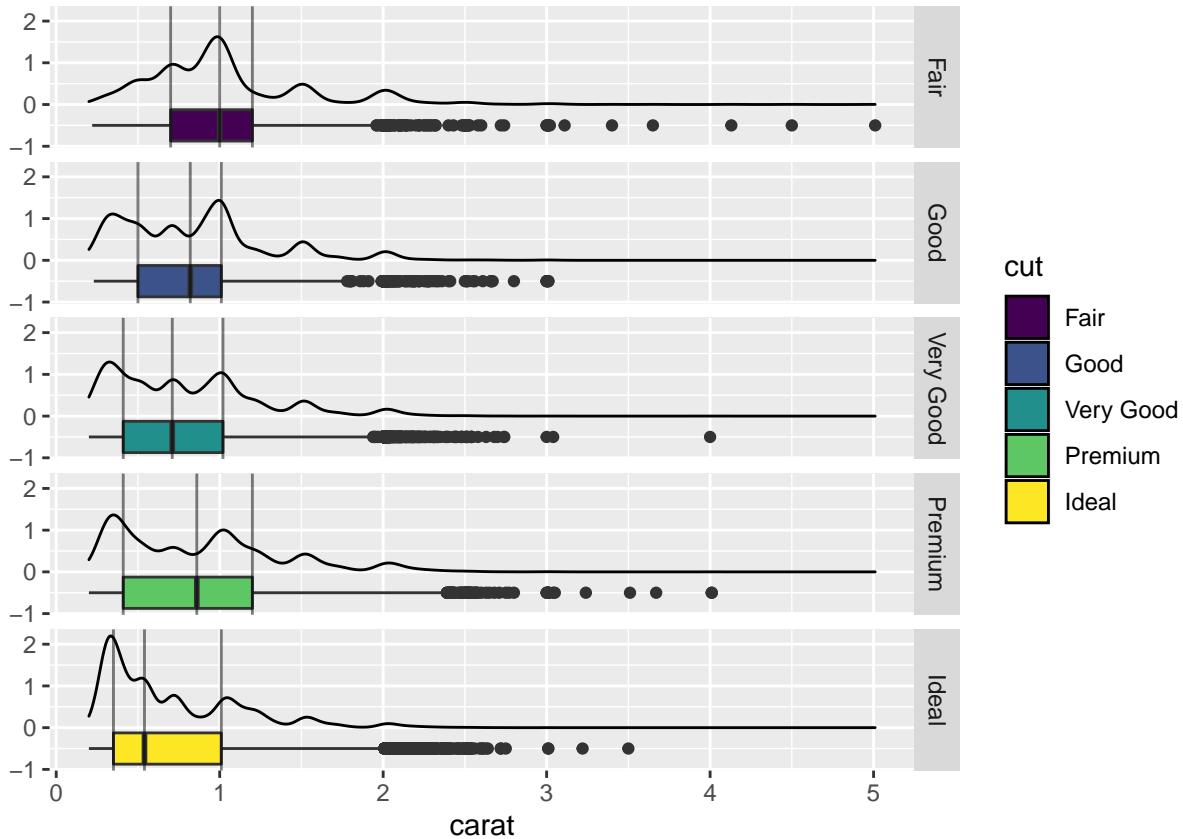
```
> ggplot(data=diamonds) + geom_boxplot(aes(x=cut,y=carat,fill=cut))
> ggplot(data=diamonds) + geom_boxplot(aes(x=cut,y=carat,fill=cut)) + coord_flip()
> ggplot(data=diamonds) + geom_density(aes(x=carat,y=..density..)) + facet_grid(cut~.)
```

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

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



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



```

> library(ggstance)
> ggplot(data=diamonds) +
+   geom_boxplot(data=diamonds, aes(y=-0.5, x=carat, fill=cut)) +
+   geom_density(aes(x=carat, y=..density..)) + facet_grid(cut~.) +
+   geom_vline(data=quantildf, aes(xintercept=quantiles), col=alpha("black", 1/2)) +
+   ylab("")
> knitr::opts_chunk$set(message=FALSE, warning=FALSE, cache=cache_carto)

```

2 Faire des cartes avec R

De nombreuses données comportent des informations de géolocalisation. Il est alors naturel d'utiliser des cartes pour les visualiser. On peut généralement s'intéresser à deux types de cartes :

- **statiques** : des cartes figées que l'on pourra exporter aux formats **pdf** ou **png** par exemple, ce type est généralement utilisé pour des rapports ;
- **dynamiques** ou **interactives** : des cartes que l'on pourra visualiser dans un navigateur et sur lesquelles on pourra zoomer ou obtenir des informations auxiliaires lorsqu'on clique sur certaines parties de la carte.

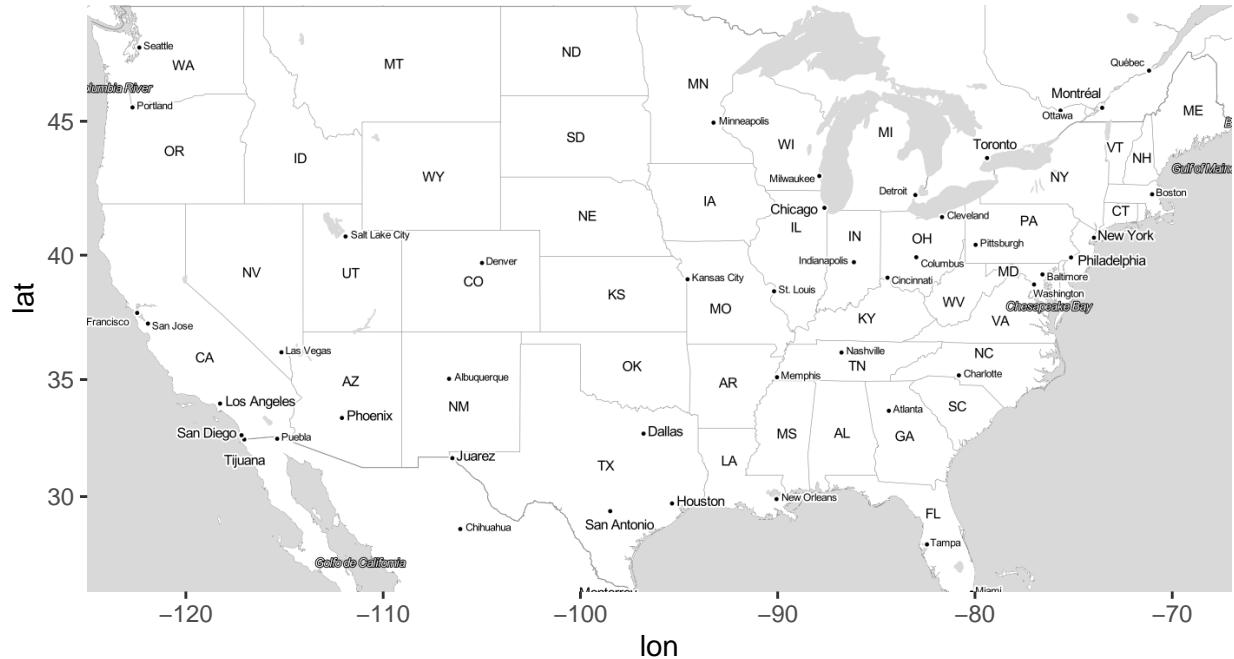
De nombreux packages **R** permettent de d'obtenir des cartes. Dans cette partie, on s'intéressera aux packages **ggmap** et **sf** pour les cartes statiques et **leaflet** pour les cartes interactives.

2.1 Le package ggmap

Nous montrons dans cette section comment récupérer des fonds de carte et ajouter quelques informations à l'aide de **ggmap**. Pour plus de détails sur ce package, on pourra consulter cet article pour plus de détails.

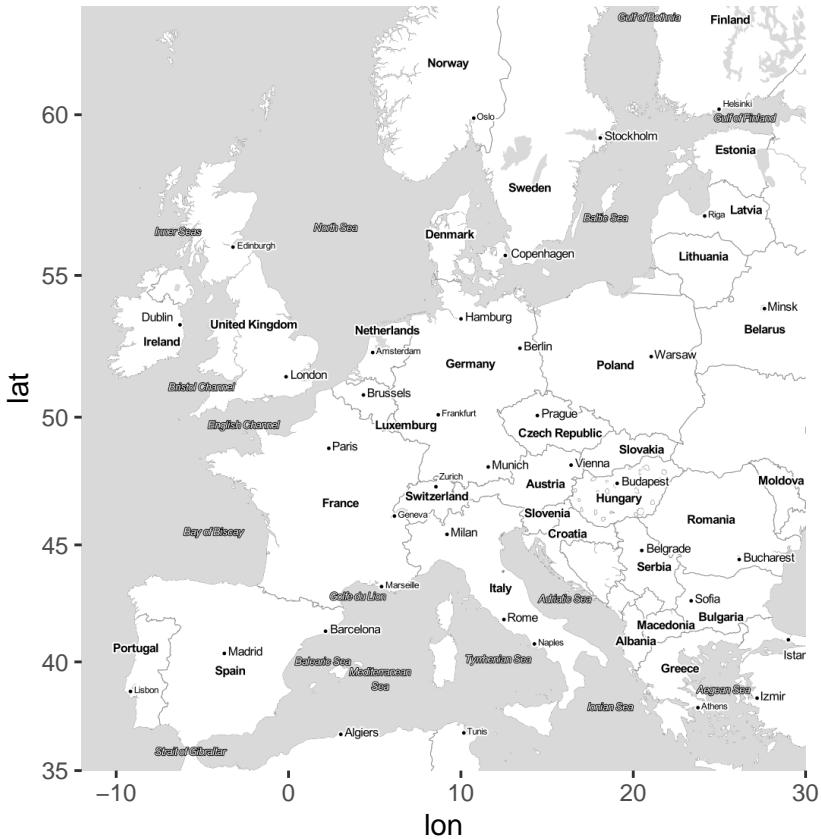
ggmap permet de récupérer facilement des fonds de carte. Par exemple :

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



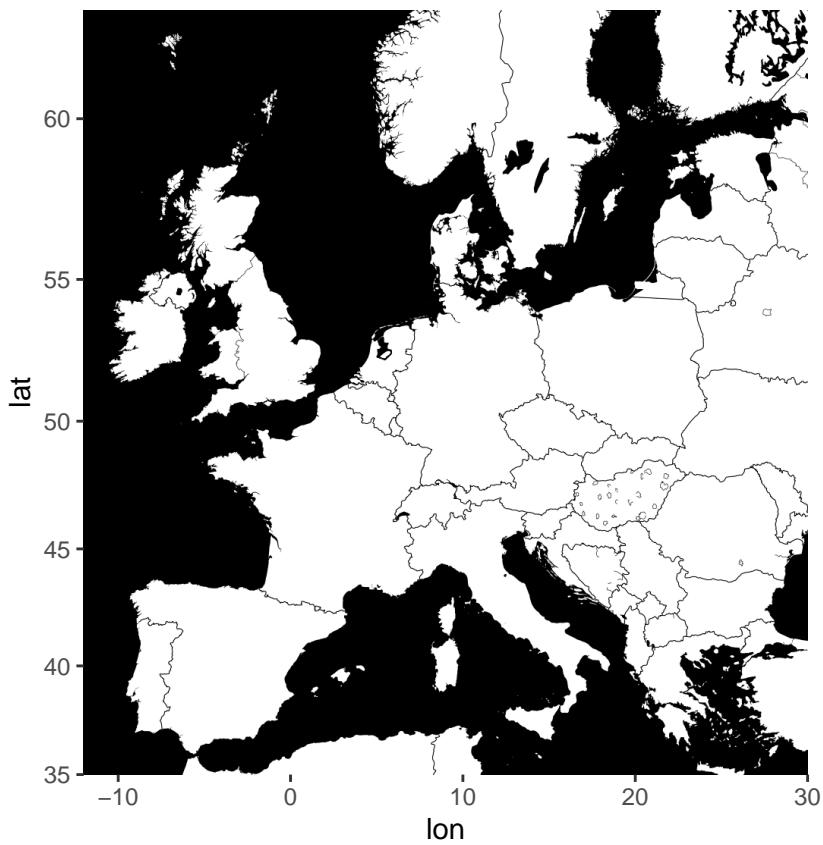
Pour l'Europe on fait

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



On peut également changer le fond de carte

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



Pour la france, on aura

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



La fonction `geocode` de `ggmap` qui permettait de récupérer des latitudes et longitudes nécessite désormais une **API**, ce qui constraint son utilisation. Nous proposons d'utiliser la fonction suivante :

```
> 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)
+ }
```

Cette fonction permet de récupérer les latitudes et longitudes de lieux à spécifier :

```
> 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

```

Exercice 2.1 (Populations des grandes villes de france).

1. Récupérer les latitudes et longitudes de Paris, Lyon et Marseille et représenter ces 3 villes sur une carte de la france.

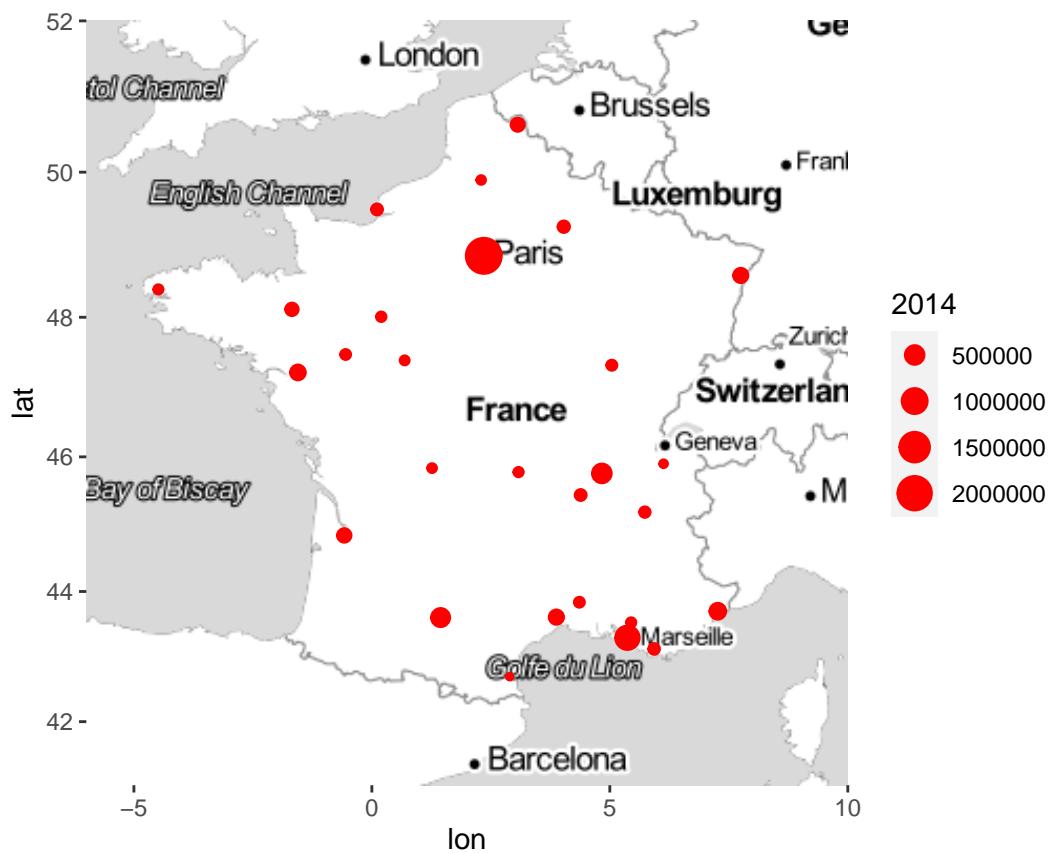
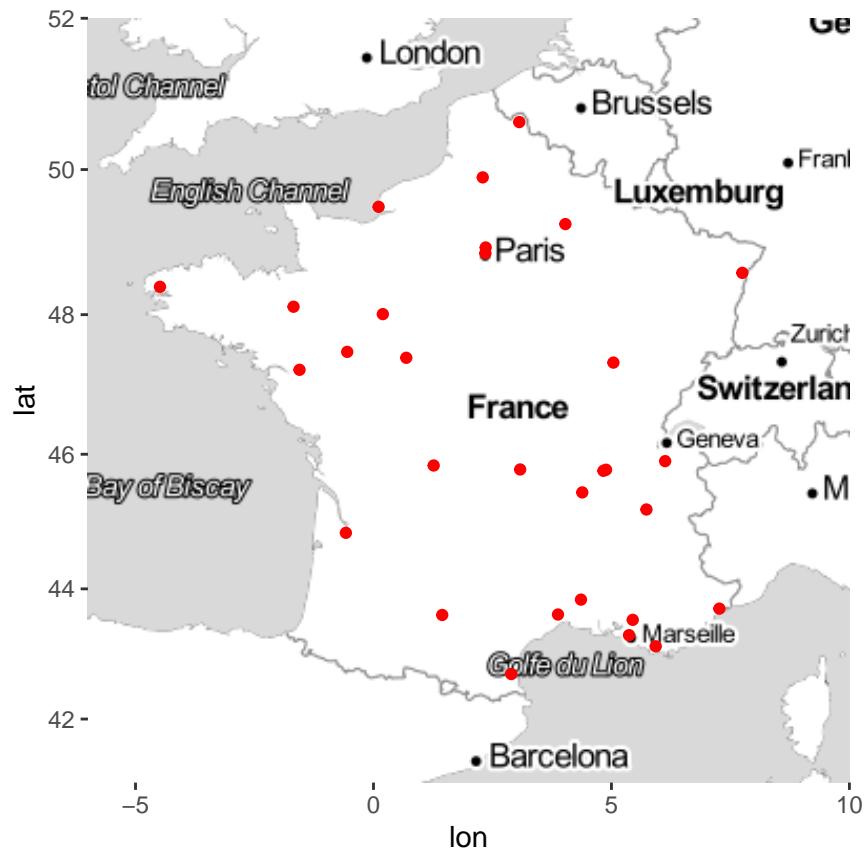
```

> V <- c("Paris", "Lyon", "Marseille")
> A <- mygeocode(V)
> A <- A %>% as_tibble() %>% mutate(Villes=V)
> fr <- c(left = -6, bottom = 41, right = 10, top = 52)
> fond <- get_stamenmap(fr, zoom = 5, "toner-lite")
> ggmap(fond)+geom_point(data=A,aes(x=lon,y=lat),color="red")

```



2. Le fichier **villes_fr.csv** contient les populations des 30 plus grandes villes de france. Représenter à l'aide d'un point les 30 plus grandes villes de france. On fera varier la taille du point en fonction de la population en 2014.



```

> df <- read_csv("villes_fr.csv")
> df$Commune <- as.character(df$Commune)
> df$Commune[10] <- "Lille"
> coord <- mygeocode(as.character(df$Commune)) %>% as_tibble()
> df1 <- bind_cols(df,coord)
> ggmap(fond)+geom_point(data=df1,aes(x=lon,y=lat),color="red")
> ggmap(fond)+geom_point(data=df1,aes(x=lon,y=lat,size=`2014`),color="red")

