

Tutoriel : visualisation avec R

Laurent Rouvière

2021-12-17

Table des matières

Présentation	1
1 Visualisation avec ggplot2	2
1.1 Fonctions graphiques conventionnelles	2
1.2 La grammaire ggplot2	11
1.3 Compléments	31
1.4 Quelques exercices supplémentaires	39
2 Faire des cartes avec R	44
2.1 Le package ggmap	44
2.2 Cartes avec contours, le format shapefile	48
2.3 Cartes interactives avec leaflet	56
3 Quelques outils de visualisation dynamique/interactive	66
3.1 Représentations classiques avec rAmCharts et plotly	66
3.2 Graphes pour visualiser des réseaux avec visNetwork	74
3.3 Dashboard	77
4 Applications web avec Shiny	80
4.1 Une première application	80
4.2 Input - output	80
4.3 Structurer l'application	83
4.4 Ajout de graphes interactifs	85
4.5 Reactive, isolation, observe, html,	86
4.6 Exercices complémentaires	89

Présentation

Ce tutoriel présente quelques outils **R** pour la **visualisation de données**. On pourra trouver :

- les supports de cours associés à ce tutoriel ainsi que les données utilisées à l'adresse suivante <https://lrouviere.github.io/VISU/>;
- le tutoriel sans les corrections à l'url https://lrouviere.github.io/TUTO_VISU/
- le tutoriel avec les corrigés (à certains moments) à l'url https://lrouviere.github.io/TUTO_VISU/correction/.

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

Des connaissances de base en **R** et en programmation sont requises. Le tutoriel 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**).

1 Visualisation avec ggplot2

Il est souvent nécessaire d'utiliser des techniques de visualisation au cours des différentes étapes d'un projet en science des données. Un des avantages de **R** est qu'il est relativement simple de mettre en œuvre tous les types de graphes généralement utilisés. Dans cette partie, 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

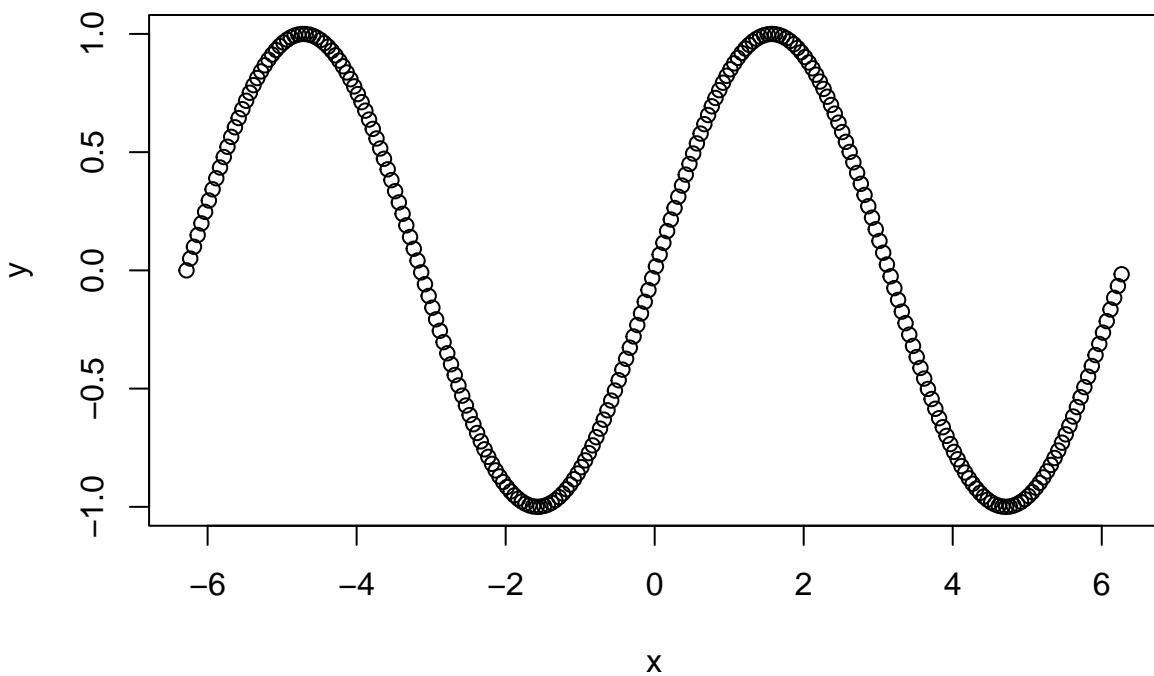
Pour commencer il est intéressant d'examiner quelques exemples de représentations graphiques construits avec **R**. On peut les obtenir à l'aide de la fonction **demo**.

```
demo(graphics)
```

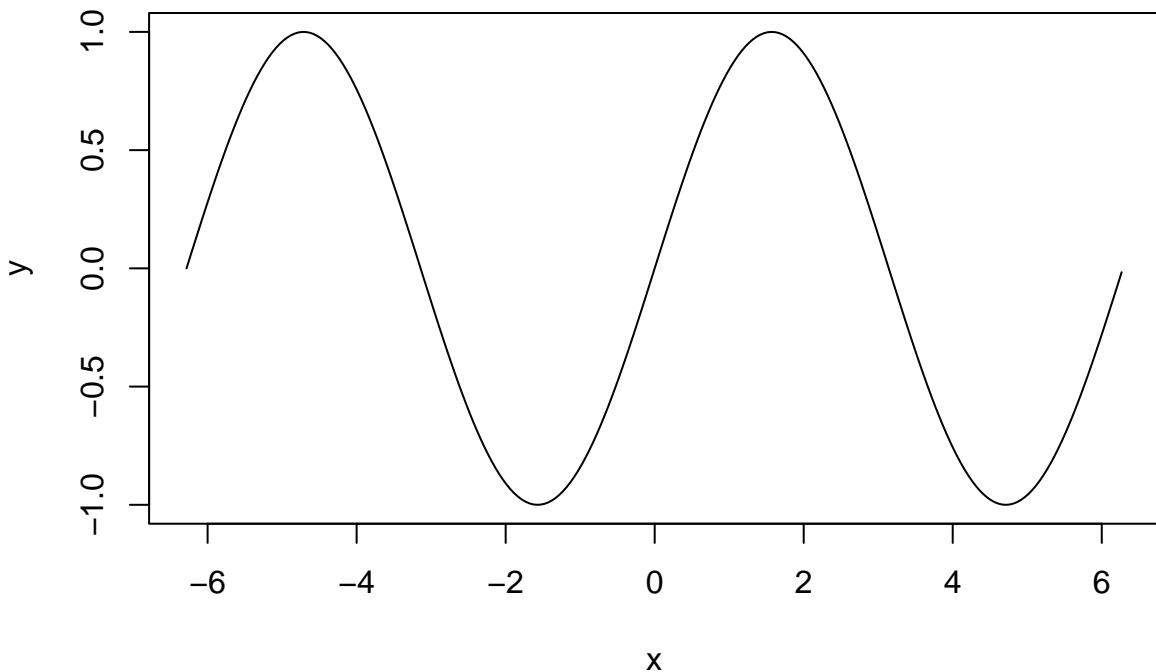
1.1.1 La fonction plot

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 travers du jeu de données **ozone.txt** que l'on importe avec

```
ozone <- read.table("data/ozone.txt")
summary(ozone)

   max03          T9          T12
Min.   : 42.00  Min.   :11.30  Min.   :14.00
1st Qu.: 70.75  1st Qu.:16.20  1st Qu.:18.60
Median : 81.50  Median :17.80  Median :20.55
Mean   : 90.30  Mean   :18.36  Mean   :21.53
3rd Qu.:106.00  3rd Qu.:19.93  3rd Qu.:23.55
Max.   :166.00  Max.   :27.00  Max.   :33.50

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

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

   Vx15          max03v          vent
Min.   :-9.000  Min.   : 42.00  Length:112
1st Qu.:-3.939  1st Qu.: 71.00  Class :character
```

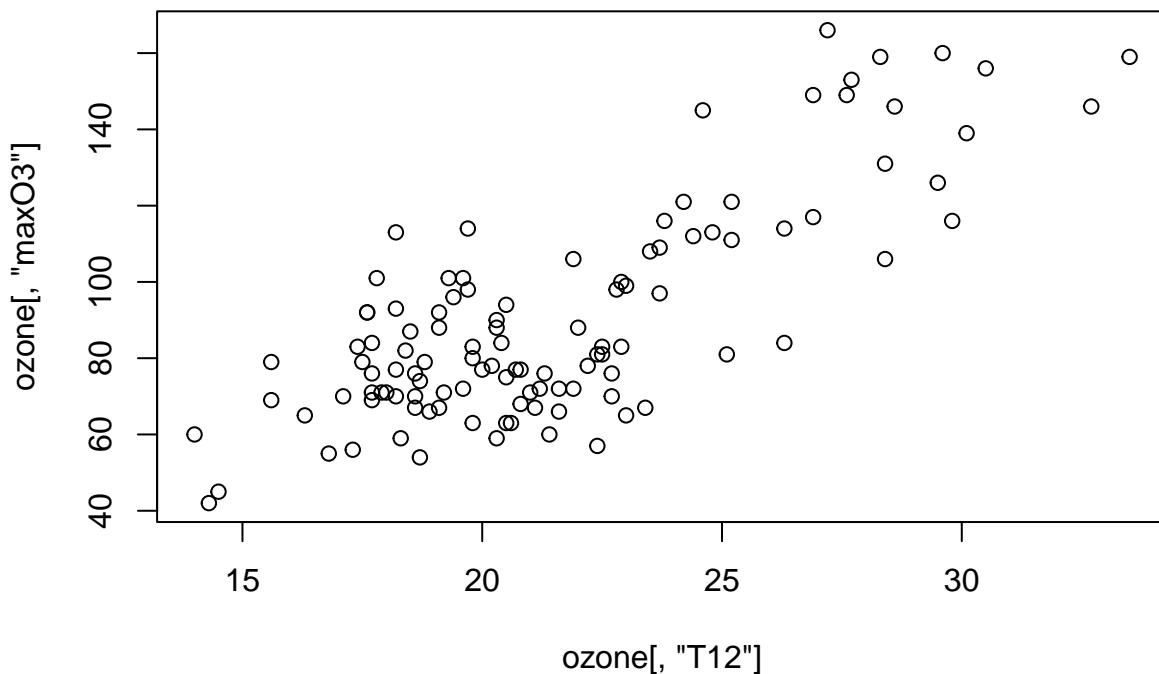
```

Median : -1.550   Median : 82.50   Mode  :character
Mean   : -1.691   Mean   : 90.57
3rd Qu.: 0.000    3rd Qu.:106.00
Max.   : 5.000    Max.   :166.00
pluie
Length:112
Class :character
Mode   :character

```

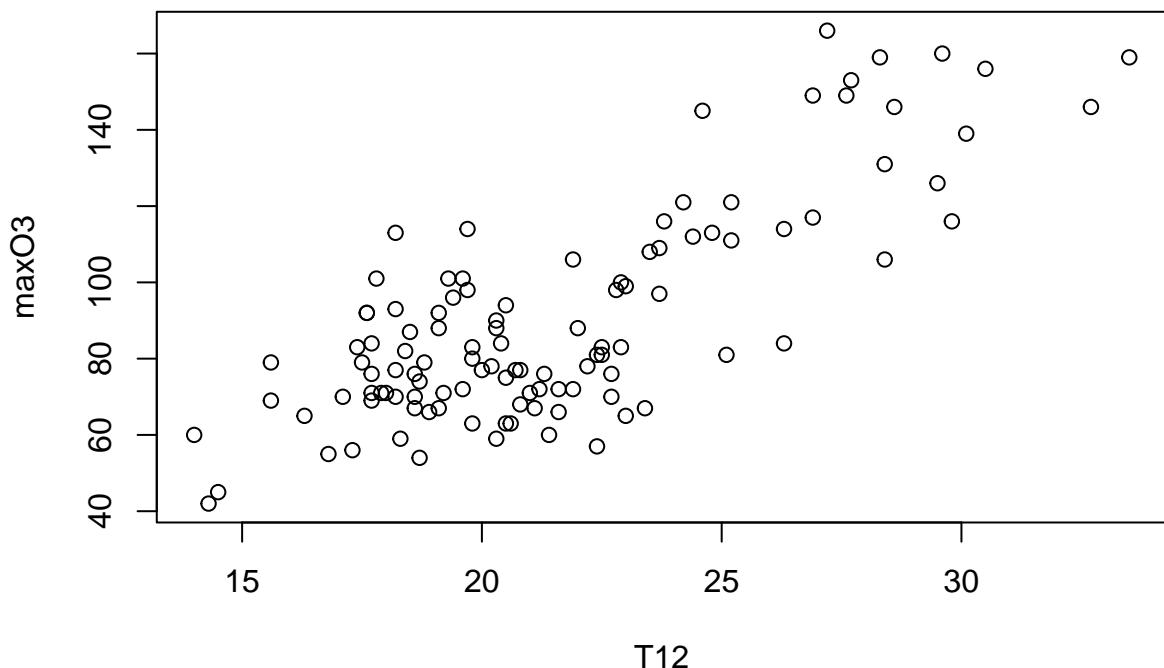
On visualise tout d'abord 2 variables quantitatives à l'aide d'un nuage de points : la concentration en ozone maximale **maxO3** en fonction de la température à 12h **T12**.

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



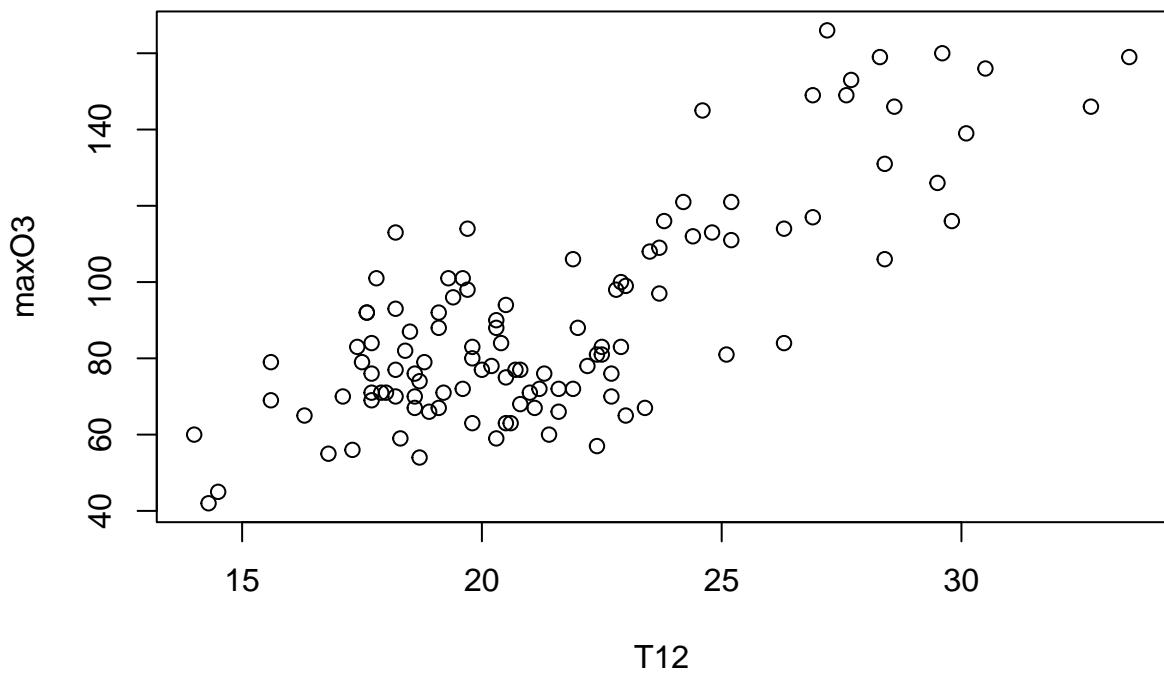
Comme les deux variables appartiennent au même jeu de données, on peut obtenir la même représentation à l'aide d'une syntaxe plus claire qui ajoute automatiquement les noms des variables sur les axes :

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



Une autre façon de faire (moins naturelle) :

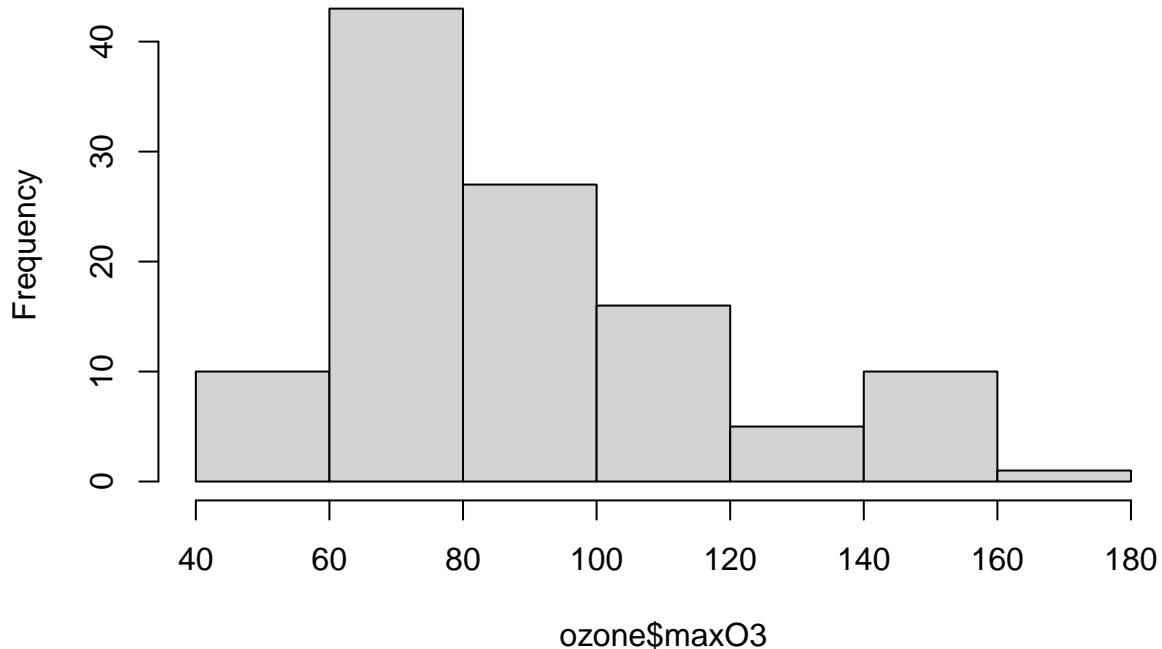
```
plot(ozone[, "T12"], ozone[, "maxO3"], xlab="T12", ylab="maxO3")
```



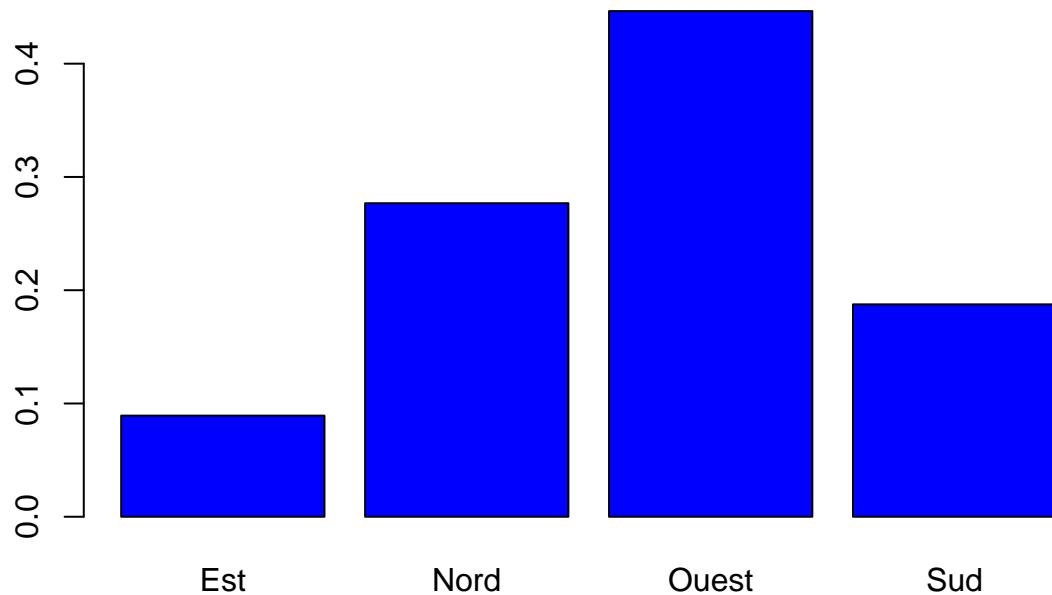
Il existe des fonctions spécifiques pour chaque type de graphes, par exemple **histogram**, **barplot** et **boxplot** :