```

2.2 Cartes avec contours, le format shapefile

`ggmap` permet de récupérer facilement des fonds de cartes et de placer des points dessus avec la syntaxe `ggplot`. Cependant, de nombreuses fonctions de ce package nécessitent une API et il est difficile de définir des contours (frontières de pays, départements ou régions) avec `ggmap`. Nous proposons ici de présenter brièvement le package `sf` qui va nous permettre de créer des cartes “avancées”, en gérant les contours à l'aide d'objets particuliers mais aussi en prenant en compte différents systèmes de coordonnées. En effet, la terre n'est pas plate... mais une carte est souvent visualisée en 2D, il faut par conséquent réaliser des projections pour représenter des lieux définis par une coordonnée (comme la latitude et la longitude) sur une carte 2D. Ces projections sont généralement gérées par les packages qui permettent de faire de la cartographie comme `sf`. On pourra trouver de la documentation sur ce package aux url suivantes :

- <https://statnmap.com/fr/2018-07-14-initiation-a-la-cartographie-avec-sf-et-compagnie/>
- dans les **vignettes** sur la page du cran de ce package : <https://cran.r-project.org/web/packages/sf/index.html>

Ce package propose de définir un nouveau format `sf` adapté à la cartographie. Regardons par exemple l'objet `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 CRESS_ID BIR74 SID74
## 1 0.114     1.442 1825  1825    Ashe 37009 37009      5 1091     1
## 2 0.061     1.231 1827  1827  Alleghany 37005 37005      3 487      0
## 3 0.143     1.630 1828  1828    Surry 37171 37171     86 3188      5
## 4 0.070     2.968 1831  1831 Currituck 37053 37053     27 508      1
## 5 0.153     2.206 1832  1832 Northampton 37131 37131     66 1421      9
## 6 0.097     1.670 1833  1833 Hertford 37091 37091     46 1452      7
## 7 0.062     1.547 1834  1834   Camden 37029 37029     15 286      0
## 8 0.091     1.284 1835  1835    Gates 37073 37073     37 420      0
## 9 0.118     1.421 1836  1836   Warren 37185 37185     93 968      4
## 10 0.124    1.428 1837  1837 Stokes 37169 37169     85 1612      1
##   NWBIR74 BIR79 SID79 NWBIR79
##   geometry
## 1      10 1364    0    19 MULTIPOLYGON ((((-81.47276 3...
## 2      10 542     3    12 MULTIPOLYGON ((((-81.23989 3...
## 3      208 3616    6    260 MULTIPOLYGON ((((-80.45634 3...
## 4      123 830     2    145 MULTIPOLYGON ((((-76.00897 3...
## 5     1066 1606    3    1197 MULTIPOLYGON ((((-77.21767 3...

```

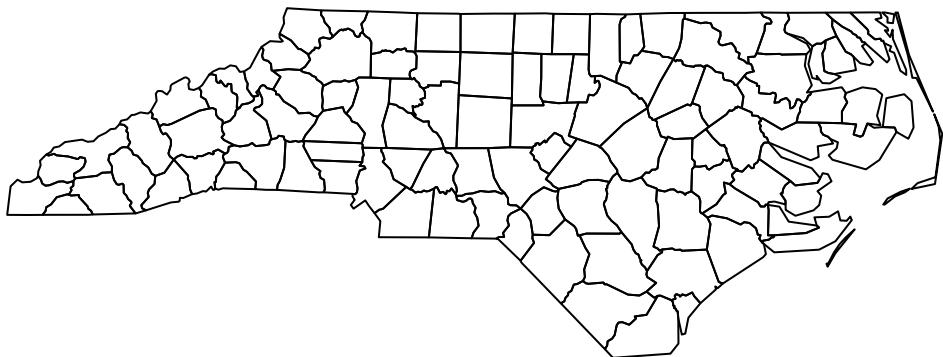
```

## 6    954 1838    5  1237 MULTIPOLYGON ((((-76.74506 3...
## 7    115 350     2   139 MULTIPOLYGON ((((-76.00897 3...
## 8    254 594     2   371 MULTIPOLYGON ((((-76.56251 3...
## 9    748 1190     2   844 MULTIPOLYGON ((((-78.30876 3...
## 10   160 2038    5   176 MULTIPOLYGON ((((-80.02567 3...

```

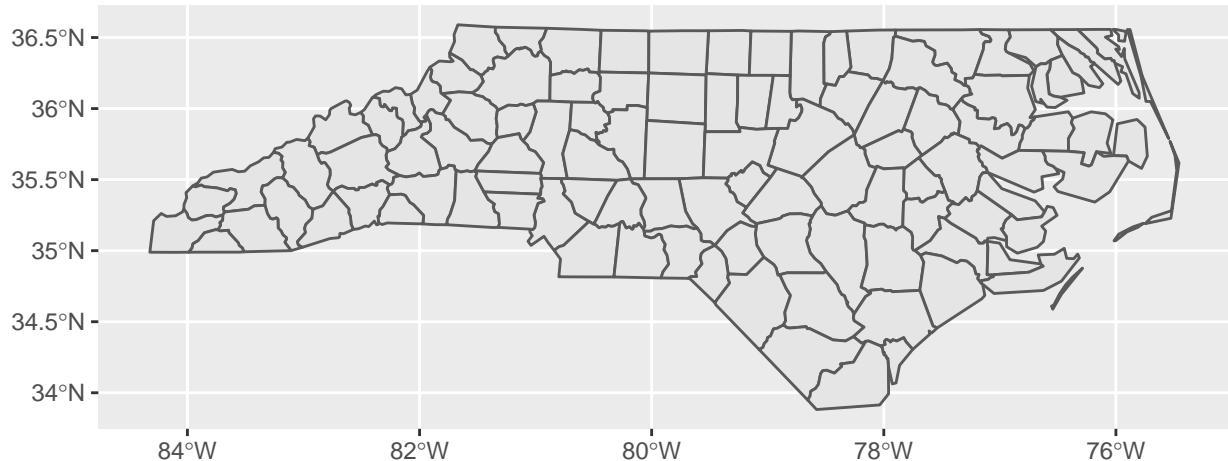
Ces données contiennent des informations sur les morts subites de nourrissons dans des villes de Caroline du Nord. On remarque que l'objet `nc` est au format `sf` et `data.frame`. On peut donc l'utiliser comme un `data.frame` classique. Le format `sf` permet l'ajout d'une colonne particulière (`geometry`) qui délimitera les villes à l'aide de polygones. Une fois l'objet obtenu au format `sf`, il est facile de visualiser la carte avec un `plot` classique

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



ou en utilisant le verbe `geom_sf` si on veut faire du `ggplot`

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

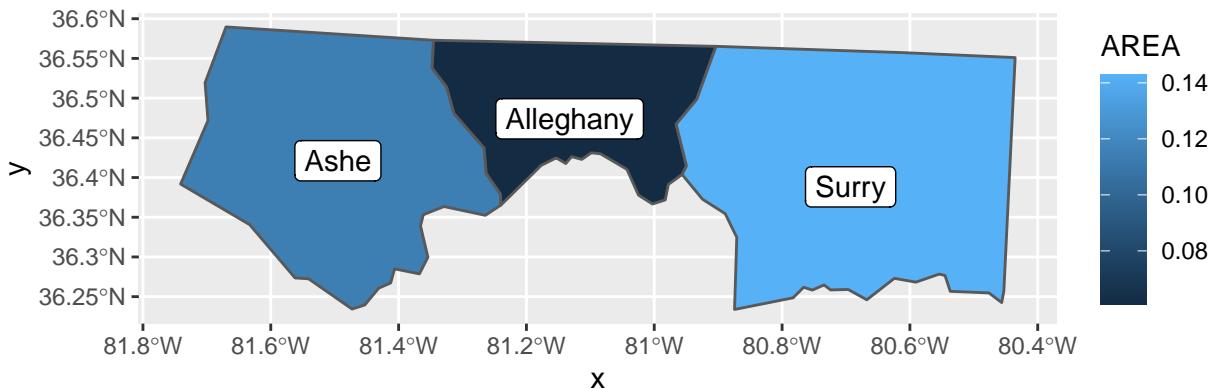


Il devient dès lors facile de colorier des villes et d'ajouter leurs noms :

```

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

```



La colonne `geometry` de `nc` est au format `MULTIPOLYGON`, elle permettra donc de délimiter les frontières des villes. Si maintenant on souhaite représenter une ville à l'aide d'un point défini par sa latitude et longitude, il va falloir modifier le format de cette colonne `geometry`. On peut le faire de la manière suivante :

1. On récupère les latitudes et longitudes de chaque ville :

```
> 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. On met ces coordonnées au format `MULTIPOINT`

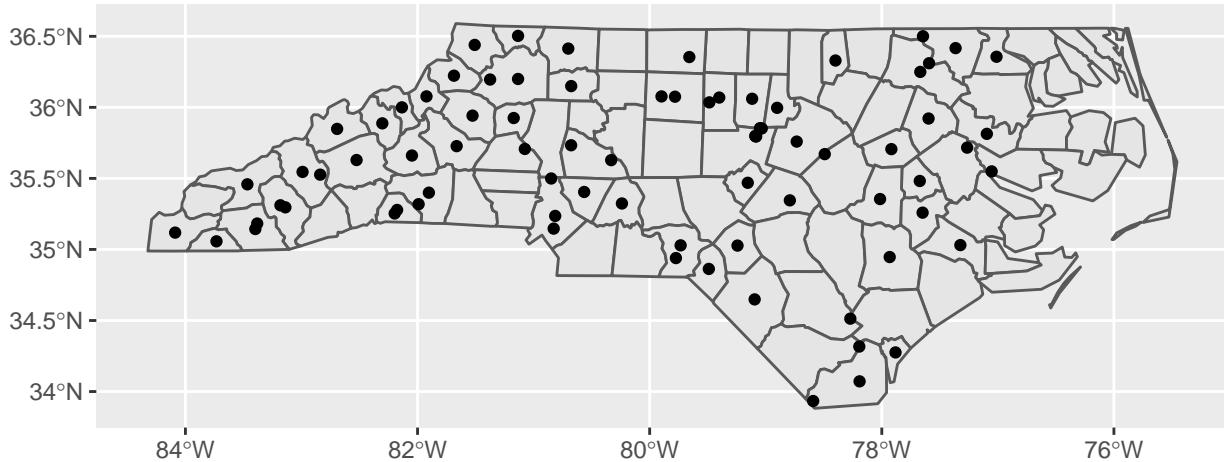
```
> coord.ville1.nc <- coord.ville.nc %>%
+   filter(lon<=-77 & lon>=-85 & lat>=33 & lat<=37) %>%
+   as.matrix() %>% st_multipoint() %>% st_geometry()
> coord.ville1.nc <- coord.ville.nc %>%
+   filter(lon<=-77 & lon>=-85 & lat>=33 & lat<=37) %>%
+   as.matrix() %>% st_multipoint() %>% st_geometry() %>% 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. On indique que ces coordonnées sont des latitudes et longitude et on ajoute la colonne aux données initiales

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

4. On peut enfin représenter la carte avec les frontières et les points :

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

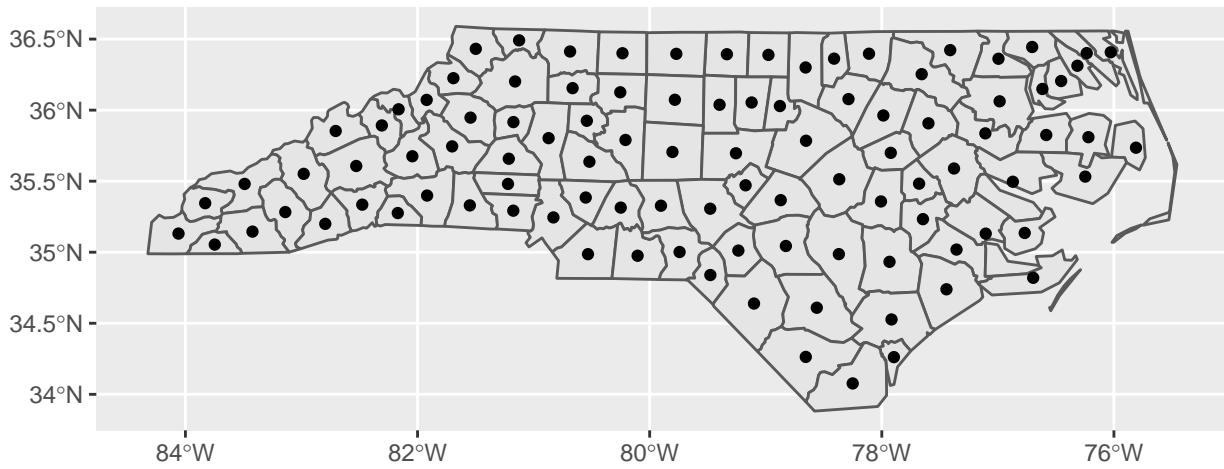


Le package `sf` possède également des fonctions très utiles pour traiter des données cartographiques, on peut citer par exemple :

- `st_distance` qui permet de calculer des distances entre coordonnées ;
- `st_centroid` pour calculer le centre d'une région.

On peut ainsi représenter les centres des villes délimitées par les polygones des données `nc` avec

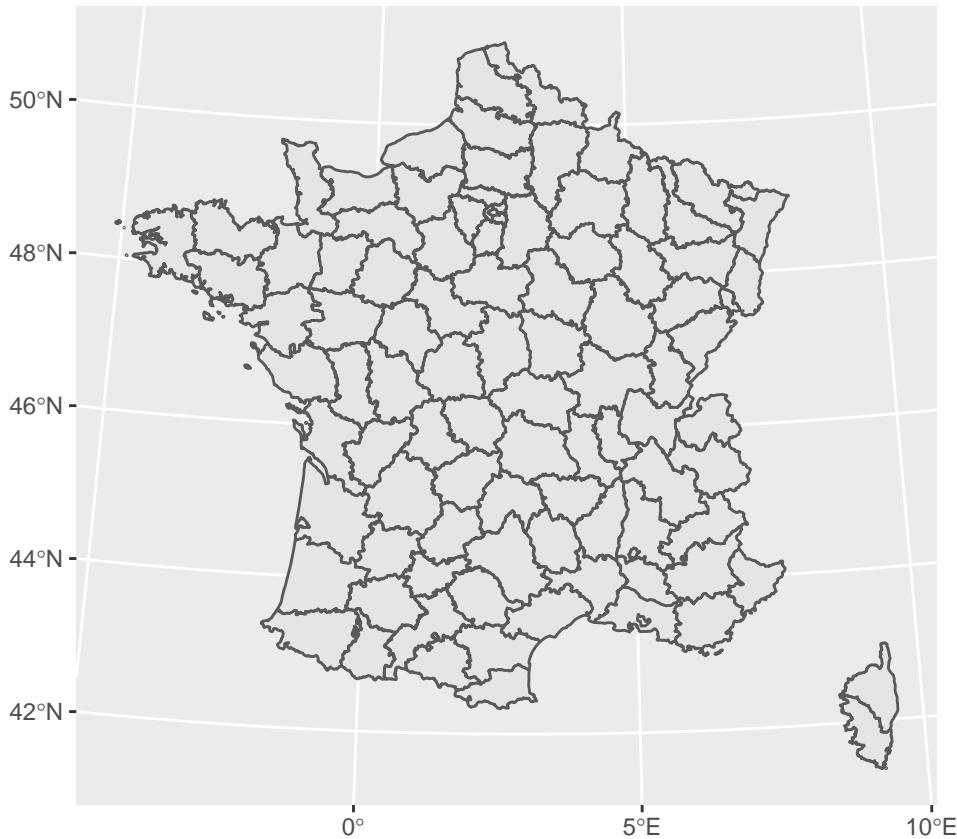
```
> nc2 <- nc %>% mutate(centre=st_centroid(nc)$geometry)
> ggplot(nc2)+geom_sf() +geom_sf(aes(geometry=centre))
```



Exercice 2.2 (Première carte avec sf).

Nous nous servons de la carte GEOFLAR proposée par l'Institut Géographique National pour récupérer un fond de carte contenant les frontières des départements français. Cette carte est disponible sur le site <http://professionnels.ign.fr/> au format `shapefile`, elle se trouve dans l'archive `dpt.zip`. Il faut décompresser pour reproduire la carte. Grâce au package `sf`, cette carte, contenue dans la série de fichiers `departement` du répertoire `dpt`, peut être importée dans un objet R :

```
> dpt <- read_sf("dpt")
> ggplot(dpt) + geom_sf()
```



Refaire la carte de l'exercice 2.1 sur ce fond de carte.

```

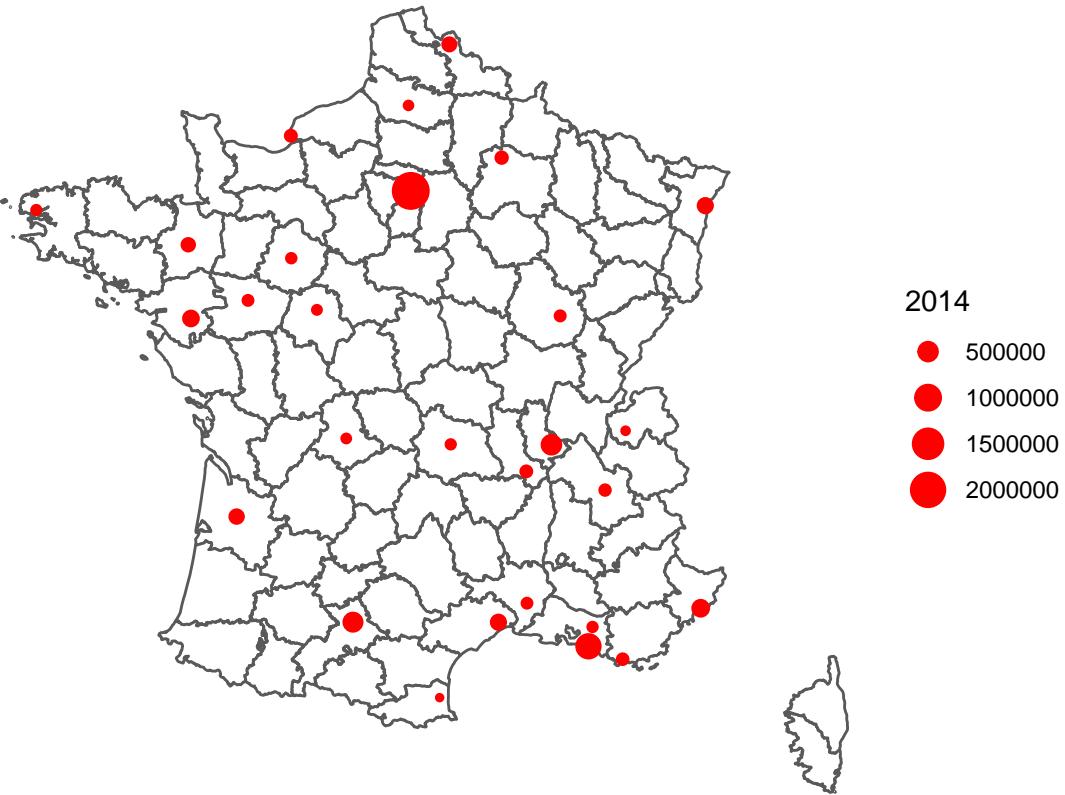
> coord.ville1 <- data.frame(df1[,14:15]) %>%
+   as.matrix() %>% st_multipoint() %>% st_geometry()
>
> coord.ville2 <- st_cast(coord.ville1, to = "POINT")

> coord.ville1
## Geometry set for 1 feature
## geometry type: MULTIPOINT
## dimension: XY
## bbox:           xmin: -4.486009 ymin: 42.69853 xmax: 7.750713 ymax: 50.63657
## CRS:            NA

> coord.ville2
## Geometry set for 30 features
## geometry type: POINT
## dimension: XY
## bbox:           xmin: -4.486009 ymin: 42.69853 xmax: 7.750713 ymax: 50.63657
## CRS:            NA
## First 5 geometries:

> st_geometry(df1) <- coord.ville2
> st_crs(df1) <- 4326
> ggplot(dpt)+geom_sf(fill="white")+
+   geom_sf(data=df1,aes(size=`2014`),color="red")+theme_void()

```



Exercice 2.3 (Visualisation de taux de chômage avec sf).

Nous souhaitons visualiser graphiquement les différences de taux de chômage par département entre deux années. Pour cela, nous disposons de chaque taux mesuré aux premiers trimestres des années 2006 et 2011 (variables TCHOMB1T06, TCHOMB1T11) qui se trouvent dans le jeu de données **tauxchomage.csv**

1. Importer le jeu de données.