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

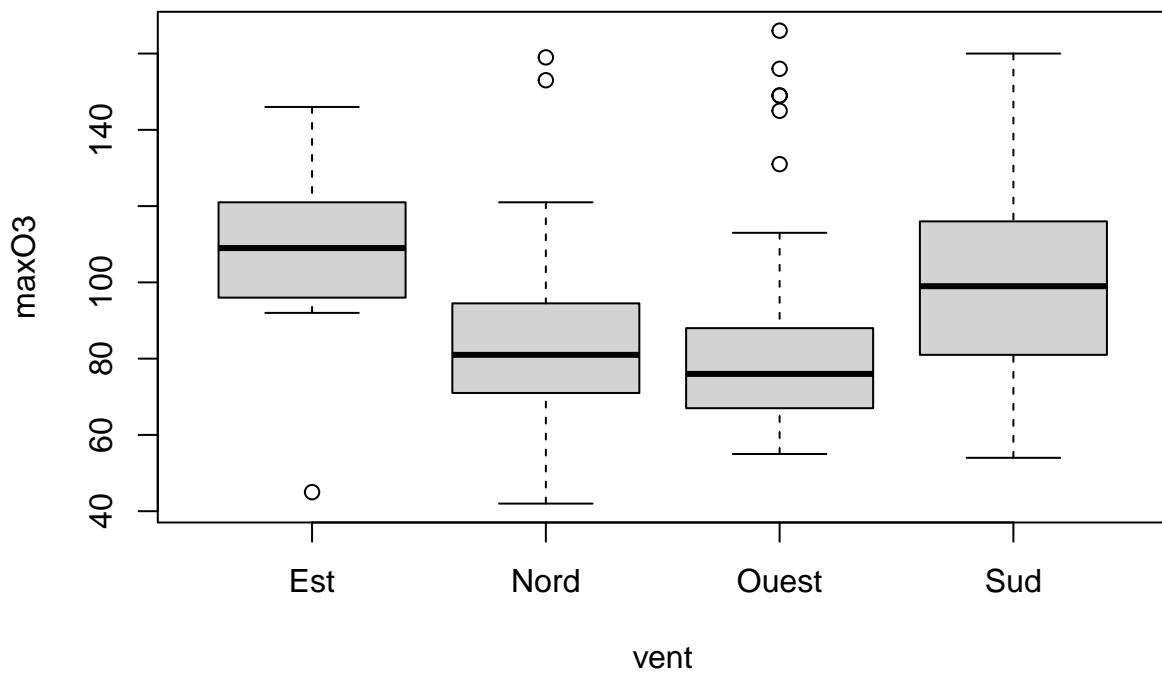
Histogram



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



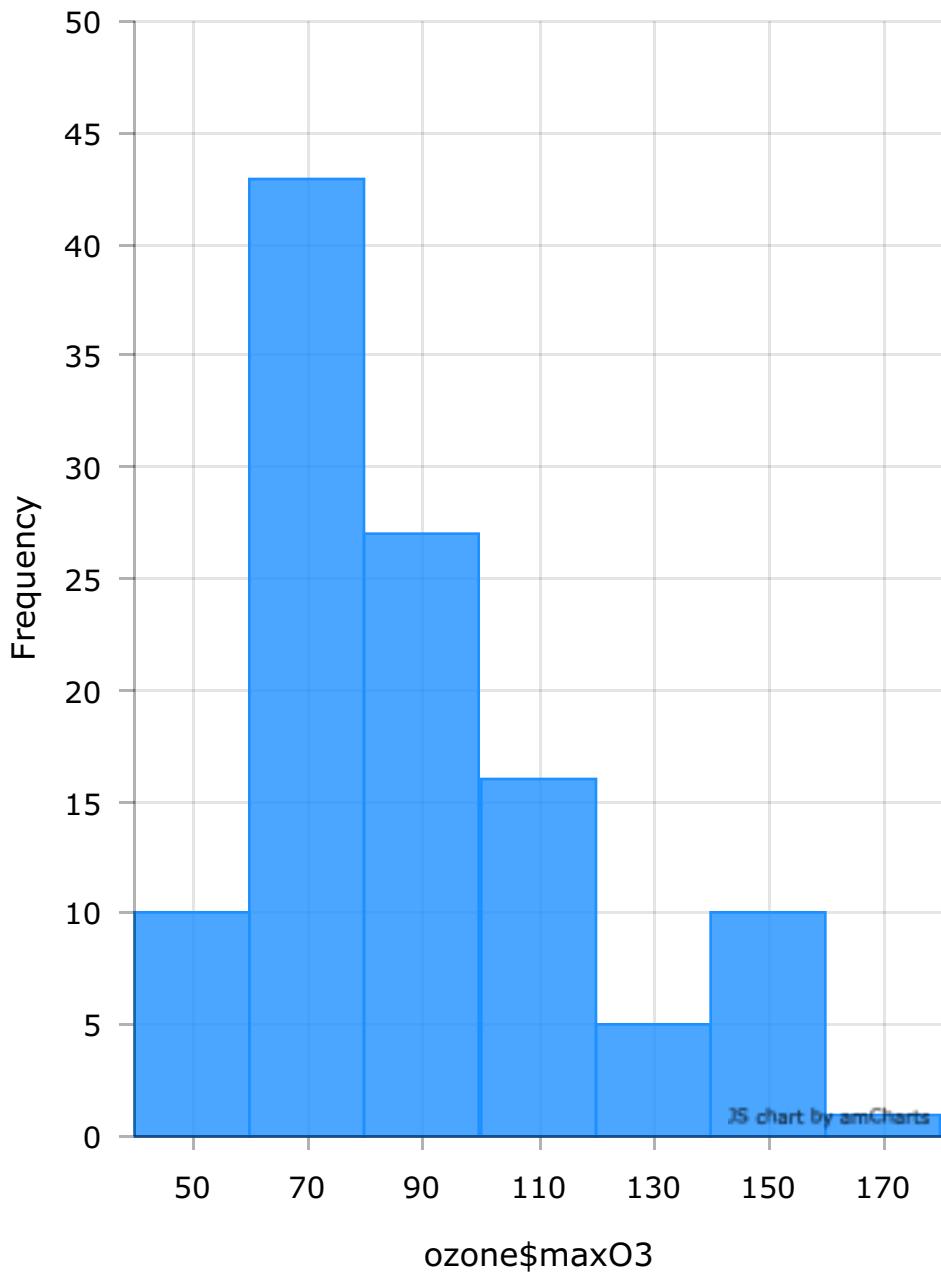
```
boxplot(maxO3~vent,data=ozone)
```



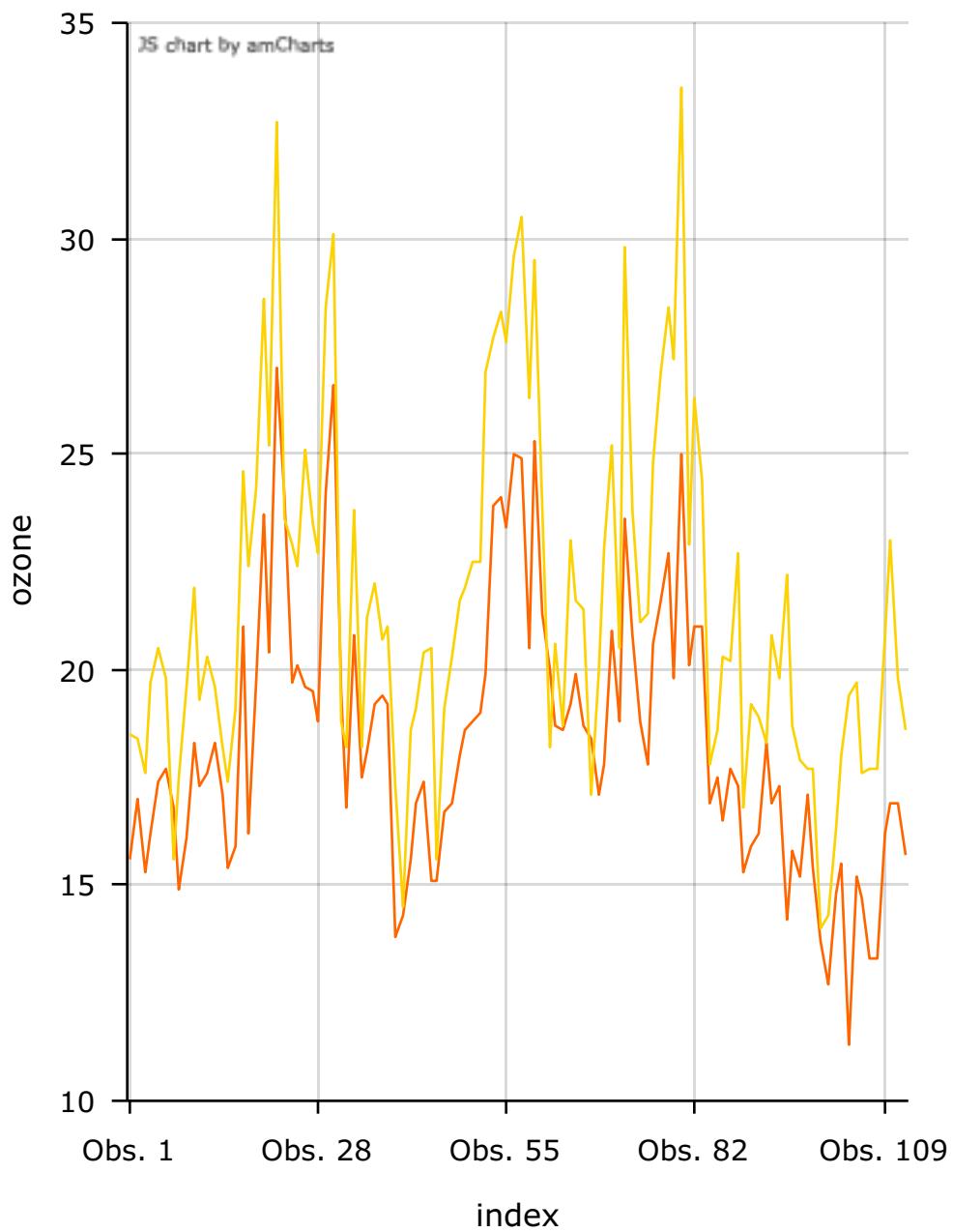
1.1.2 Graphes interactifs avec rAmCharts

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 :

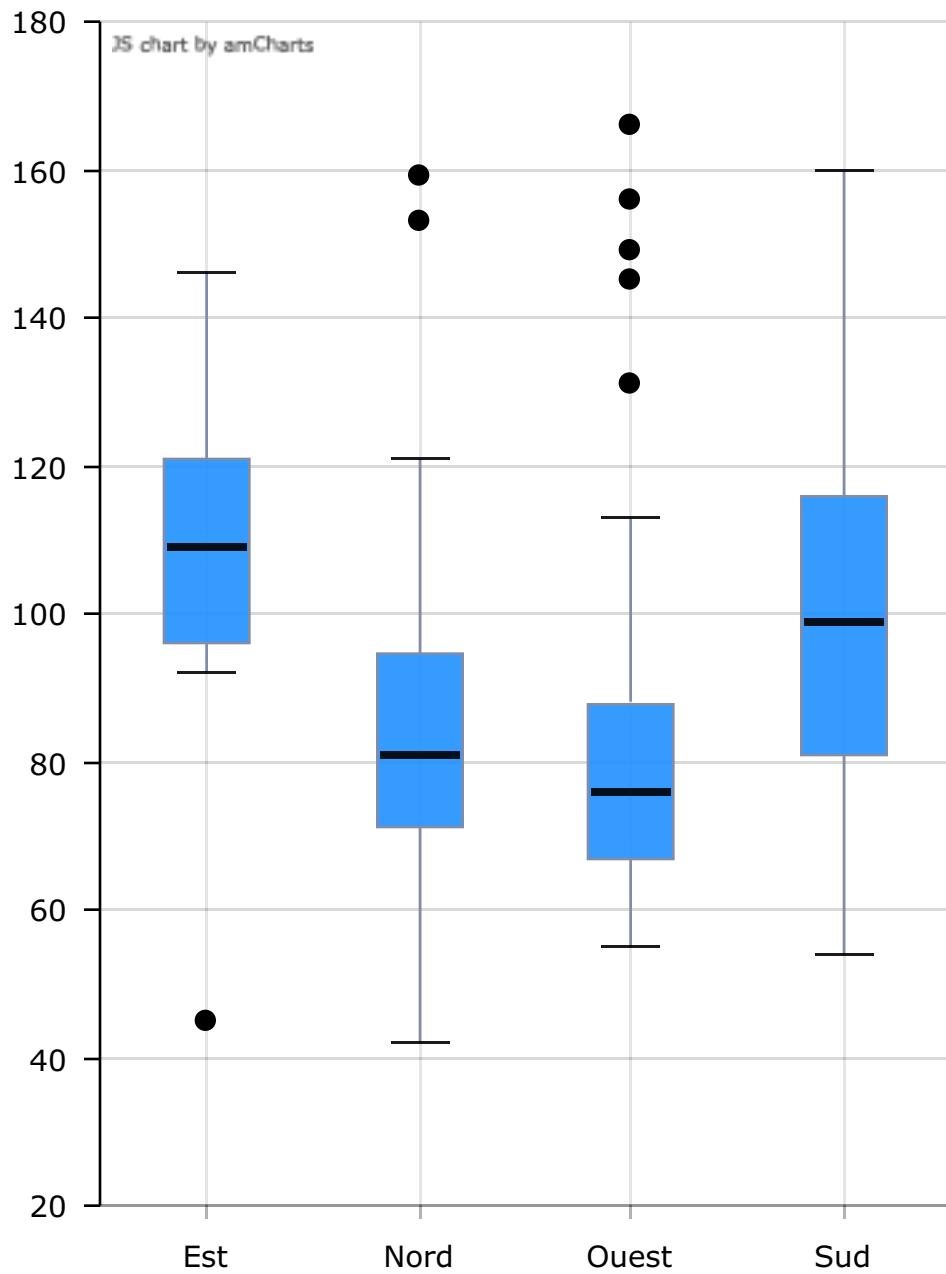
```
library(rAmCharts)
amHist(ozone$max03)
```



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



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



1.1.3 Quelques exercices

Exercice 1.1 (Premier graphe). On s'intéresse à quelques graphes simples.

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

Exercice 1.2 (Tracé de densités). On souhaite ici visualiser et comparer des densités de probabilité.

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 l'option **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 permette d'identifier chaque densité (fonction **legend**).

Exercice 1.3 (Tâches solaires). On souhaite ici visualiser une série temporelle.

1. Importer la série **taches_solaires.csv** qui donne, date par date, un nombre de tâches solaires observées.
2. A l'aide de la fonction **cut_interval** du package **ggplot2**, créer un facteur qui sépare l'intervalle d'années d'observation en 8 intervalles de tailles à peu près égales. On appellera **periode** ce facteur.
3. Utiliser les levels suivants pour le facteur **periode**.

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

4. Expliquer la sortie de la fonction
5. Visualiser la série du nombre de tâches en utilisant une couleur différente pour chaque période.

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

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

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 <https://ggplot2.tidyverse.org> et <https://ggplot2-book.org/index.html>

1.2.1 Premiers graphes ggplot2

Nous considérons un sous échantillon du jeu de données **diamonds** du package **ggplot2** (que l'on peut également charger avec le package **tidyverse**).

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

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

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

```

      x           y           z
Min. : 0.000  Min. :3.720  Min. :0.000
1st Qu.: 4.720 1st Qu.:4.720 1st Qu.:2.920
Median : 5.690 Median :5.700 Median :3.520
Mean   : 5.728 Mean  :5.731 Mean  :3.538
3rd Qu.: 6.530 3rd Qu.:6.520 3rd Qu.:4.030
Max.   :10.000 Max. :9.850 Max. :6.430

```

```
help(diamonds)
```

Un graphe **ggplot** est défini à partir de **couches** que l'on assemblera avec l'opérateur **+**. Il faut à minima spécifier :

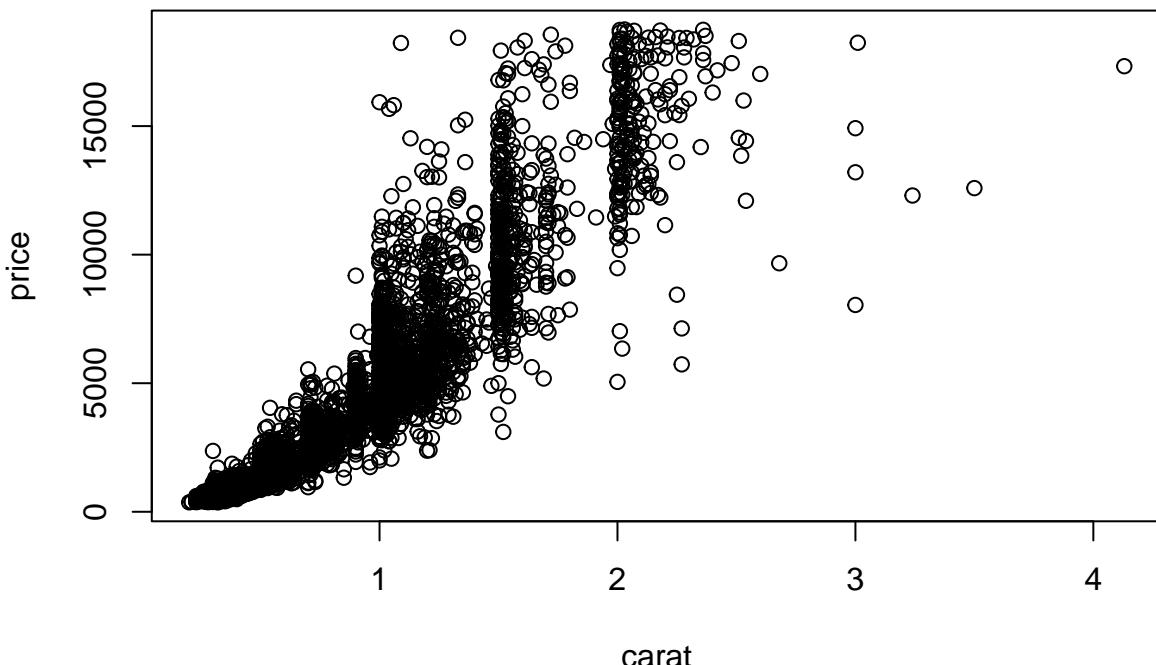
- les données
- les variables que l'on souhaite représenter
- le type de représentation (nuage de points, boxplot...).

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** :

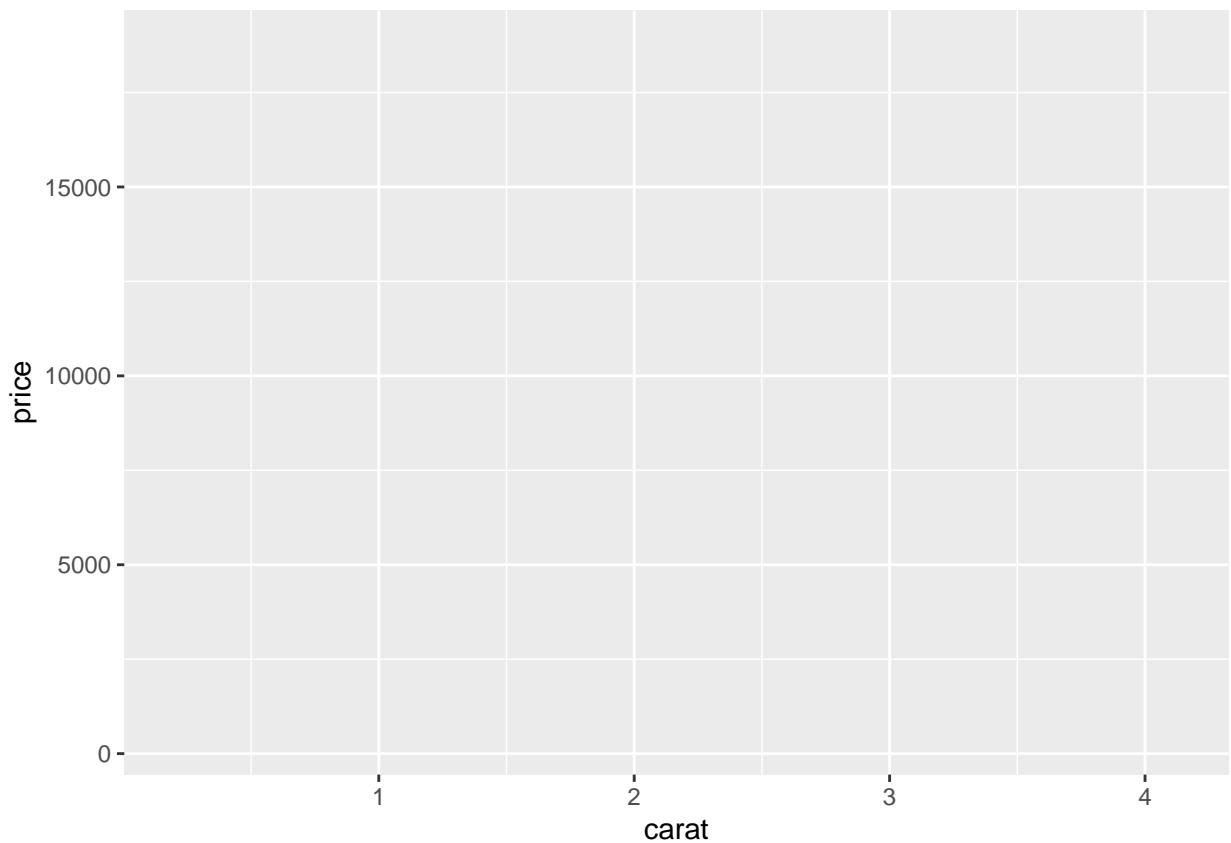
```
plot(price~carat, data=diamonds2)
```