```
> chomage <- read_delim("tauxchomage.csv", delim=";")
```

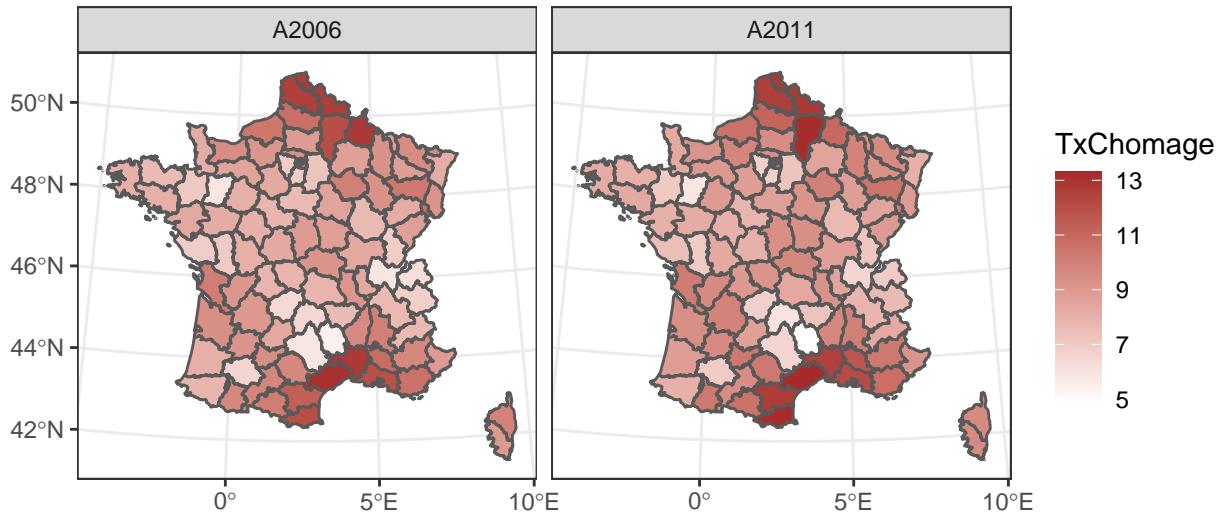
2. Faire la jointure de cette table avec celle des départements. On pourra utiliser **inner_join**.

```
> dpt <- read_sf("dpt")
> dpt2 <- inner_join(dpt, chomage, by="CODE_DEPT")
```

3. Comparer les taux de chômage en 2006 et 2011 (on le fera une carte avec les taux en 2006 et une autre avec les taux en 2011).

```
> dpt3 <- dpt2 %>% select(A2006=TCHOMB1T06, A2011=TCHOMB1T11, geometry) %>%
+   gather("Annee", "TxChomage", -geometry)

> ggplot(dpt3) + aes(fill = TxChomage) + geom_sf() +
+   facet_wrap(~Annee, nrow = 1) +
+   scale_fill_gradient(low="white", high="brown") + theme_bw()
```



2.2.1 Challenge 1 : carte des températures avec ggmap

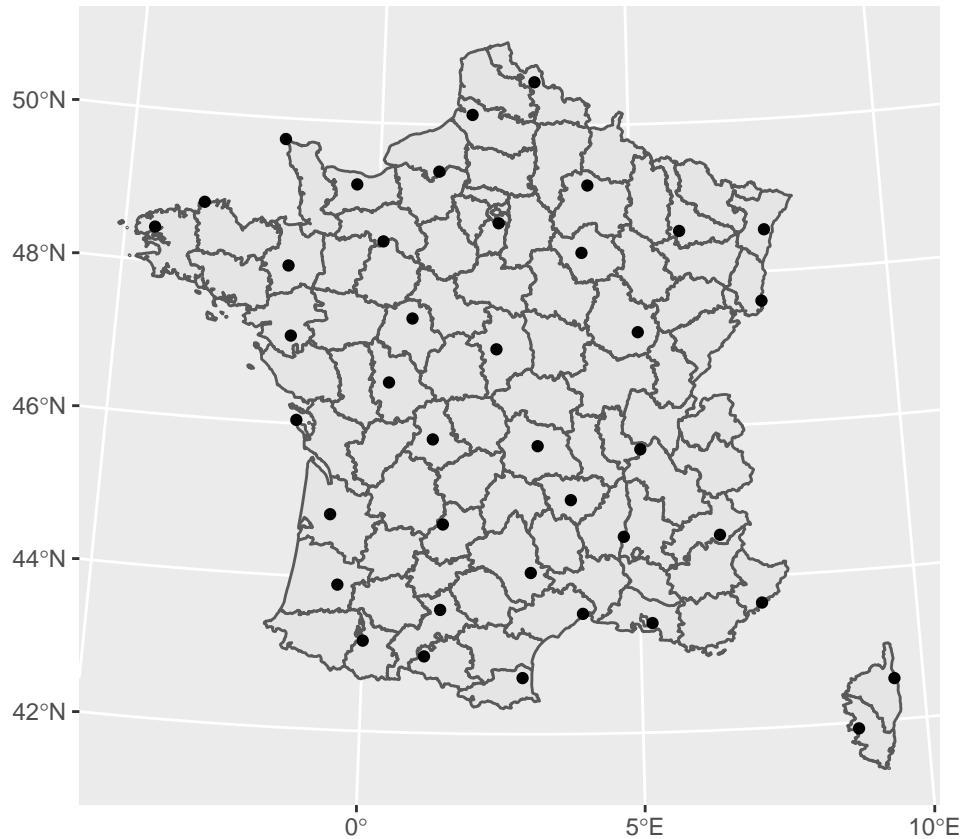
On souhaite ici faire une carte permettant de visualiser les température en France à un moment donné. Les données se trouvent sur le site des données publiques de meteo france. On peut notamment récupérer les températures observées dans certaines stations en france ainsi que la géolocalisation de ces stations.

1. Importer les 2 bases nécessaires. On pourra les lire directment sur le site. Convertir les degrés Kelvin en degrés Celsius et faire la jointure de ces bases.

```
> donnees <- read_delim("https://donneespubliques.meteofrance.fr/donnees_libres/Txt/Synop/synop.20200522")
> station <- read_delim("https://donneespubliques.meteofrance.fr/donnees_libres/Txt/Synop/postesSynop.csv")
> donnees$t <- donnees$t-273.15 #on passe en degrés celcius
> temp <- donnees %>% select(numer_sta,t)
> names(temp)[1] <- c("ID")
> D <- inner_join(temp, station, by = c("ID"))
```

2. Eliminer les station d'outre mer (on pourra conserver uniquement les stations qui ont une longitude entre -20 et 25). On appellera ce tableau **station1**. Visualiser les stations sur la carte contenant les frontières des départements français.

```
> station1 <- D %>% filter(Longitude<25 & Longitude>-20) %>% na.omit()
> station4326 <- st_multipoint(as.matrix(station1[,5:4])) %>% st_geometry()
> st_crs(station4326) <- 4326
> ggplot(dpt) + geom_sf() + geom_sf(data=station4326)
```



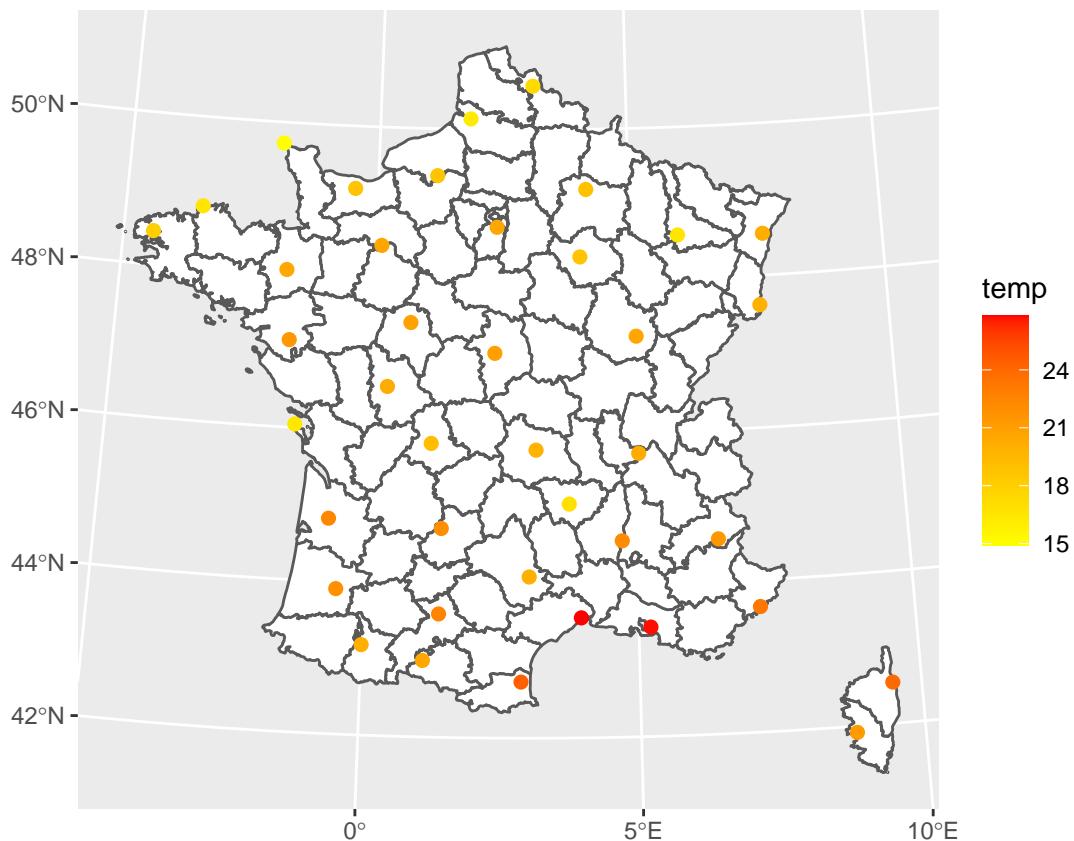
3. Créer un dataframe au format `sf` qui contient les températures des stations ainsi que leurs coordonnées dans la colonne `geometry`. On pourra commencer avec

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

> df <- data.frame(temp=station1$t)
> st_geometry(df) <- station2
```

4. Représenter les stations sur une carte de france. On pourra mettre un point de couleur différente en fonction de la température.

```
> ggplot(dpt) + geom_sf(fill="white") +
+   geom_sf(data=df, aes(color=temp), size=2) +
+   scale_color_continuous(low="yellow", high="red")
```



5. On obtient les coordonnées des centroïdes des départements à l'aide de

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

On déduit les distances entre ces centroïdes et les stations avec (**df** étant la table **sf** obtenue à la question 3).

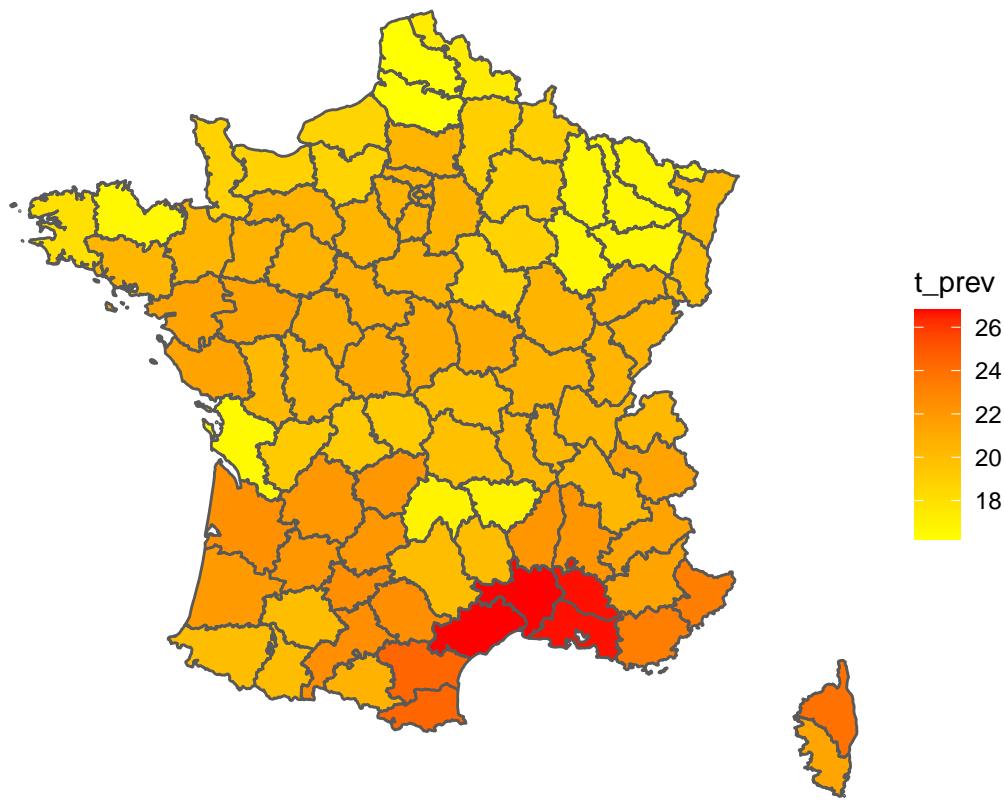
```
> DD <- st_distance(df,centro)
```

Prédire la température de chaque département à l'aide de la règle du 1 plus proche voisin.

```
> NN <- apply(DD,2,order)[1,]
> t_prev <- station1[NN,2]
```

6. Colorier les départements en fonction de la température prédite dans le département. On pourra faire varier le dégradé de couleur du jaune (pour les faibles températures) au rouge (pour les fortes).

```
> dpt1 <- dpt %>% mutate(t_prev=as.matrix(t_prev))
> ggplot(dpt1) + geom_sf(aes(fill=t_prev)) +
+   scale_fill_continuous(low="yellow",high="red") + theme_void()
```



2.3 Cartes interactives avec leaflet

Leaflet est un package permettant de faire de la *cartographie interactive*. On pourra consulter un descriptif synthétique ici. Le principe est similaire à ce qui a été présenté précédemment : les cartes sont construites à partie de couches qui se superposent. Un fond de carte s'obtient avec les fonctions `leaflet` et `addTiles`

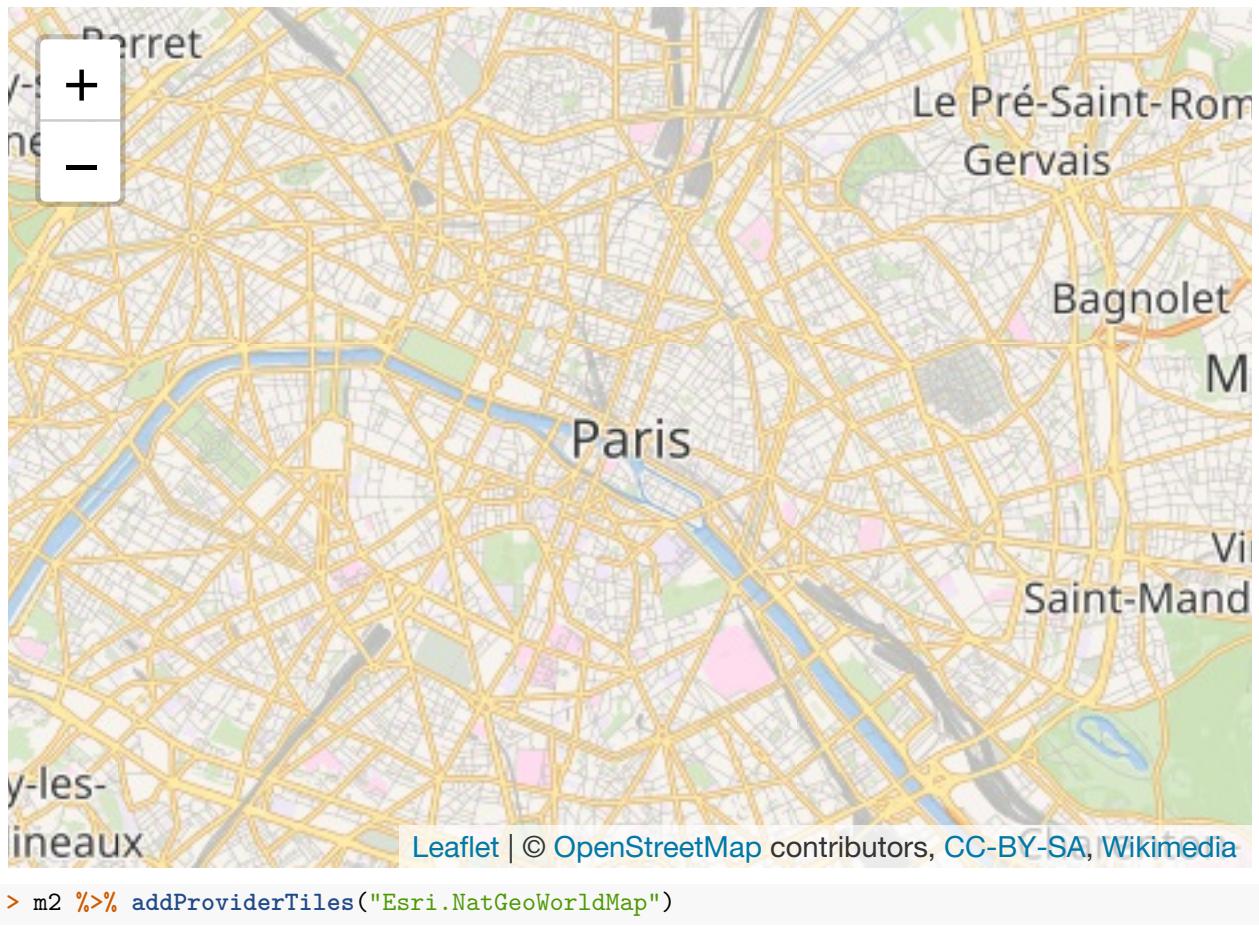
```
> library(leaflet)
> leaflet() %>% addTiles()
```



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

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





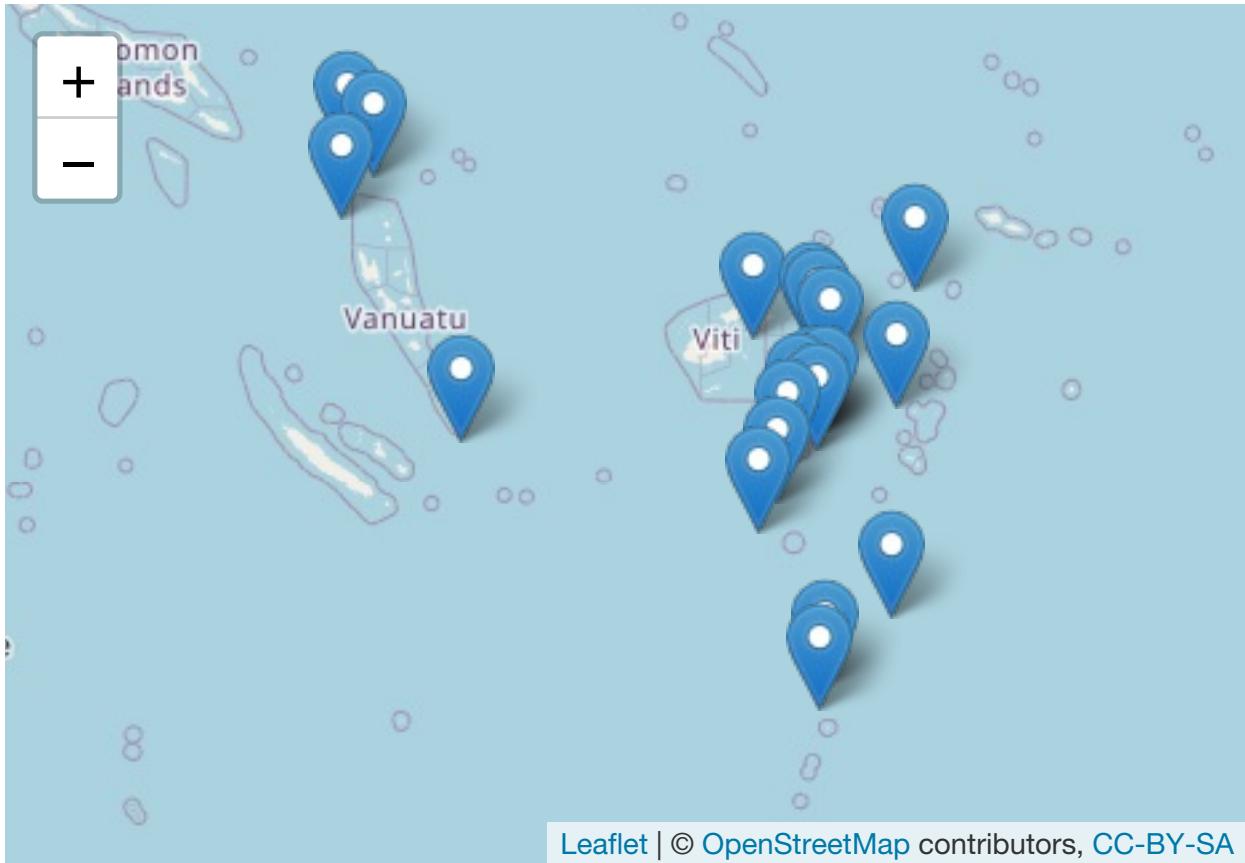


```
> m2 %>%
+   addProviderTiles("Stamen.Watercolor") %>%
+   addProviderTiles("Stamen.TonerHybrid")
```



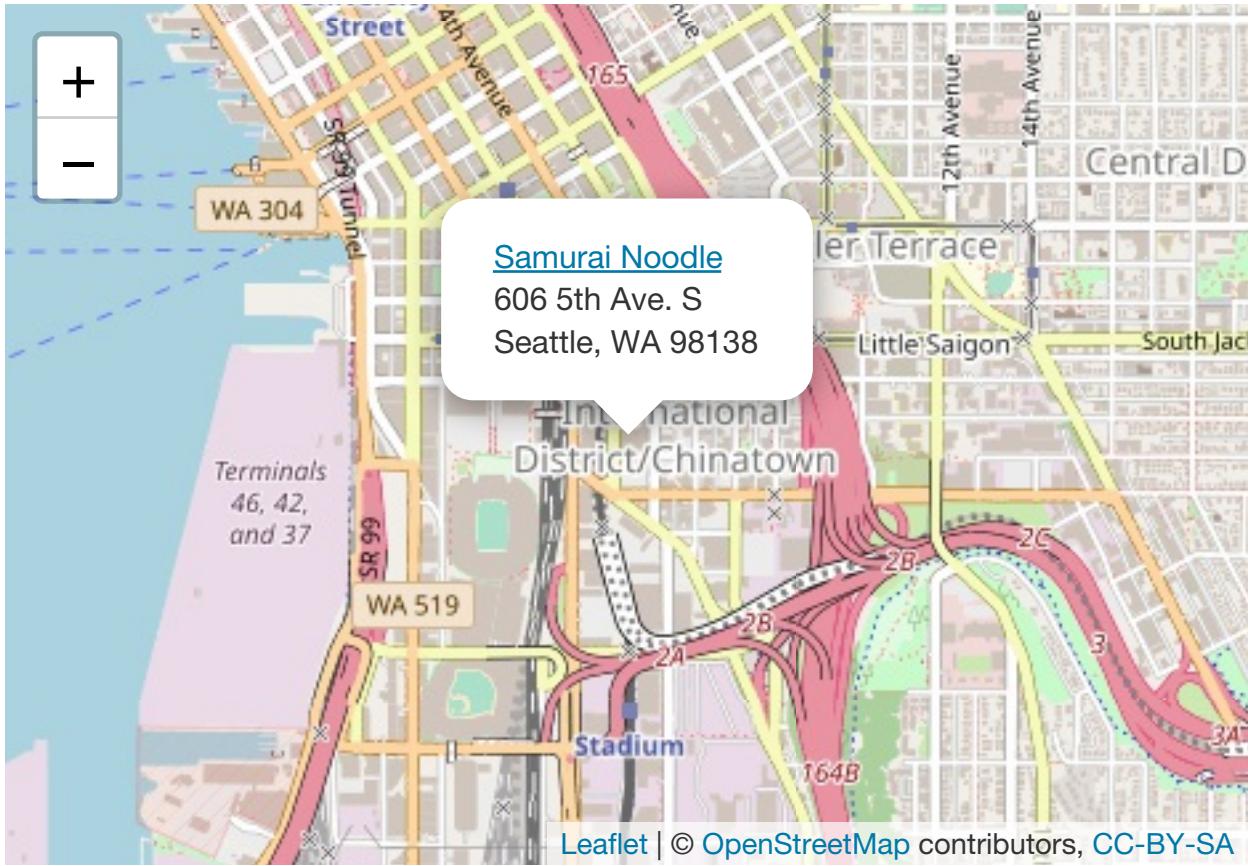
Il est fréquemment utile de repérer des lieux sur une carte à l'aide de symboles. On pourra effectuer cela à l'aide des fonctions `addMarkers` et `addCircles`...

```
> 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 :

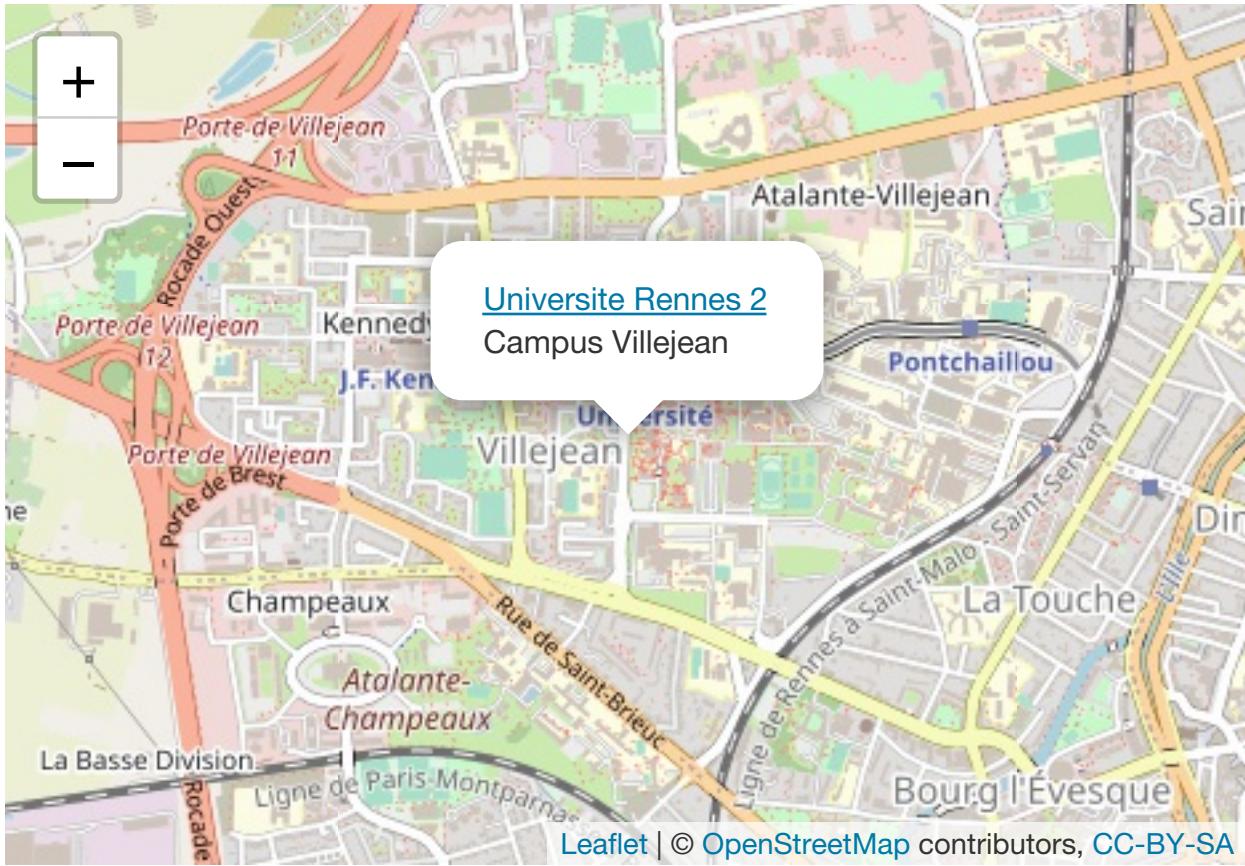
```
> content <- paste(sep = "<br/>",
+   "<b><a href='http://www.samurainoodle.com'>Samurai Noodle</a></b>",
+   "606 5th Ave. S",
+   "Seattle, WA 98138"
+ )
>
> leaflet() %>% addTiles() %>%
+   addPopups(-122.327298, 47.597131, content,
+   options = popupOptions(closeButton = FALSE)
+ )
```



Exercice 2.4 (Popup avec leaflet).

Placer un **popup** localisant l'Université Rennes 2 (Campus Villejean). On ajoutera un lien renvoyant sur le site de l'Université.

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



2.3.1 Challenge 2 : 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 vlistats 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 `url` dans `read.csv`.

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

2. Décrire les variables du jeu de données.

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

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

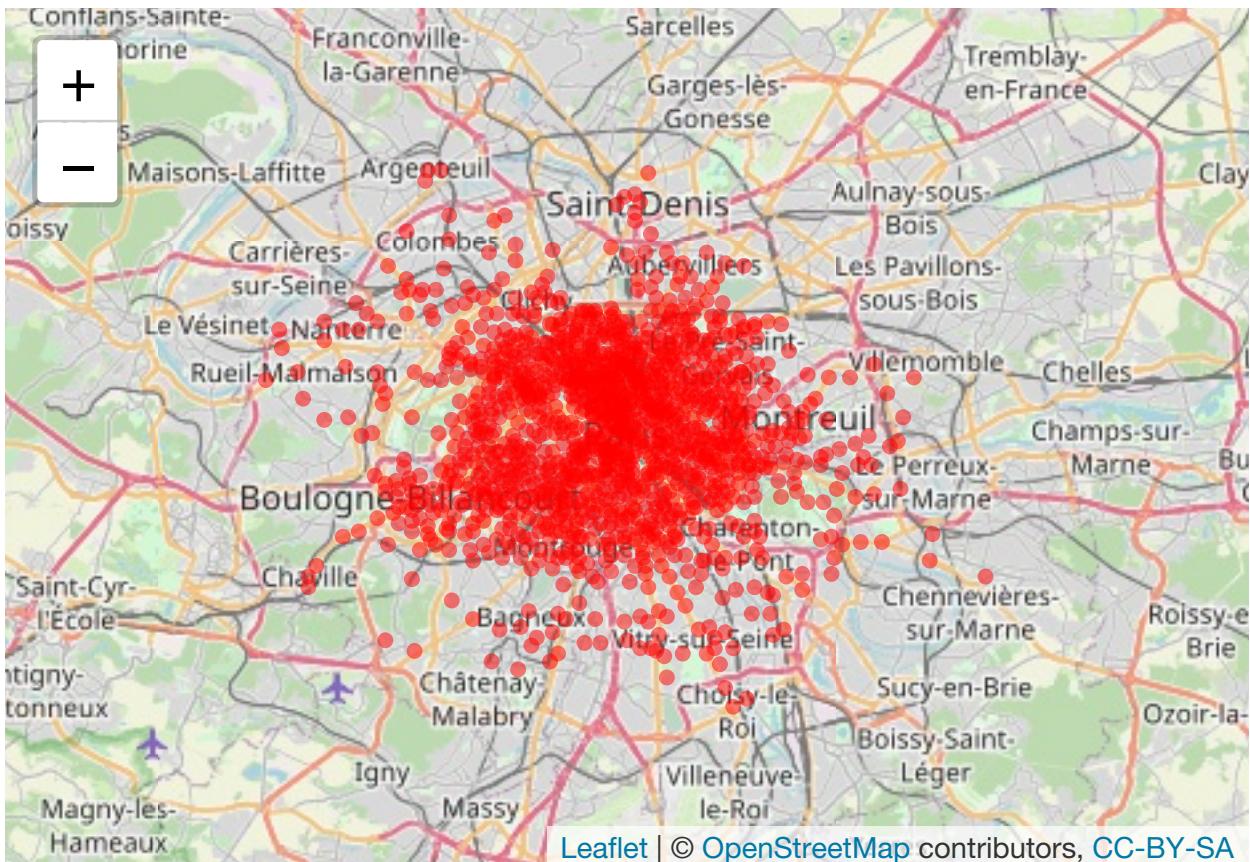
```
> sta.Paris1 <- sta.Paris %>% separate(`Coordonnées géographiques`,
+                                         into=c("lat","lon"),sep=",") %>%
+   mutate(lat=as.numeric(lat),lon=as.numeric(lon))
```

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

```

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

```



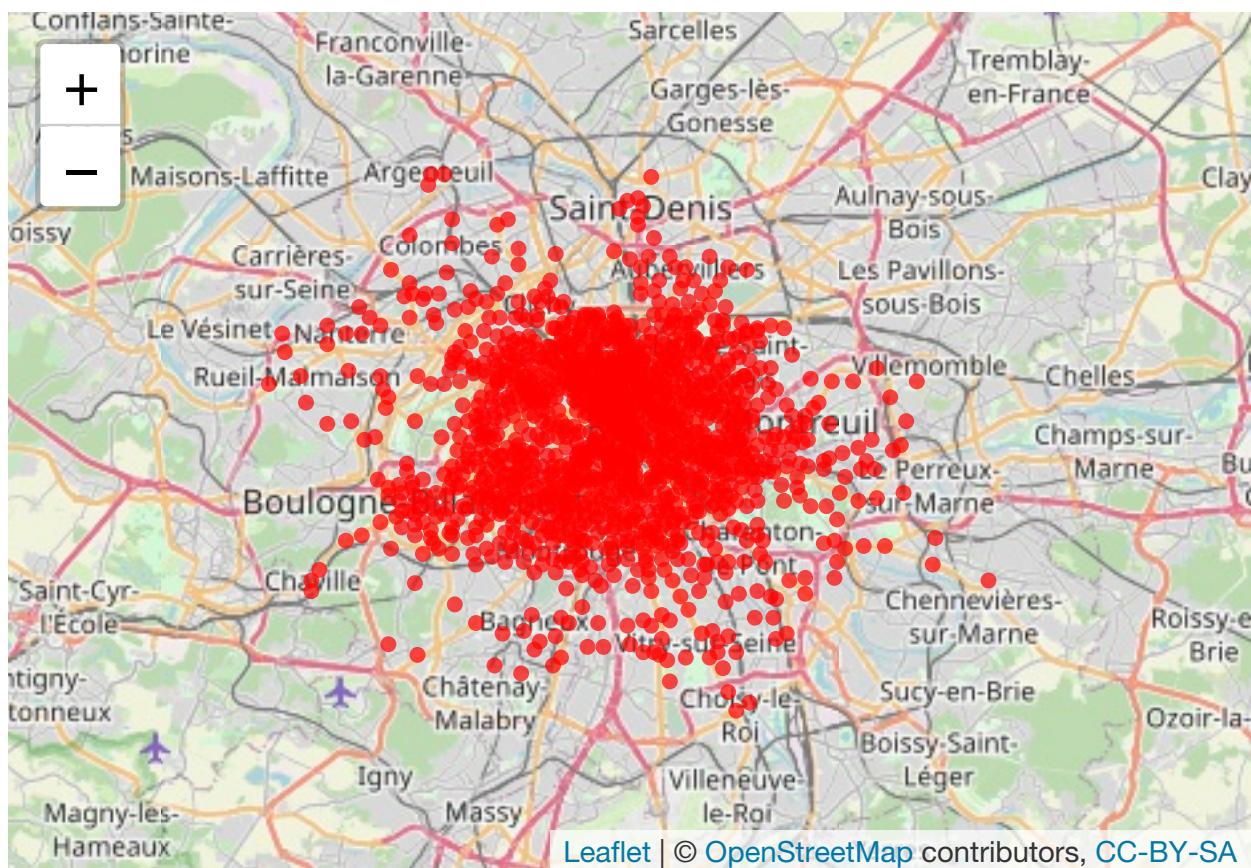
- Ajouter un popup qui permet de connaître le nombre de vélos disponibles (électriques+mécanique) quand on clique sur la station (on pourra utiliser l'option **popup** dans la fonction **addCircleMarkers**).