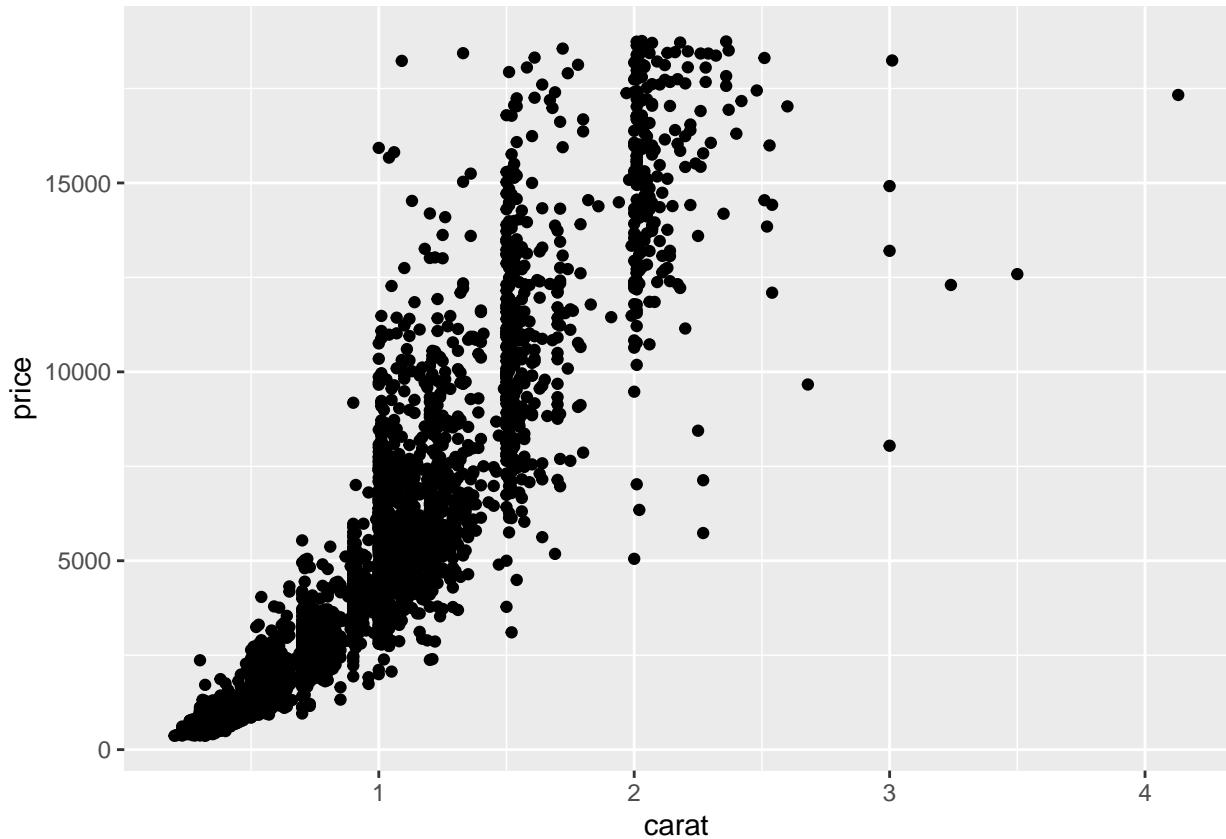
Avec **ggplot**, on va faire

```
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. Même question en utilisant 10 classes pour l'histogramme (`help(geom_histogram)`).
3. Tracer le diagramme en barres de la variable `cut` (utiliser `geom_bar`).

La syntaxe **ggplot** est définie 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` sert à spécifier 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 les variables que l'on souhaite visualiser. Par exemple, pour le nuage de points **price vs carat** la syntaxe débute avec

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

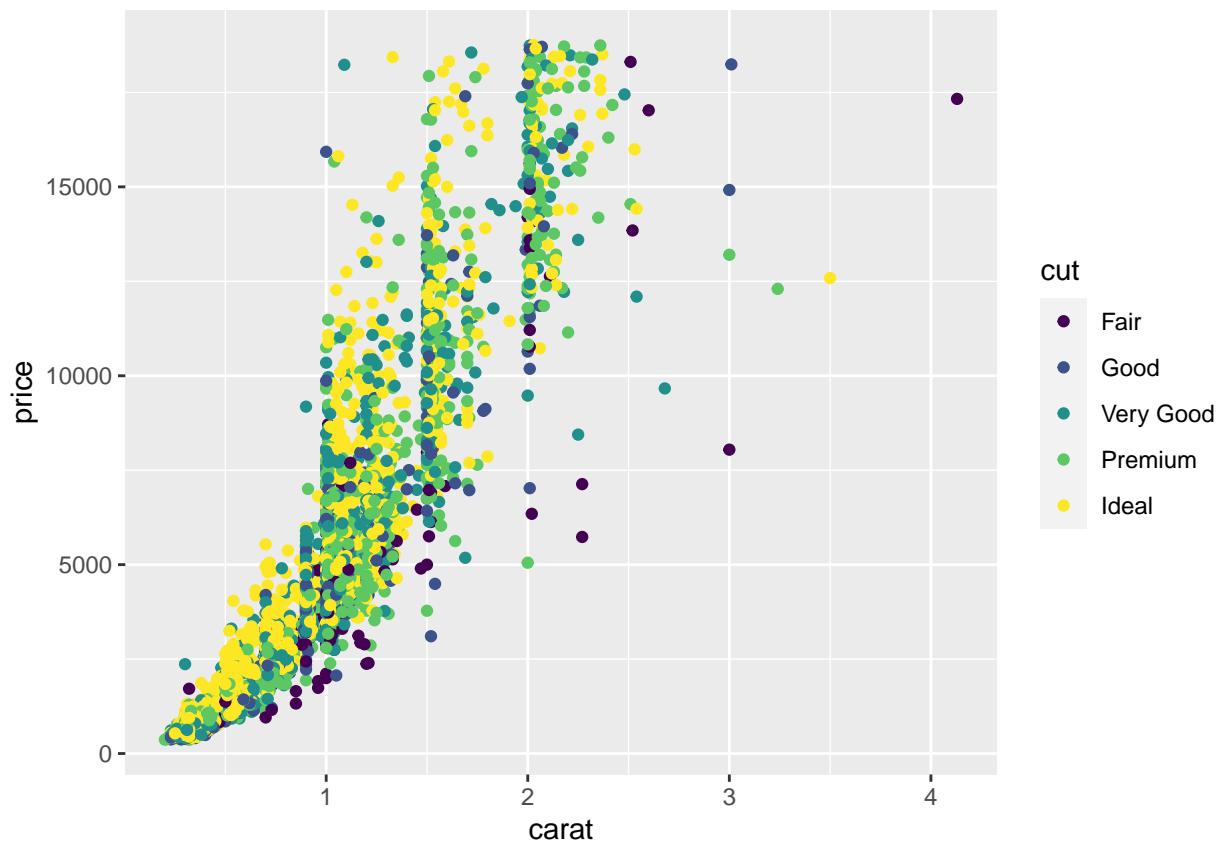
Les variables peuvent également être utilisées pour colorier des points ou des barres, définir des tailles... Dans ce cas on pourra renseigner les options **color**, **size**, **fill** dans la fonction **aes**. Par exemple

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

1.2.3 Geometrics

Ce verbe décrira le type de représentation souhaité. 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
geom_point()	nuage de points	x, y, shape, fill
geom_line()	Ligne (ordonnée selon x)	x, y, linetype
geom_abline()	Ligne	slope, intercept
geom_path()	Ligne (ordonnée par l'index)	x, y, linetype
geom_text()	Texte	x, y, label, hjust, vjust
geom_rect()	Rectangle	xmin, xmax, ymin, ymax, fill, linetype

Geom	Description	Aesthetics
geom_polygon()	Polygone	x, y, fill, linetype
geom_segment()	Segment	x, y, xend, yend, fill, linetype
geom_bar()	Diagramme en barres	x, fill, linetype, weight
geom_histogram()	Histogramme	x, fill, linetype, weight