```

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

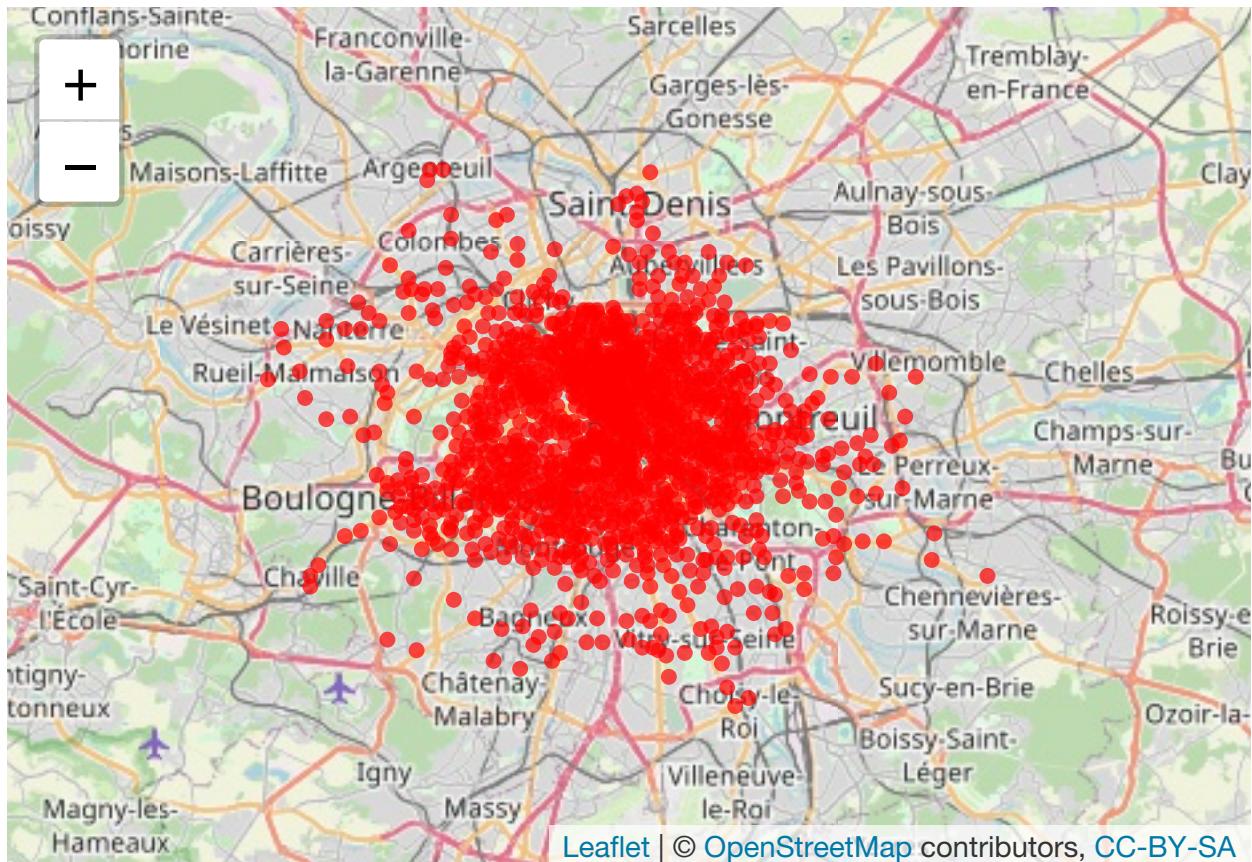
```

```
>  
> map.velib2
```



6. Ajouter le nom de la station dans le popup.

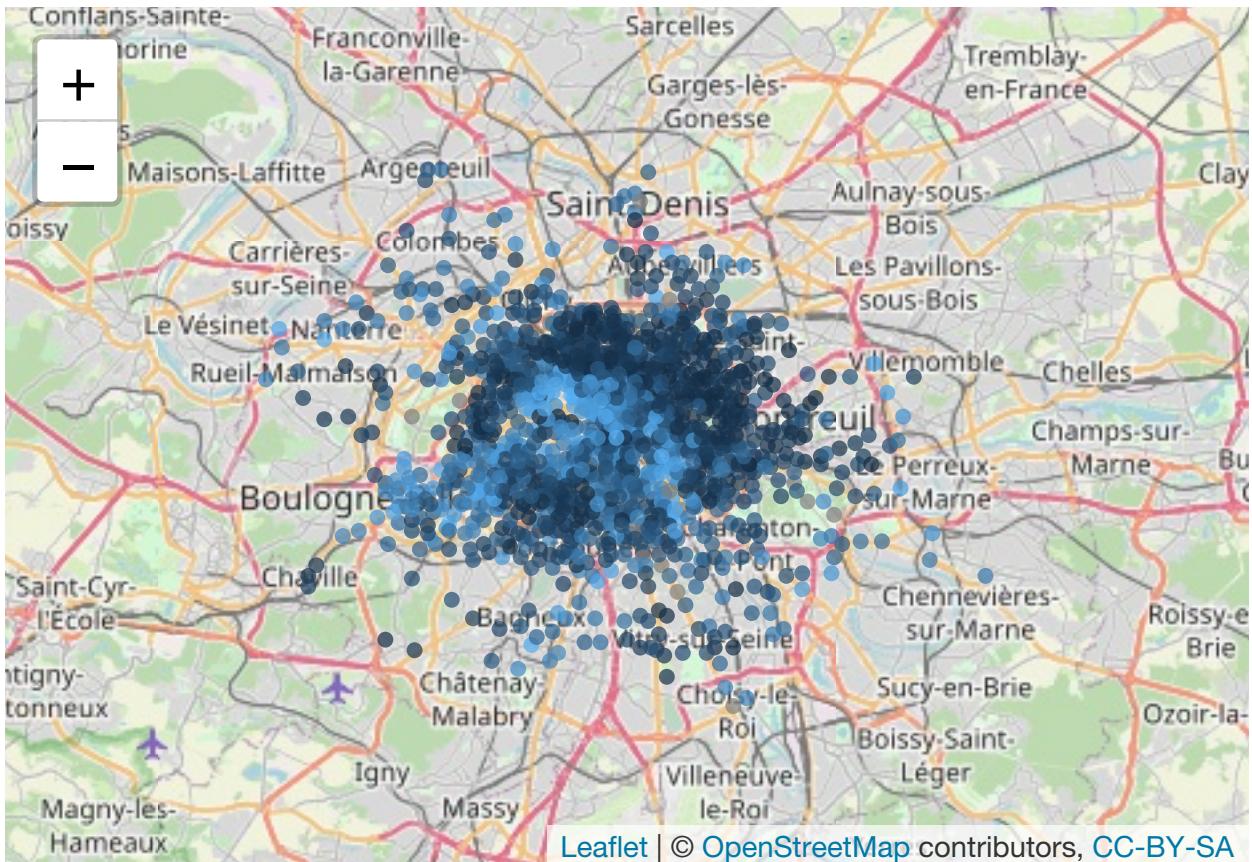
```
> map.velib3 <- leaflet(data = sta.Paris1) %>%  
+   addTiles() %>%  
+   addCircleMarkers(~ lon, ~ lat, radius=3, stroke = FALSE,  
+                   fillOpacity = 0.7, color="red",  
+                   popup = ~ paste(as.character(`Nom station`), ", Vélos dispos :",  
+                     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



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

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

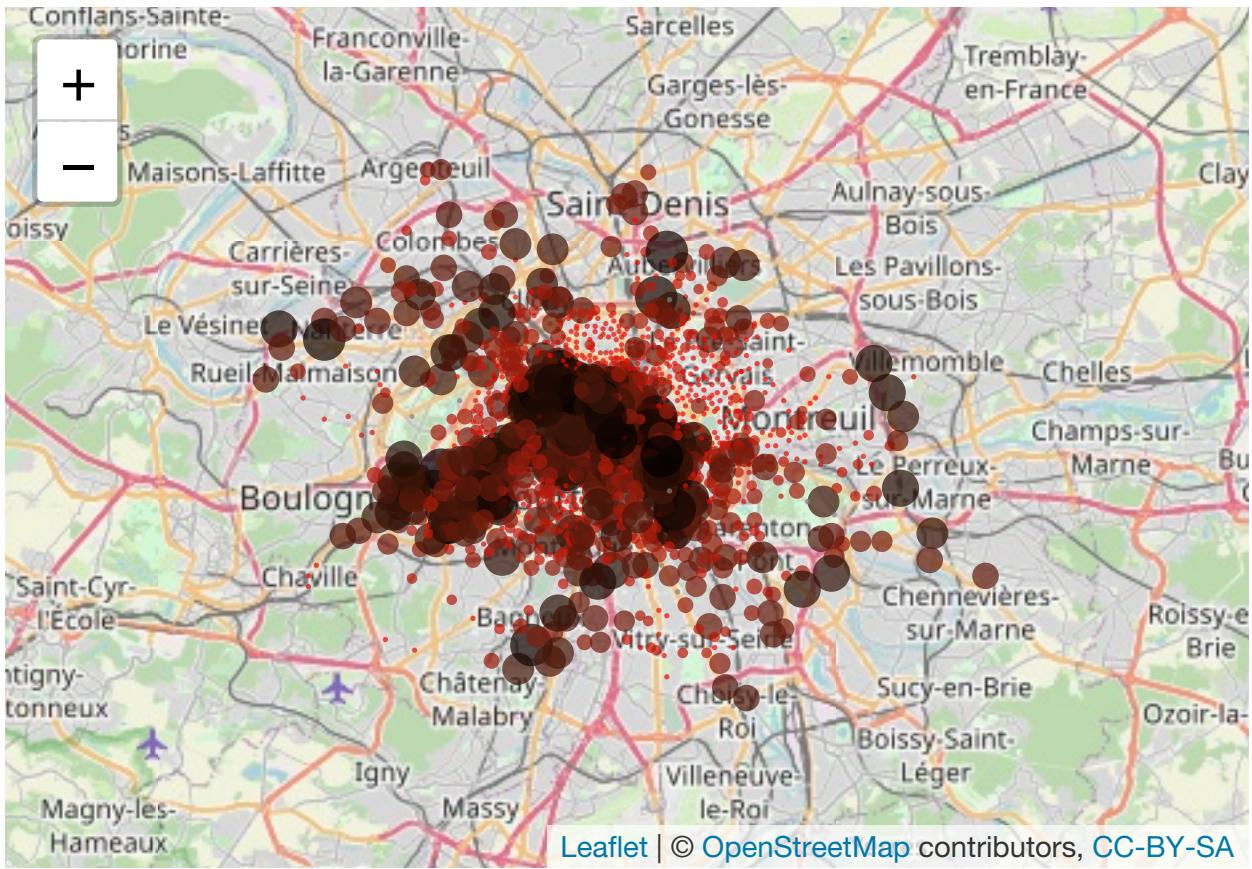
> map.velib4 <- leaflet(data = sta.Paris1) %>%
+   addTiles() %>%
+   addCircleMarkers(~ lon, ~ lat, radius=3,stroke = FALSE, fillOpacity = 0.7,
+                   color=~ColorPal1(`Nombre total vélos disponibles`/
+                                     `Capacité de la station`),
+                  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



```

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