geom_boxplot()	Boxplot	x, fill, weight
geom_density()	Densité	x, y, fill, linetype
geom_contour()	Lignes de contour	x, y, fill, linetype
geom_smooth()	Lisseur (linéaire ou non linéaire)	x, y, fill, linetype
Tous		color, size, group

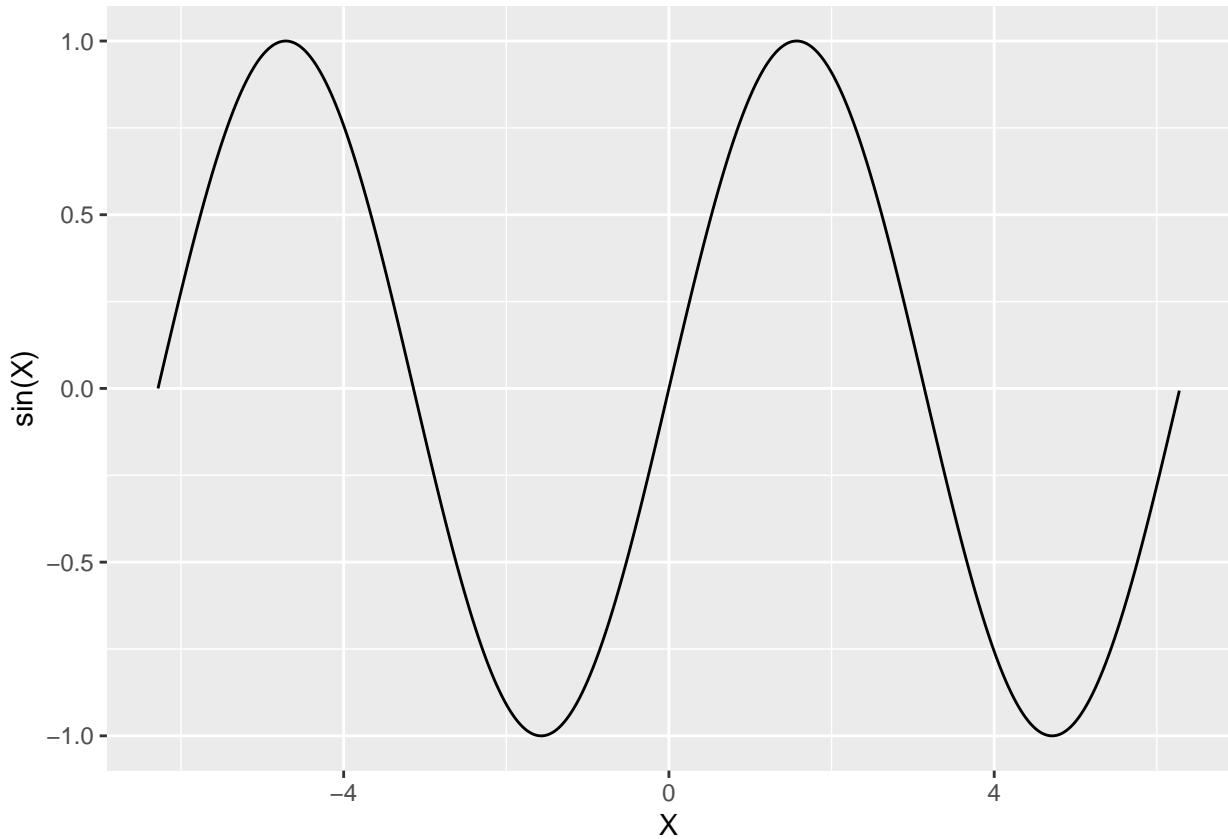
Exercice 1.6 (Diagrammes en barres). On étudie différentes façons de changer la couleur dans un diagramme en barres.

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).

1.2.4 Statistics

Certains graphes nécessitent des calculs d'indicateurs statistiques pour être tracé. C'est par exemple le cas pour le diagramme en barres et l'histogramme où il faut calculer des hauteurs de rectangles ou barres. On peut spécifier les transformations simples facilement, par exemple

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

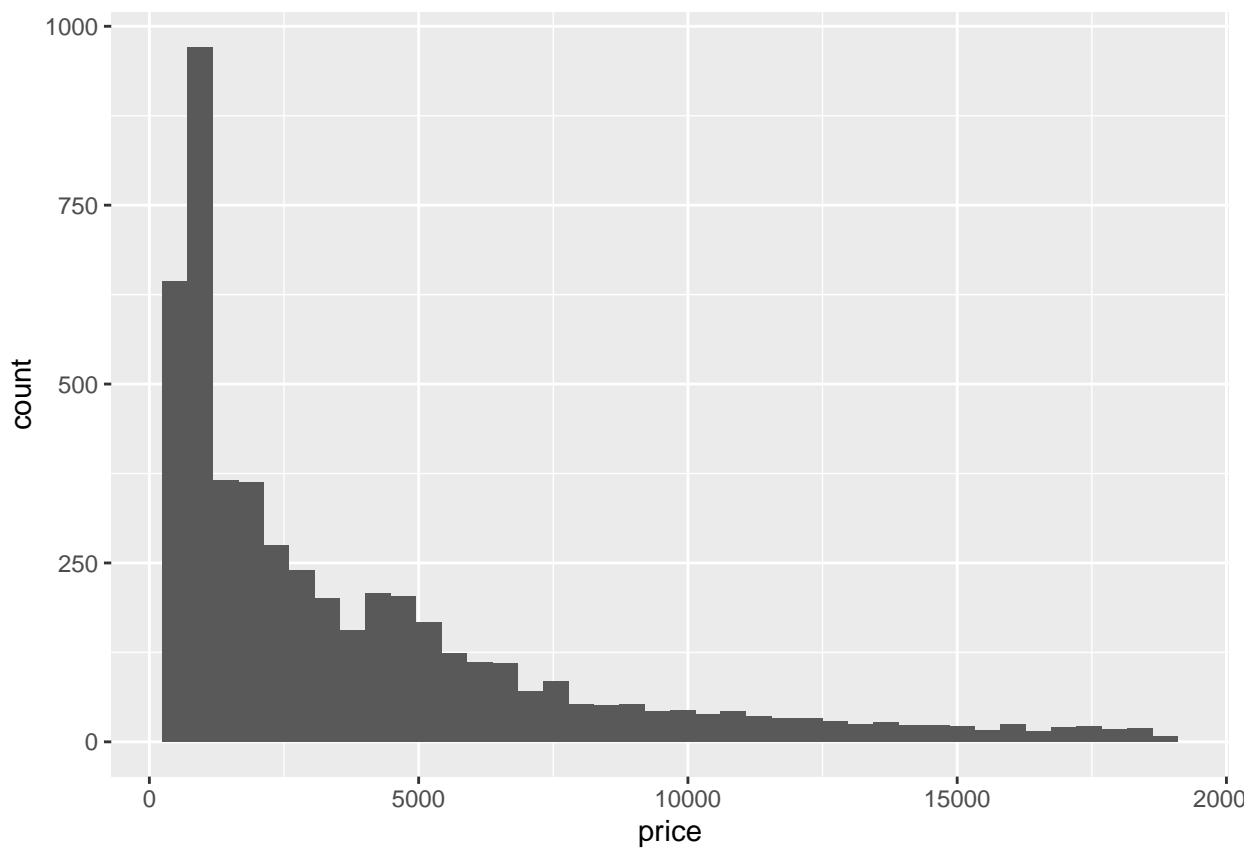


La transformation est spécifiée dans la fonction **aes**. Pour des transformations plus complexes, nous devons utiliser le verbe **statistics**. Une fonction **stat_** permet de définir des nouvelles variables à partir du jeu de données initial, il est ensuite possible de représenter ces nouvelles variables. Par exemple, la fonction **stat_bin**, qui est utilisée par défaut pour construire des histogrammes, calcule 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.

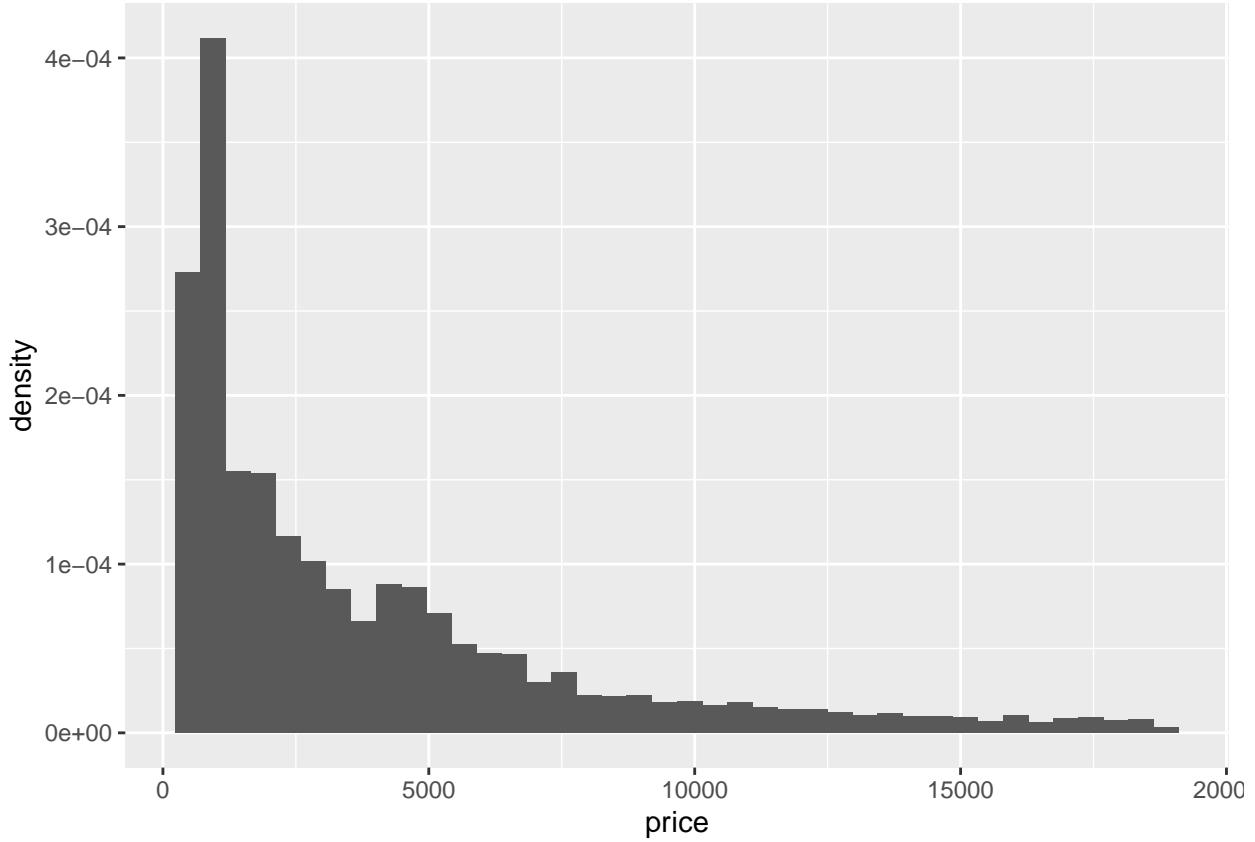
Par défaut **geom_histogram** fait appel à cette fonction **stat_bin** grâce à l'option **stat="bin"**. On visualise ainsi sur l'axe **y** le nombre d'observations dans chaque classe (la variable **count**).

```
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)
```



Les fonctions **stat_** peuvent être utilisées 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 exemples 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

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

Représenter cette distribution de probabilité avec un diagramme en barres.

Exercice 1.8 (Lissage). On étudie différentes façons de visualiser un 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.

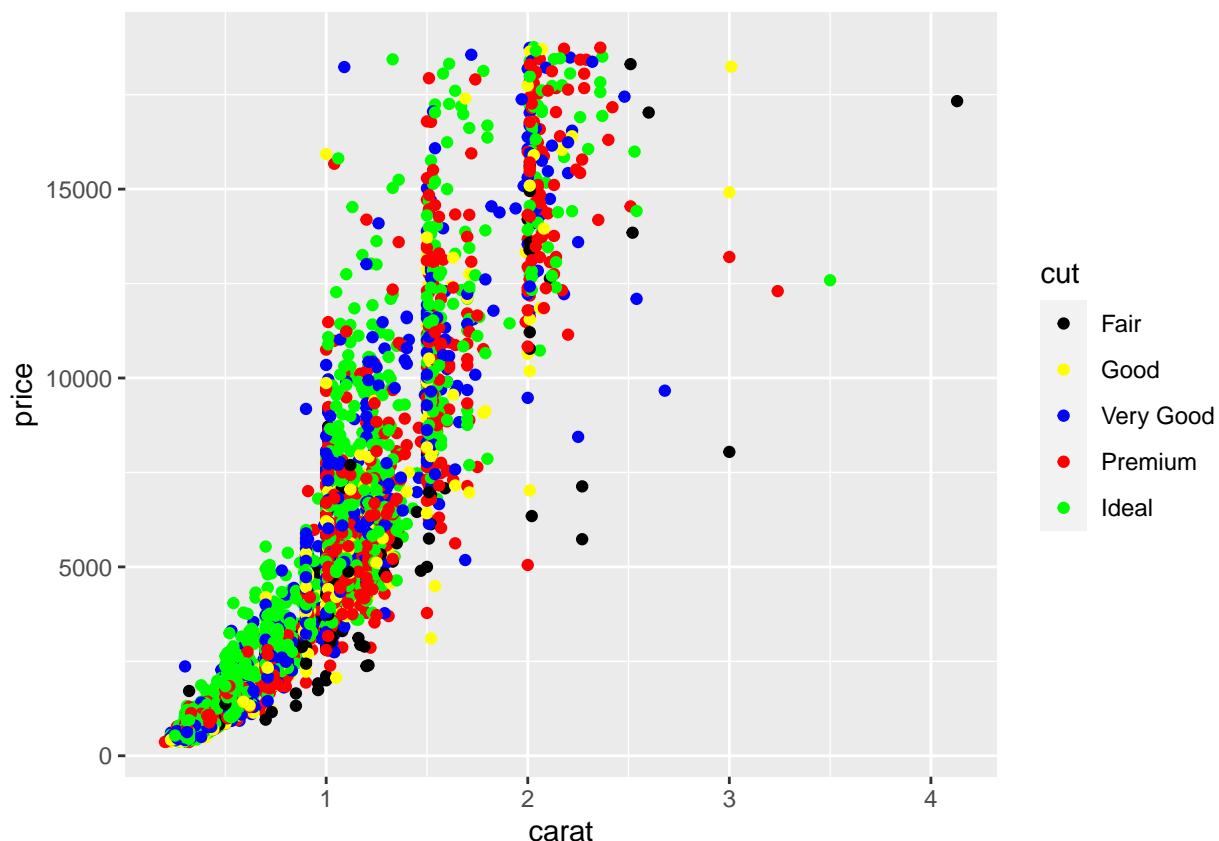
1.2.5 Scales

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 :

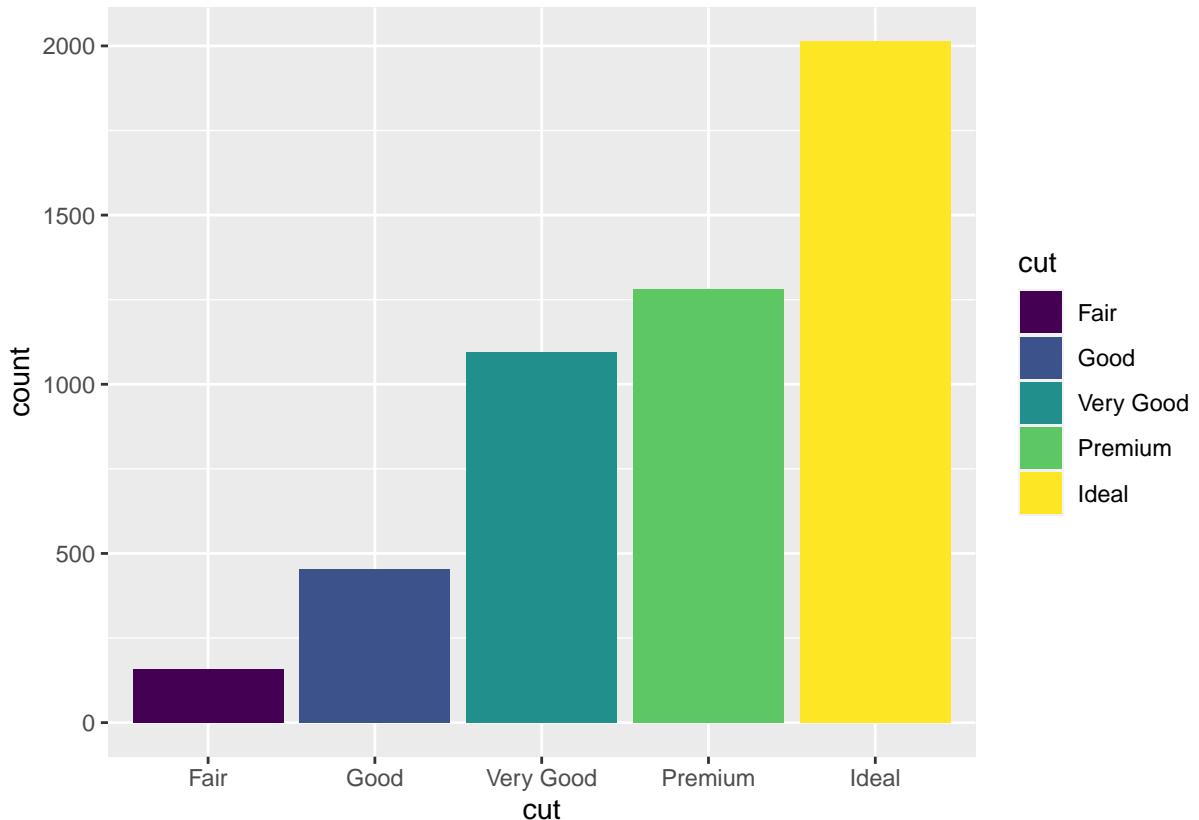
TABLE 3 : Exemples d'échelles

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 :

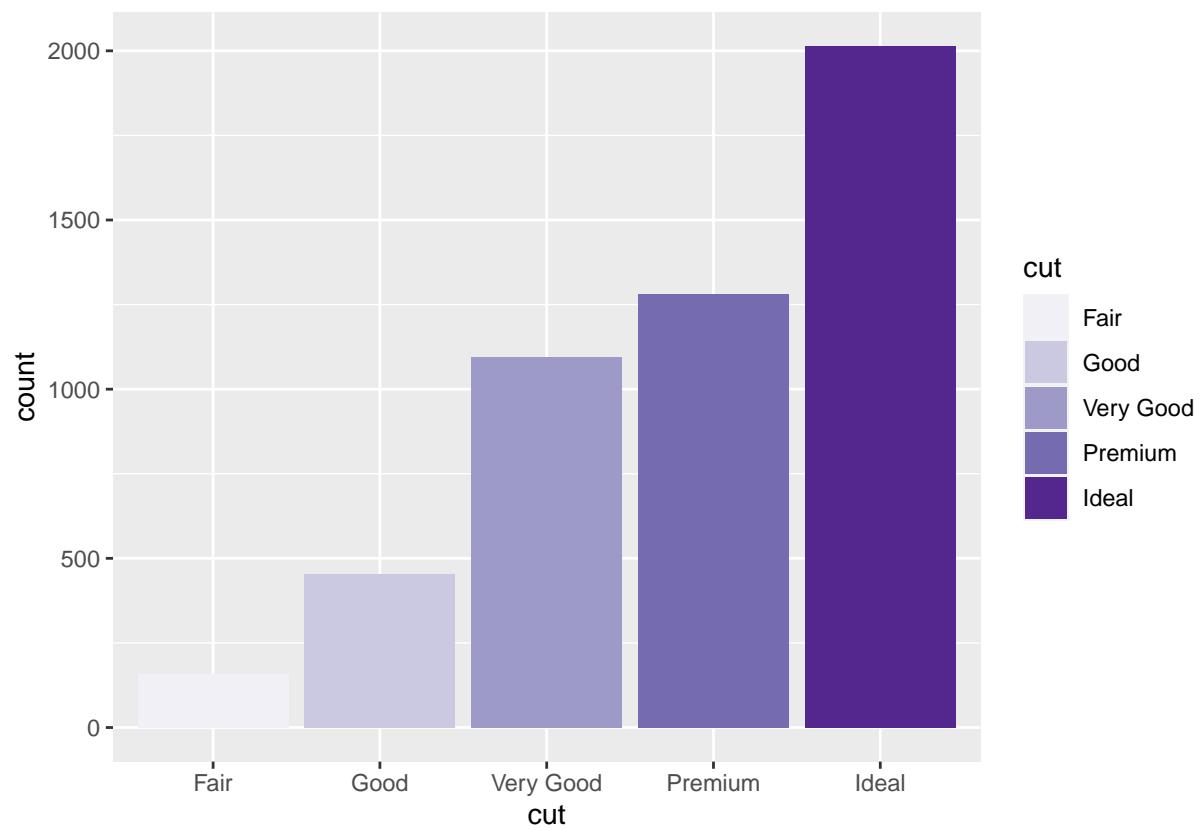
— Couleur dans un diagramme en barres