```

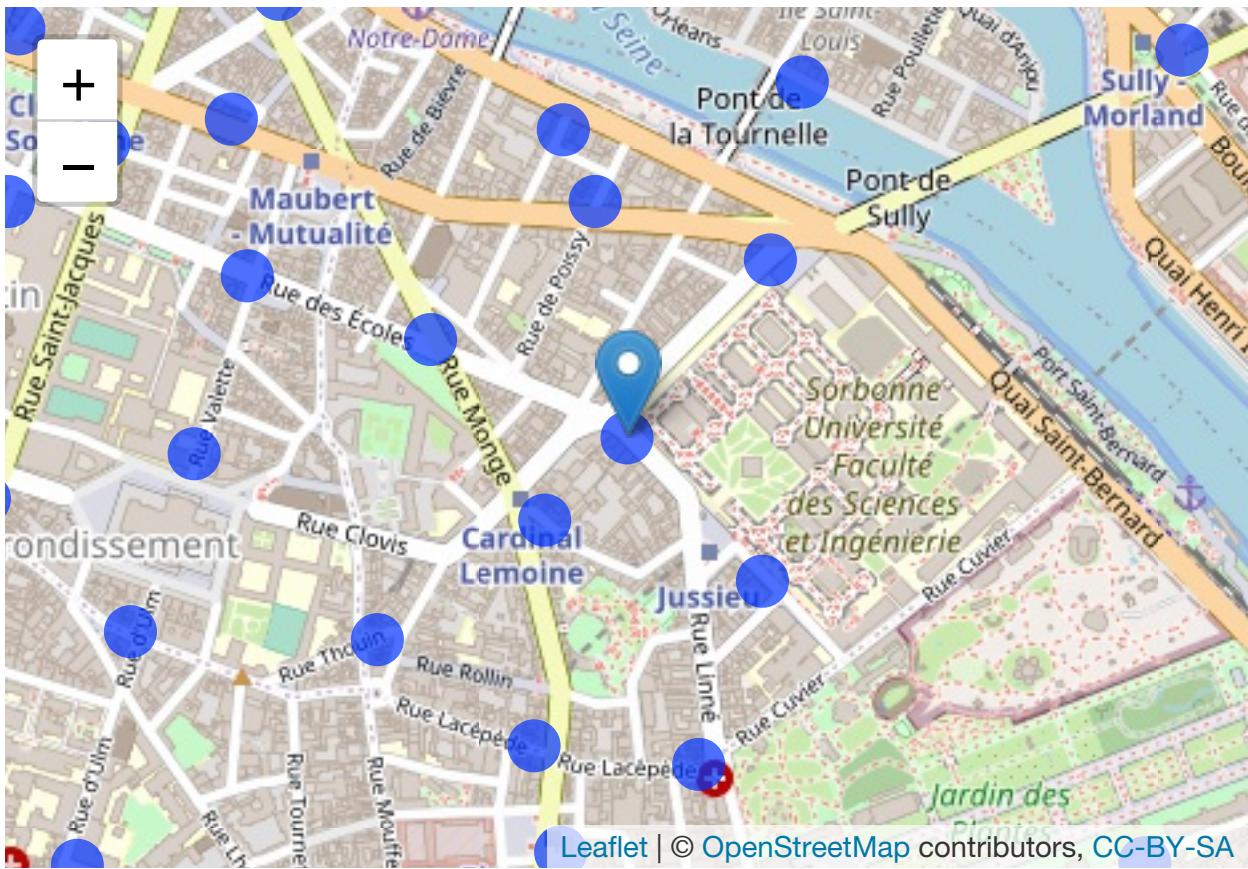


8. Créer une fonction `local.station` qui permette de visualiser quelques stations autour d'une station choisie.

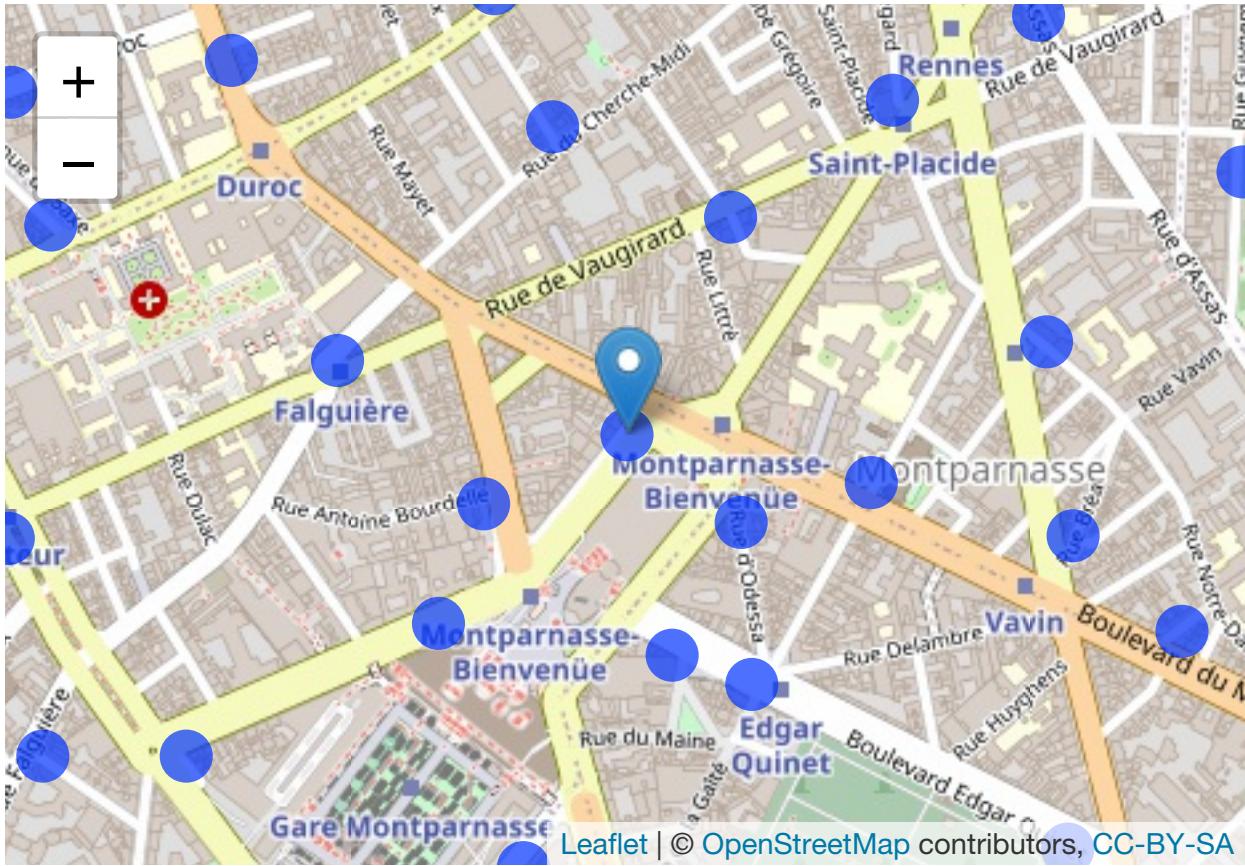
```
> nom.station <- "Jussieu - Fossés Saint-Bernard"
> local.station <- function(nom.station){
+   df <- sta.Paris1 %>% filter(`Nom station` == nom.station)
+   leaflet(data = sta.Paris1) %>% setView(lng=df$lon, lat=df$lat, zoom=15) %>%
+     addTiles() %>%
+     addCircleMarkers(~ lon, ~ lat, stroke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

La fonction devra par exemple renvoyer

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



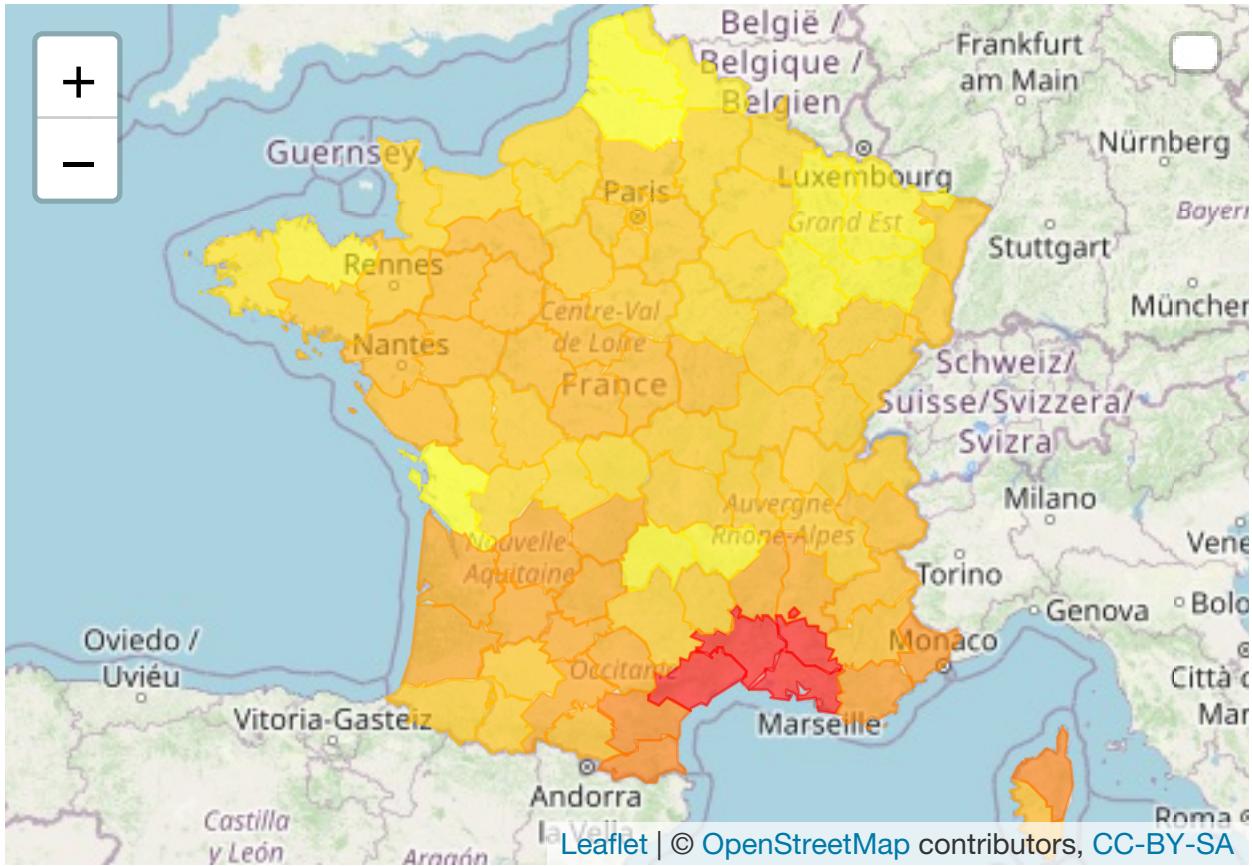
> local.station("Gare Montparnasse - Arrivée")



2.3.2 Carte des températures avec leaflet

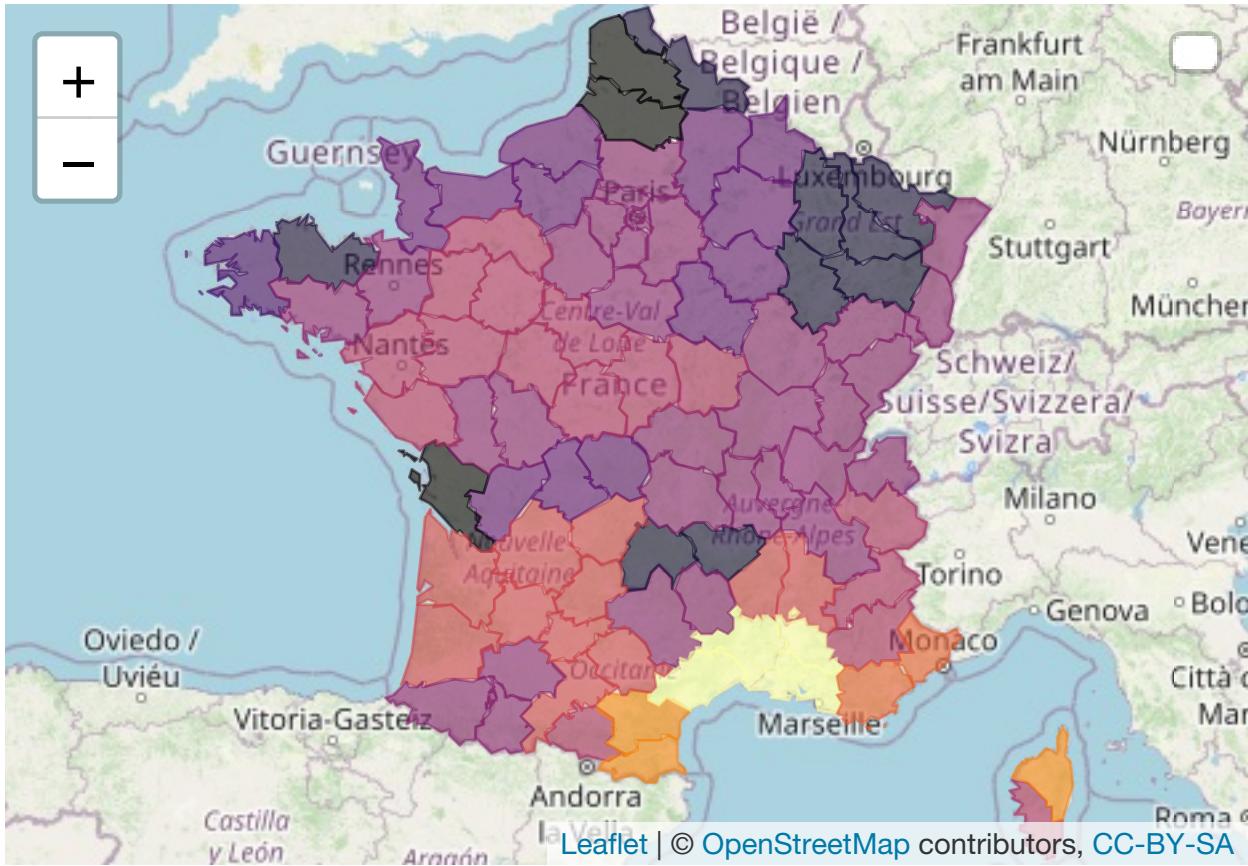
Exercice 2.5 (Challenge). Refaire la carte des températures du premier challenge (voir section 2.2.1) en utilisant **leaflet**. On utilisera la table construite dans le challenge 1 et la fonction **addPolygons**. On pourra également ajouter un popup qui permet de visualiser le nom du département ainsi que la température prévue lorsqu'on clique dessus.

```
> dpt2 <- st_transform(dpt1, crs = 4326)
> dpt2$t_prev <- round(dpt2$t_prev)
> pal <- colorNumeric(scales::seq_gradient_pal(low = "yellow", high = "red",
+ space = "Lab"), domain = dpt2$t_prev)
> m <- leaflet() %>% addTiles() %>%
+   addPolygons(data = dpt2, color=~pal(t_prev), fillOpacity = 0.6,
+               stroke = TRUE, weight=1,
+               popup=~paste(as.character(NOM_DEPT),as.character(t_prev),sep=" : ")) %>%
+   addLayersControl(options=layersControlOptions(collapsed = FALSE))
> m
```



ou avec une autre palette de couleur

```
> pal1 <- colorNumeric(palette = c("inferno"), domain = dpt2$t_prev)
> m1 <- leaflet() %>% addTiles() %>%
+   addPolygons(data = dpt2, color=~pal1(t_prev), fillOpacity = 0.6,
+               stroke = TRUE, weight=1,
+               popup=~paste(as.character(NOM_DEPT), as.character(t_prev), sep=" : ")) %>%
+   addLayersControl(options=layersControlOptions(collapsed = FALSE))
> m1
```



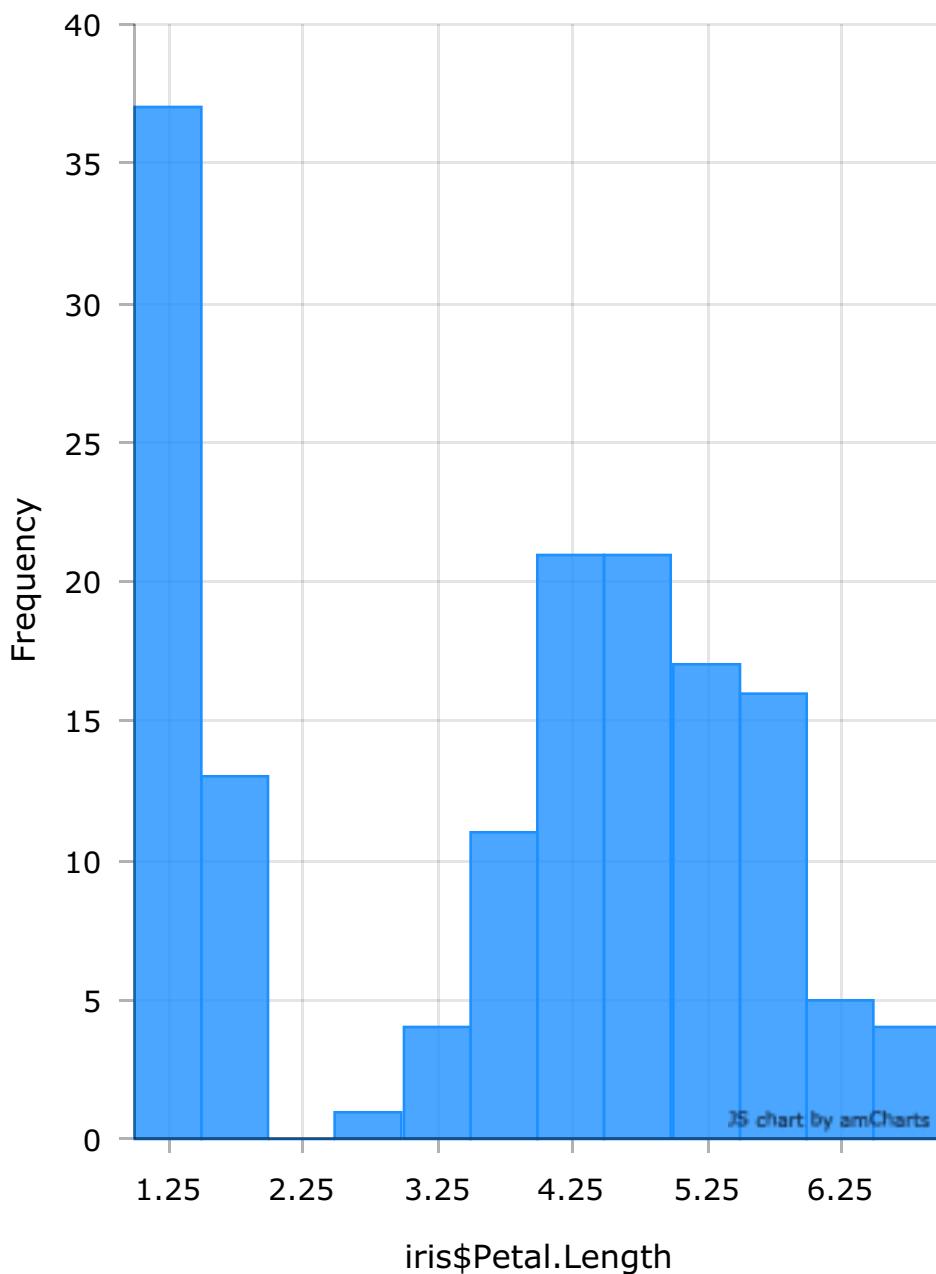
3 Quelques outils de visualisation interactive - Compléments shiny

Tout comme **leaflet** pour les cartes, il existe de nombreux outils **R** dédiés à la visualisation interactive. Nous en présentons quelques uns dans cette partie.

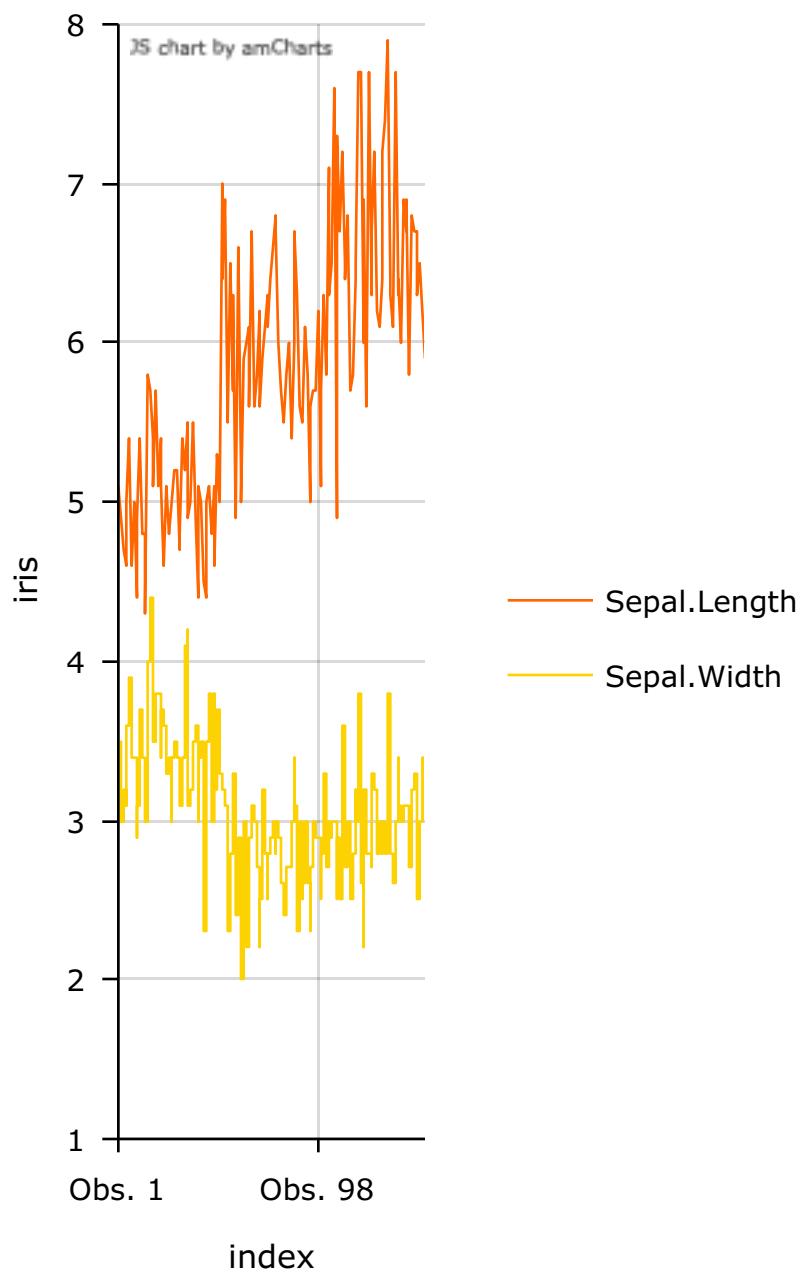
3.1 Représentations classiques avec **rAmCharts** et **plotly**

Le package **rAmCharts** est très utile pour donner un caractère interactif à des représentations graphiques standards (nuages de points, séries temporelles, histogrammes...). Ce package a été fait dans l'esprit d'utiliser les fonctions graphiques de **R** en utilisant le préfixe **am**. La syntaxe est très proche de celle des fonctions graphiques standards. On a par exemple :

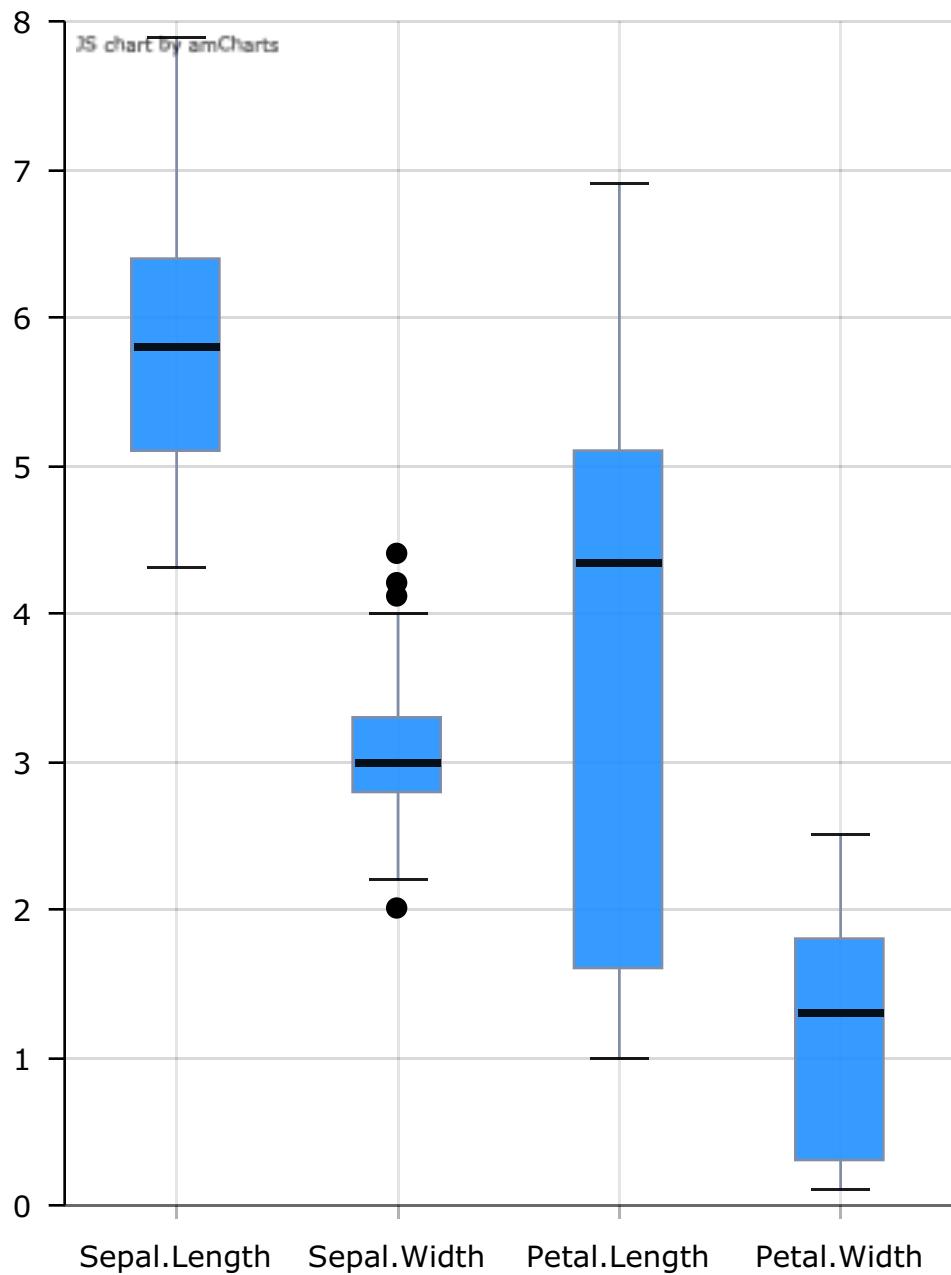
```
> library(rAmCharts)
> amHist(iris$Petal.Length)
```



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



```
> amBoxplot(iris)
```



plotly permet de faire des choses semblables avec une syntaxe spécifique. Les commandes **plotly** se décomposent essentiellement en 3 parties :

- le type de représentation graphique (**plot_ly**) ;
- les ajouts que l'on souhaite effectuer (**add_trace**) ;
- la gestion de la fenêtre graphique (axes, titres...) (**layout**).

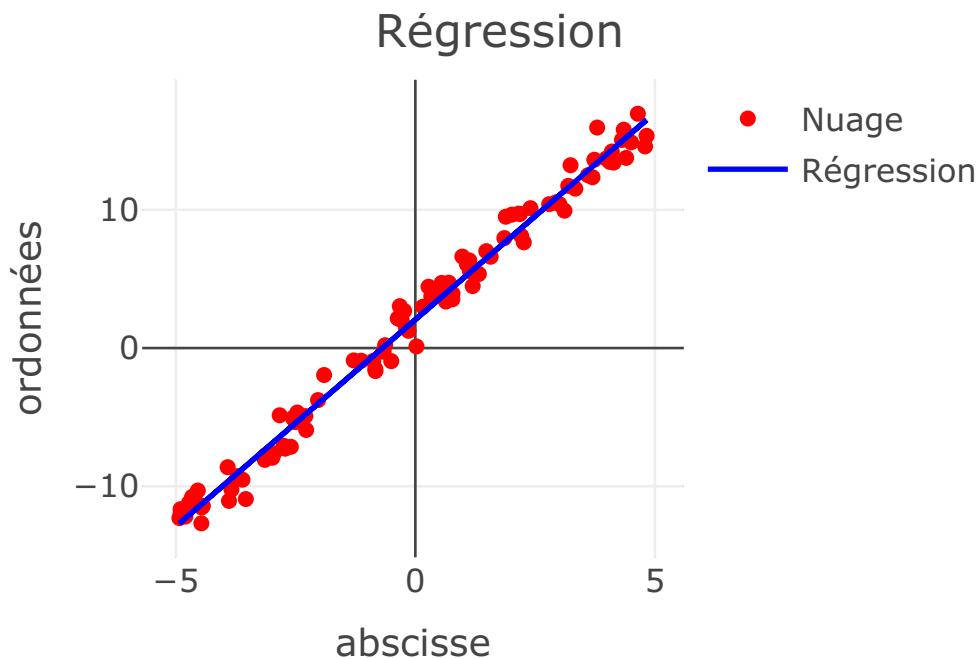
On trouvera un descriptif complet de ces 3 composantes ici. On propose ici de tracer un nuage de points en dimension 2 avec la droite de régression. On commence par générer le nuage et à ajuster le modèle linéaire :

```
> 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)
```

On effectue maintenant le trace

```
> 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"))
```

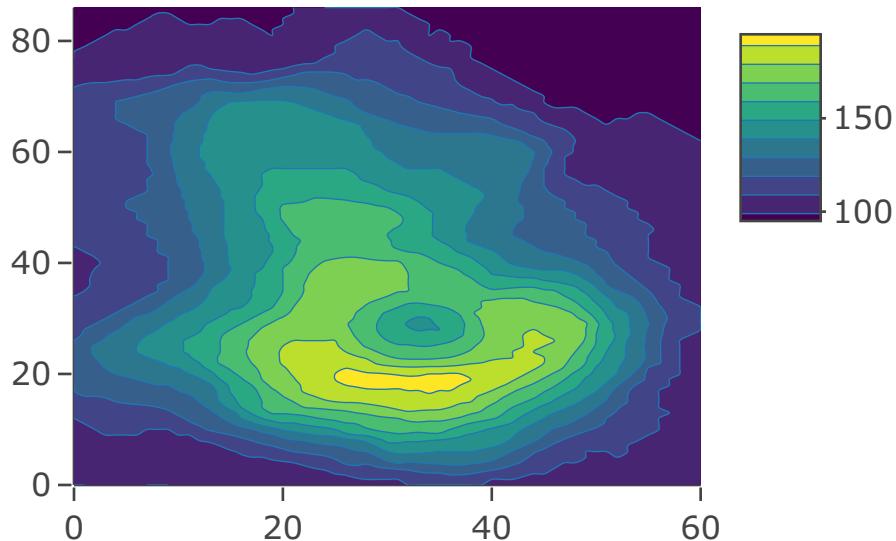


Contrairement à **ggplot**, **plotly** permet de faire de la 3D. Par exemple

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

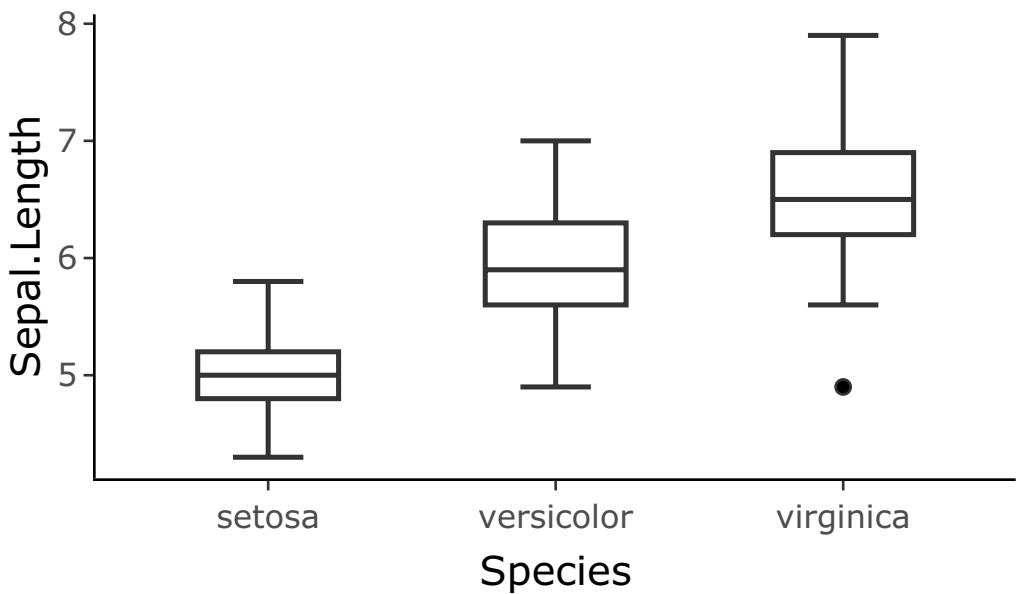
WebGL is not supported by your browser - visit <https://get.webgl.org> for more info

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



Il est possible de convertir des graphes **ggplot** au format **plotly** avec la fonction **ggplotly** :

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

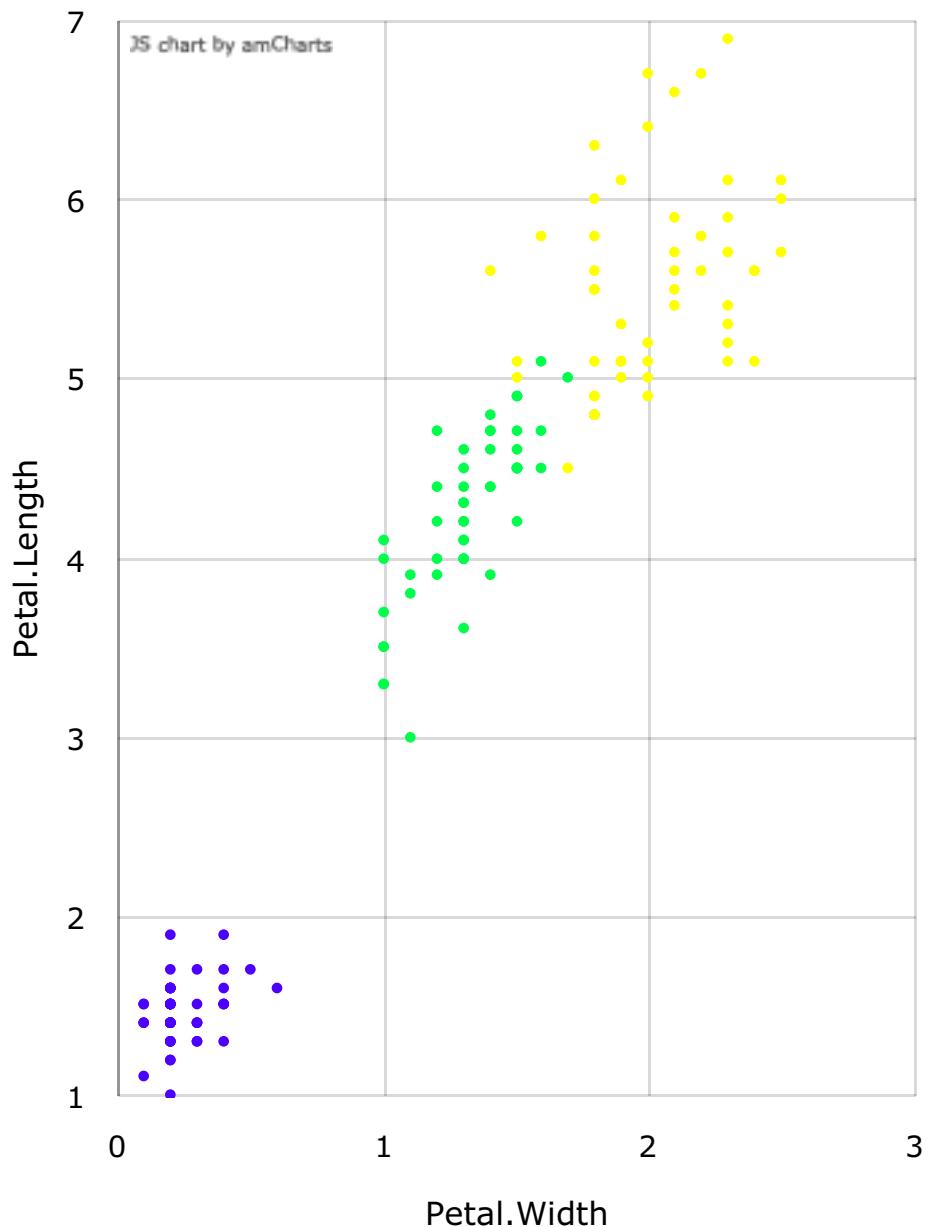


Exercice 3.1 (Graphes basiques avec ‘rAmCharts’ et ‘plotly’).

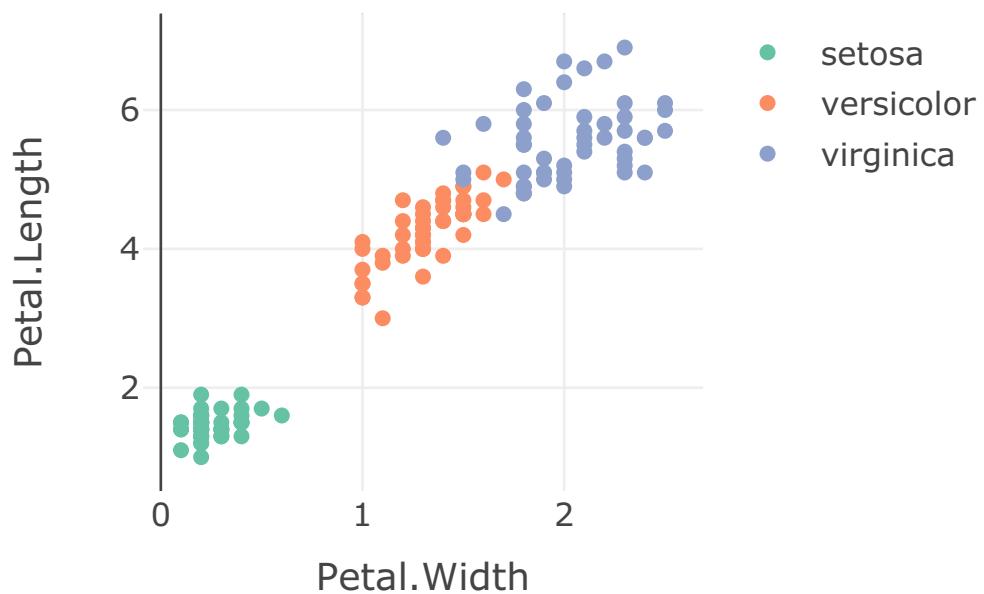
Pour le jeu de données **iris** on effectuera les graphes suivants en **rAmCharts** et **plotly**.

1. Nuage de points représentant les longueurs et largeurs de Sépales. On utilisera une couleur différente en fonction de l'espèce.

```
> amPlot(Petal.Length~Petal.Width, data=iris, col=iris$Species)
```

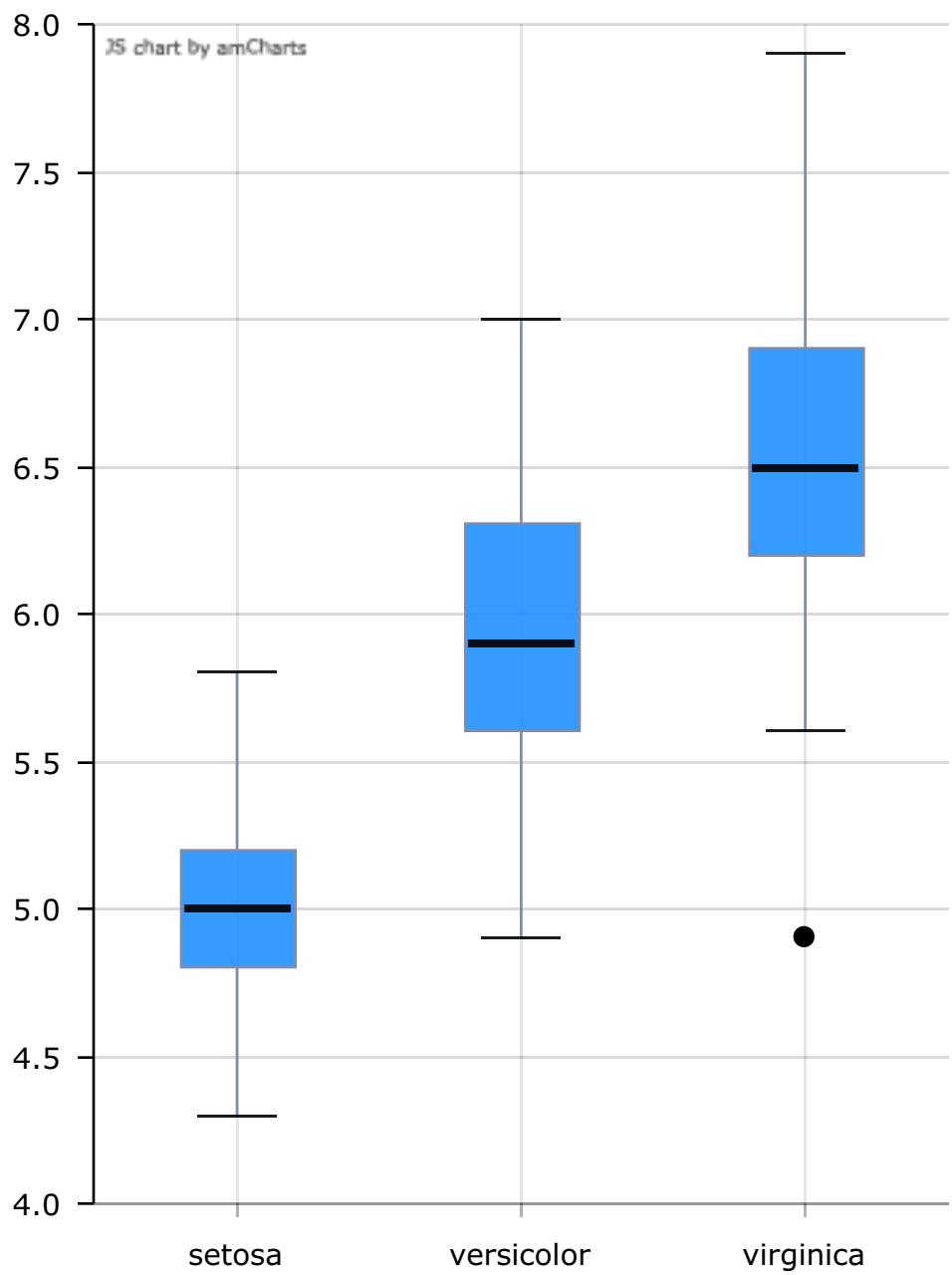


```
> iris %>% plot_ly(x=~Petal.Width, y=~Petal.Length, color=~Species) %>%  
+   add_markers(type="scatter", mode="markers")
```

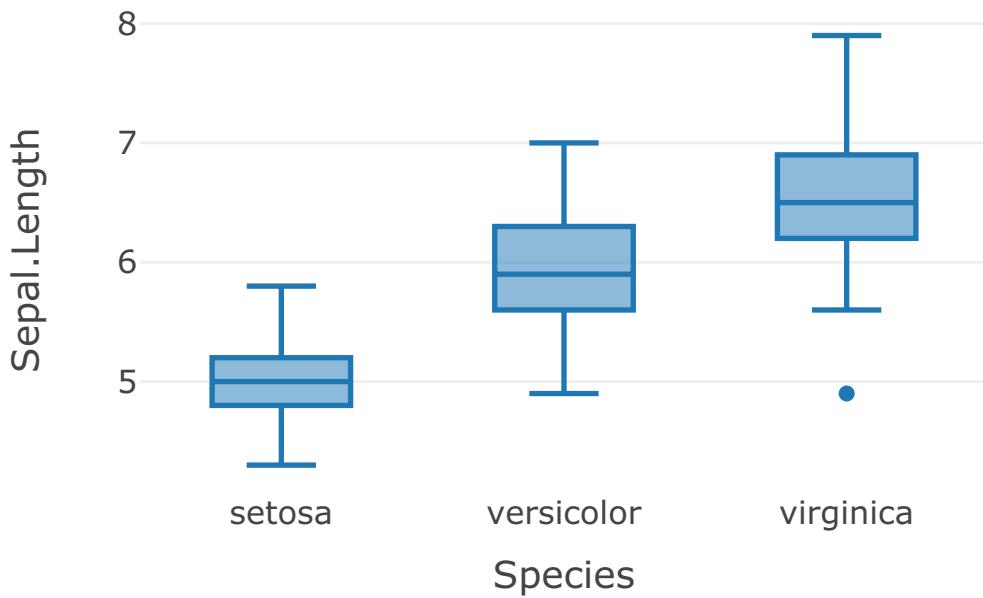


2. Boxplot permettant de visualiser la distribution de la variable Petal.Length en fonction de l'espèce.

```
> amBoxplot(Sepal.Length~Species,data=iris)
```



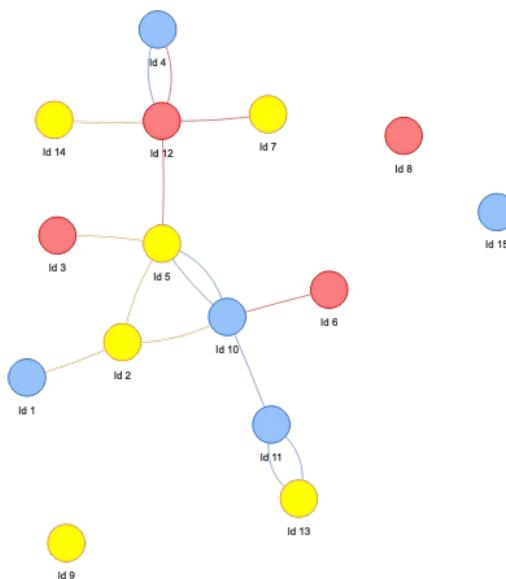
```
> iris %>% plot_ly(x=-Species,y=-Sepal.Length) %>% add_boxplot()
```



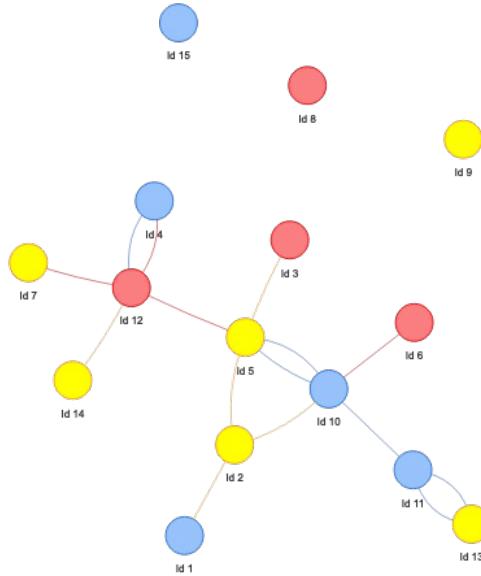
3.2 Graphes pour visualiser des réseaux avec visNetwork

De nombreuses données peuvent être visualisées à l'aide d'un graphe, notamment lorsqu'il s'agit de représenter des connexions entre individus. Un individu est alors représenté par un noeud et les individus connectés sont reliés par des arêtes. Le package igraph propose une visualisation statique d'un réseau. Pour donner un caractère dynamique à ce type de représentation, on pourra utiliser le package visNetwork. Une représentation standard **visNetwork** s'effectue en spécifiant les noeuds et connections d'un graphe, par exemple :

```
> 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)
```

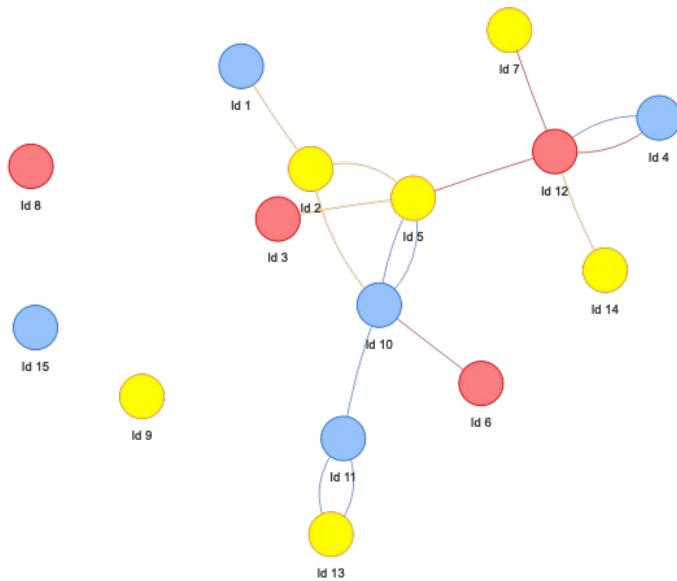


```
> visNetwork(nodes, edges) %>% visOptions(highlightNearest = TRUE)
```



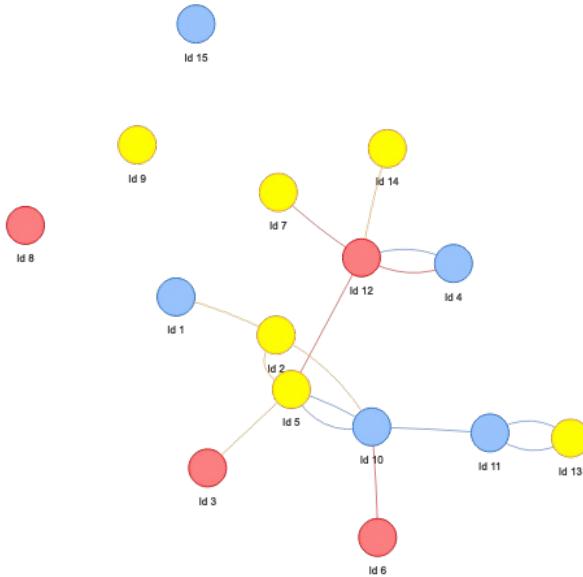
```
> visNetwork(nodes, edges) %>% visOptions(highlightNearest = TRUE, nodesIdSelection = TRUE)
```

Select by id ▾



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

Select by group ▾



Exercice 3.2 (Interactions entre media).

On considère un graphe qui représente des liens entre différents médias. Les données sont présentées ici et on peut les importer avec

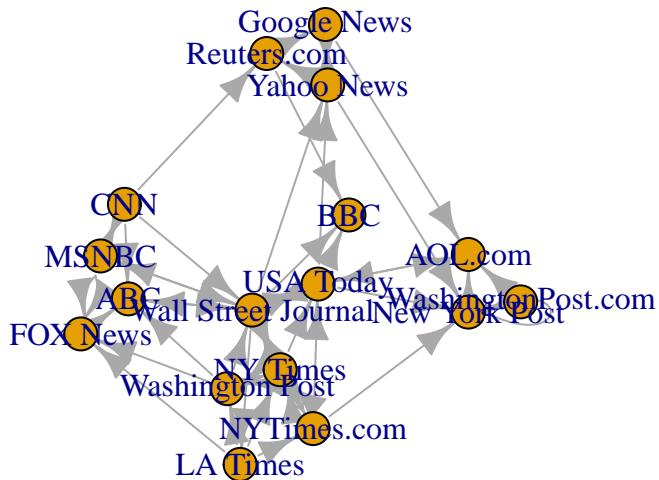
```
> nodes <- read.csv("Dataset1-Media-Example-NODES.csv", header=T, as.is=T)
> links <- read.csv("Dataset1-Media-Example-EDGES.csv", header=T, as.is=T)
> head(nodes)
##   id          media media.type type.label audience.size
## 1 s01        NY Times      1  Newspaper         20
## 2 s02  Washington Post      1  Newspaper         25
## 3 s03 Wall Street Journal      1  Newspaper         30
## 4 s04       USA Today      1  Newspaper         32
## 5 s05        LA Times      1  Newspaper         20
## 6 s06 New York Post      1  Newspaper         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
```

L'objet `nodes` représente les noeuds du graphe et l'objets `links` les arêtes. On définit l'objet `graphe` avec

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

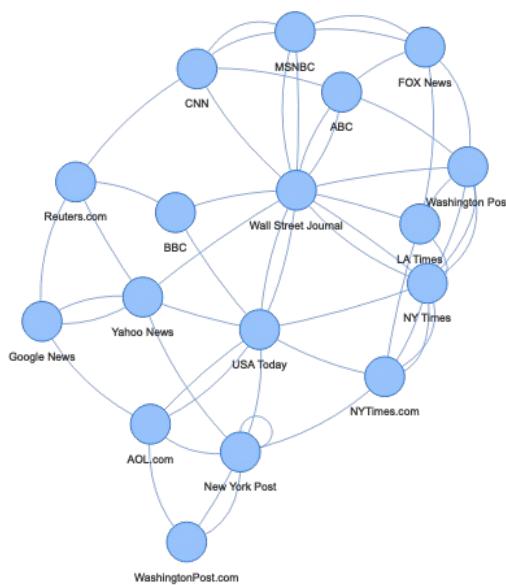
et on peut le visualiser en faisant un plot de cet objet

```
> plot(media)
```



1. Visualiser ce graphe avec **VisNetwork**. On pourra utiliser la fonction **toVisNetworkData**

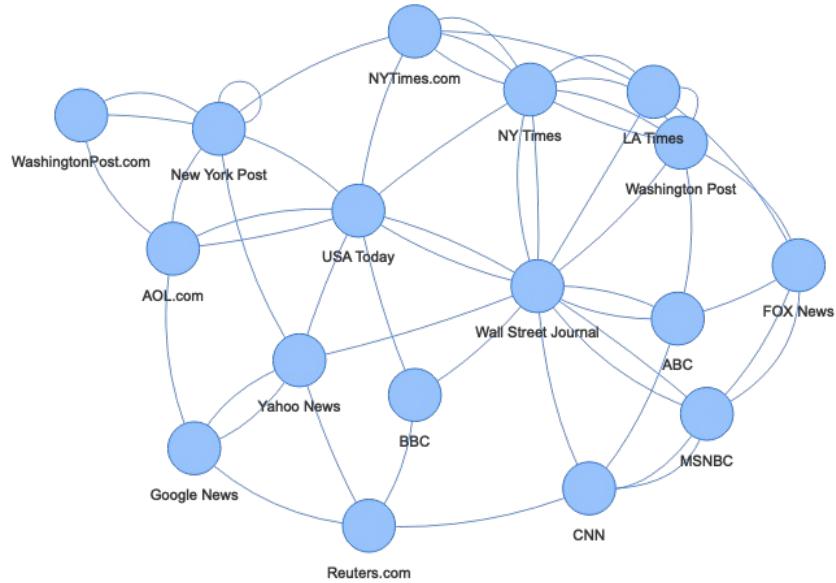
```
> media.VN <- toVisNetworkData(media)
> visNetwork(nodes=media.VN$nodes, edges=media.VN$edges)
```



2. Ajouter une option qui permette de sélectionner le type de media (Newspaper, TV ou Online).

```
> visNetwork(nodes=media.VN$nodes, edges=media.VN$edges) %>%
+   visOptions(selectedBy = "type.label")
```

Select by type.lab

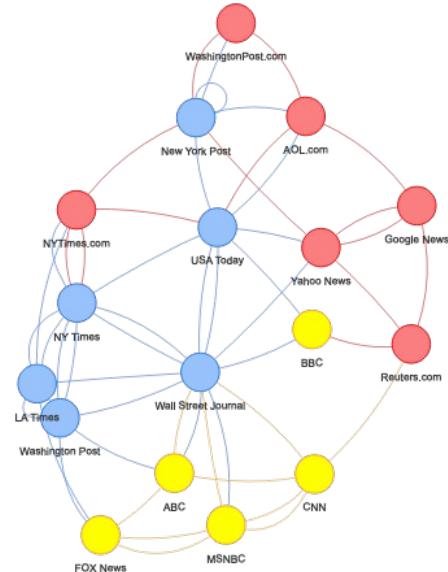


3. Utiliser une couleur différente pour chaque type de media.

Il suffit de donner le nom `group` à la variable `type.label`.

```
> media.VN1 <- media.VN  
> names(media.VN1$nodes)[3] <- "group"  
> visNetwork(nodes=media.VN1$nodes,edges=media.VN1$edges) %>%  
+   visOptions(selectedBy = "type.label")
```

Select by type.lab



4. Faire des flèches d'épaisseur différente en fonction du poids (weight). On pourra également ajouter l'option `visOptions(highlightNearest = TRUE)`.

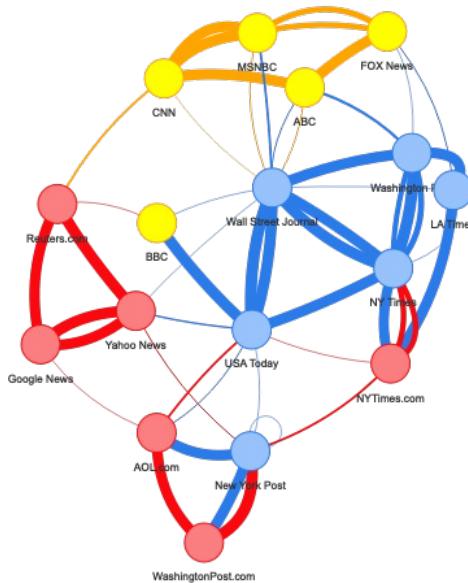
Il suffit de donner le nom **value** à la variable `weight`.

```

> names(media.VN1$edges)[3] <- "value"
> visNetwork(nodes=media.VN1$nodes,edges=media.VN1$edges) %>%
+   visOptions(selectedBy = "type.lab",highlightNearest = TRUE)

```

Select by type.lab ▾



3.3 Dashboard

Un tableau de bord permet de visualiser “facilement” et “rapidement” divers graphes et/ou résumés statistiques en lien avec une problématique donnée. Sur **R** le package **flexdashboard** permet de construire de tels tableaux de bord. On trouvera un descriptif précis de ce package à cette url : <https://rmarkdown.rstudio.com/flexdashboard/>. On utilisera cette documentation pour faire l’exercice suivant.

Exercice 3.3 (Dashboard pour modèles linéaires).

On considère le jeu de données **ozone.txt**. Le problème est d’expliquer la concentration maximale en ozone quotidienne (variable **maxO3**) par d’autres variables météorologiques (températures et indicateurs de nébulosité relevés à différents moments de la journée...). On souhaite faire un tableau de bord qui permettra de :

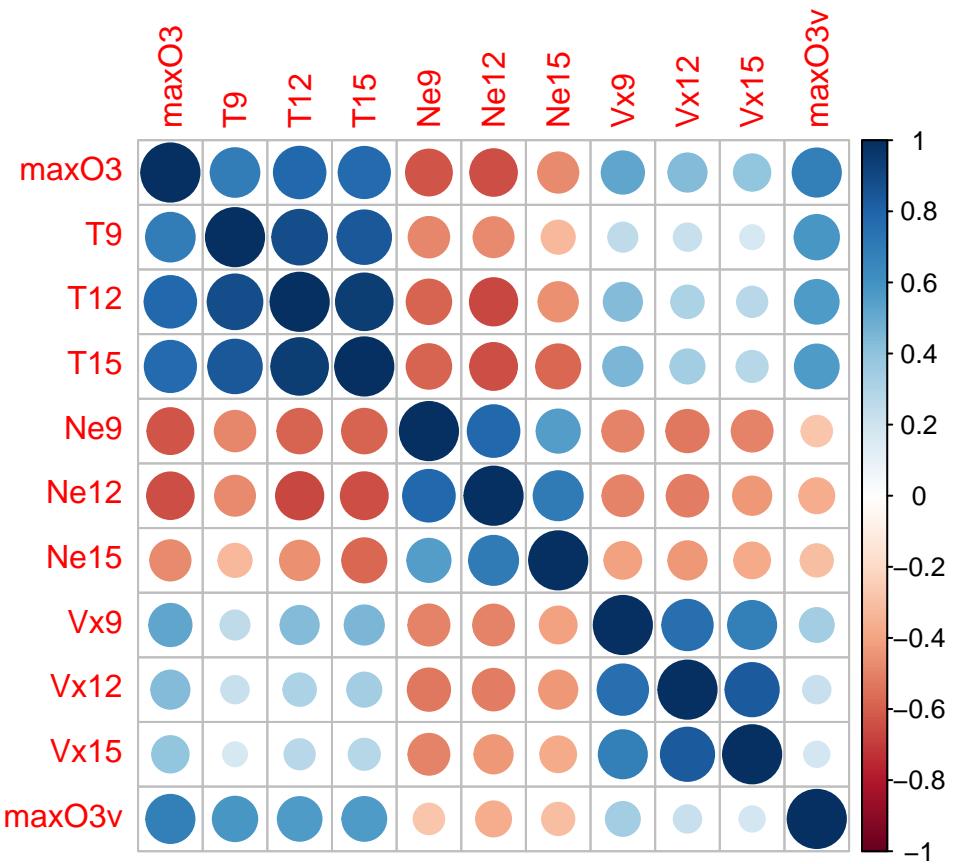
- visualiser les données : la base de données ainsi qu’un ou deux graphes descriptifs sur la variable à expliquer ;
- visualiser les modèles linéaires simples : on choisit une variable explicative et on visualise le graphe de la régression ainsi que le modèle ;
- visualiser le modèle linéaire complet : on affiche le résultat de la régression avec toutes les variables et on représente le graphe des résidus ;
- choisir les variables explicatives.

1. Avant de réaliser le dashboard, on propose d’écrire quelques commandes pour calculer les différentes sorties :
 - a. On considère uniquement les variables quantitatives du jeu de données. Visualiser les corrélations entre ces variables à l’aide de la fonction **corrplot** du package **corrplot**.

```

> df <- read.table("ozone.txt")
> cc <- cor(df[,1:11])
> mat.cor <- corrplot::corrplot(cc)

```



- b. Représenter l'histogramme de la variable **maxO3**, on fera le graphe **ggplot** et **rAmCharts** et **plotly** (en utilisant **ggplotly** par exemple).

```
> gg.H <- ggplot(df)+aes(x=maxO3)+geom_histogram(bins = 10)
> am.H <- amHist(df$maxO3)
> pl.H <- ggplotly(gg.H)
```

- c. Construire le modèle linéaire permettant d'expliquer **maxO3** par les autres variables. Calculer les résidus studentisés (**rstudent**) et visualiser ces résidus en fonction de la variable **maxO3**. Là encore on pourra ajouter un lissoir sur le graphe.

```
> mod <- lm(maxO3~.,data=df)
> res <- rstudent(mod)
> df1 <- data.frame(maxO3=df$maxO3,r.student=res)
> Ggg <- ggplot(df1)+aes(x=maxO3,y=res)+geom_point() +geom_smooth()
> Gggp <- ggplotly(Ggg)
```

2. On peut maintenant passer au tableau de bord. On utilise le menu **File -> Rmdardown -> From Template -> Flex Dashboard**.

- Construire un premier dashboard permettant de visualiser :
 - le jeu de données sur une colonne (on pourra utiliser la fonction **datatable** du package **DT**)
 - l'histogramme de la variable **maxO3** ainsi que la matrice des corrélations entre les variables quantitatives.
- Ajouter un onglet permettant de visualiser les modèles simples à une variable explicative. La variable explicative pourra être choisie à l'aide de **Shiny**. On pourra par exemple utiliser

```
> radioButtons("variable1",
+               label="Choisir la variable explicative",
```

```
+         choices=names(df)[-1] ,
+         selected=list("T9"))
```

On n'oubliera pas d'ajouter

```
> runtime: shiny
```

dans l'entête.

- c. Ajouter un onglet permettant de visualiser les coefficients du modèle linéaire complet ainsi que le graphe des résidus effectués à la questions 1.c.
- d. Ajouter un nouvel onglet permettant de choisir les variables explicatives dans le modèle linéaire. Là encore on pourra utiliser des commandes **Shiny**, par exemple

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

Pour les variables choisies, on affichera dans ce nouvel onglet les coefficients du modèle linéaire ainsi que le graphe des résidus studentisés.

Le tableau de bord finalisé pourra ressembler à



Il est disponible à l'url <https://lrouviere.shinyapps.io/dashboard/>