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



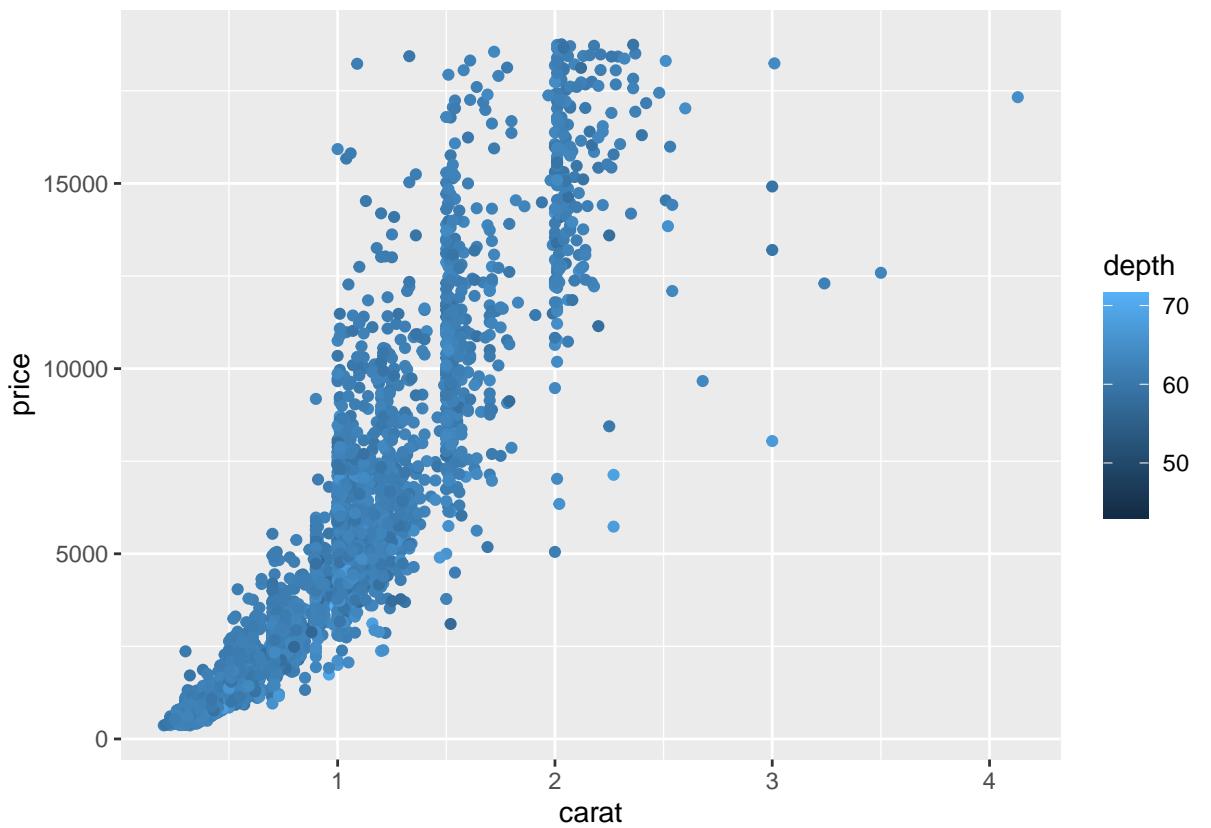
On change la couleur en utilisant la palette **Purples** :

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



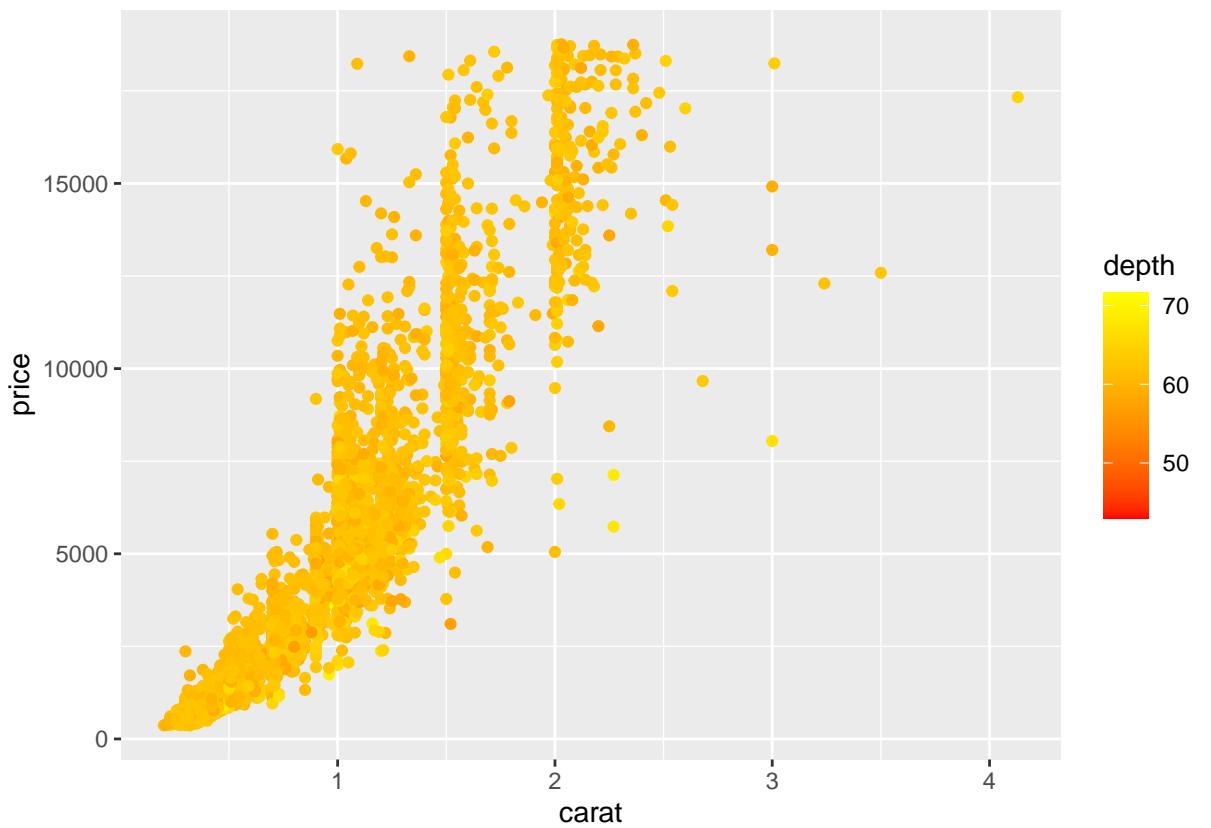
- Gradient de couleurs pour un nuage de points :

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



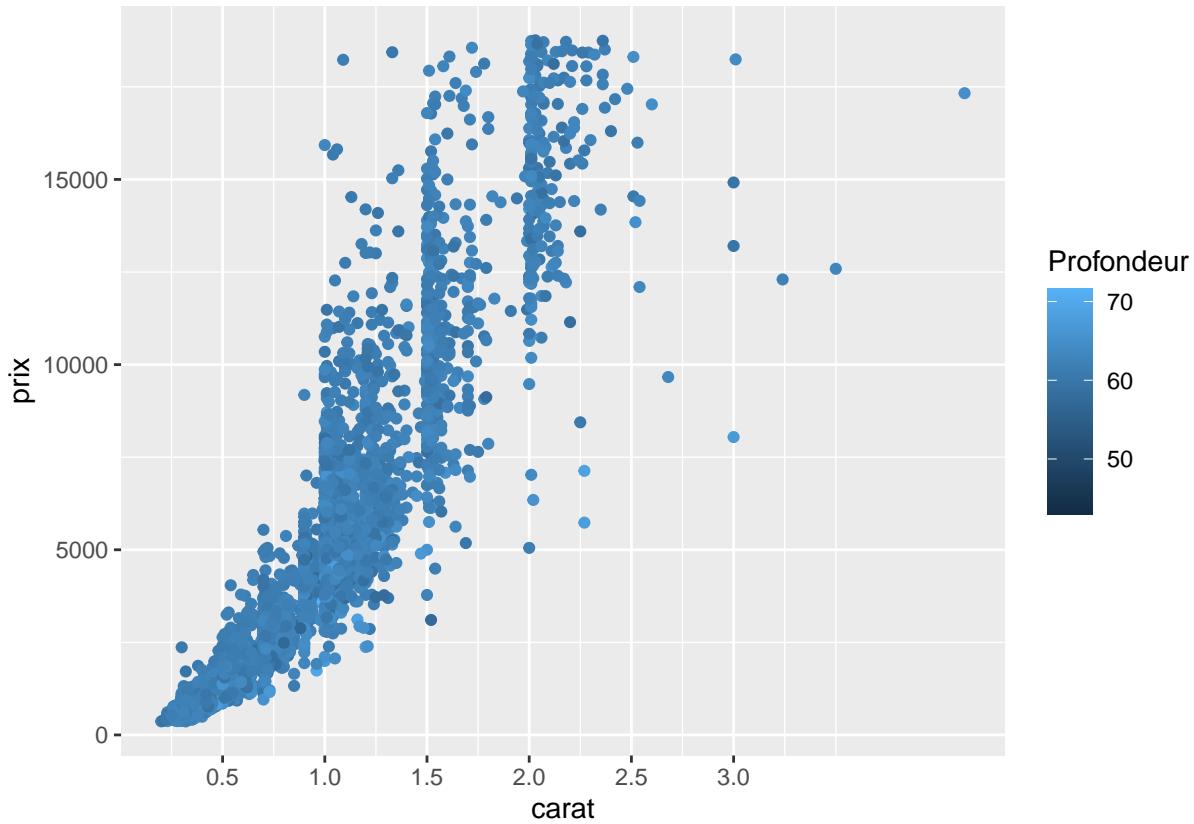
On change le gradient de couleur

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



— Modifications sur les axes

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



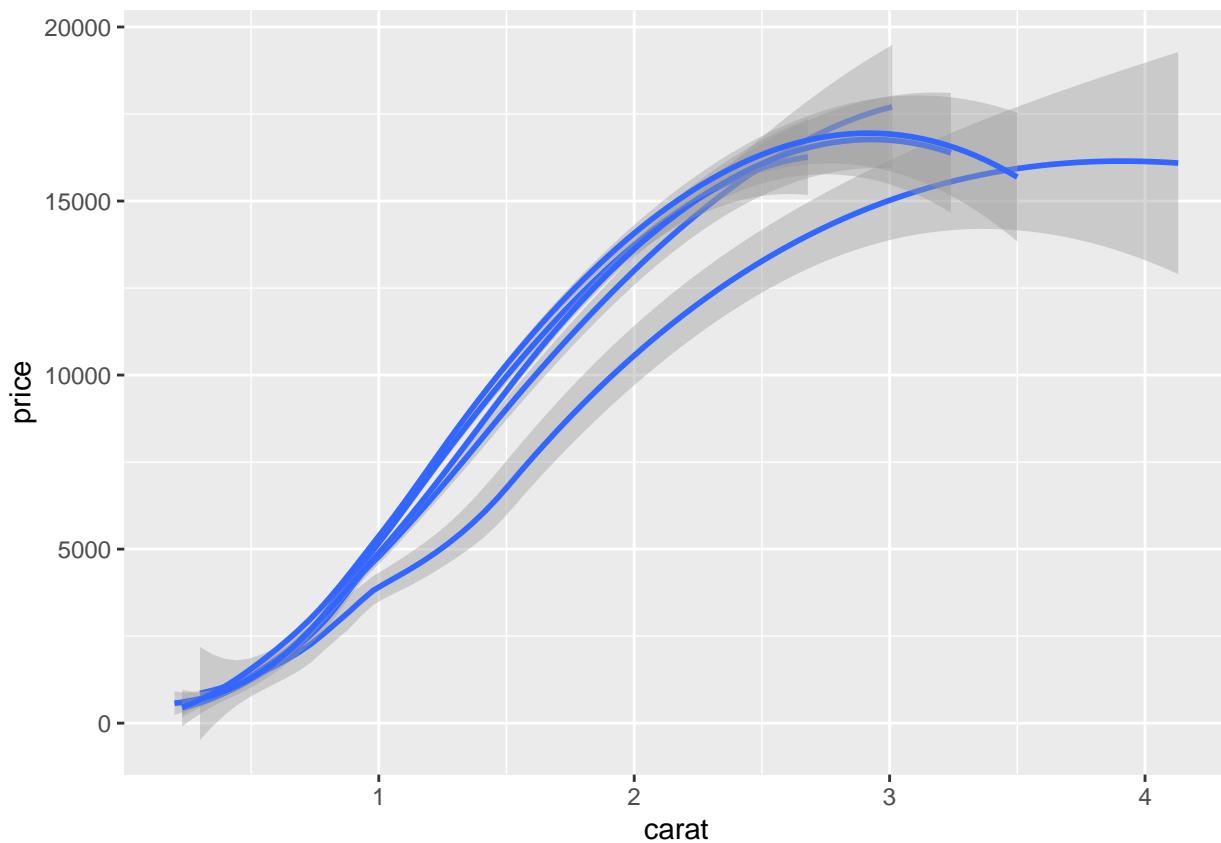
1.2.6 Group et facets

`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 le verbe `aes` ;
- visualisation de sous groupes sur des graphes différents, on utilise le verbe `facet_wrap` ou `facet_grid`.

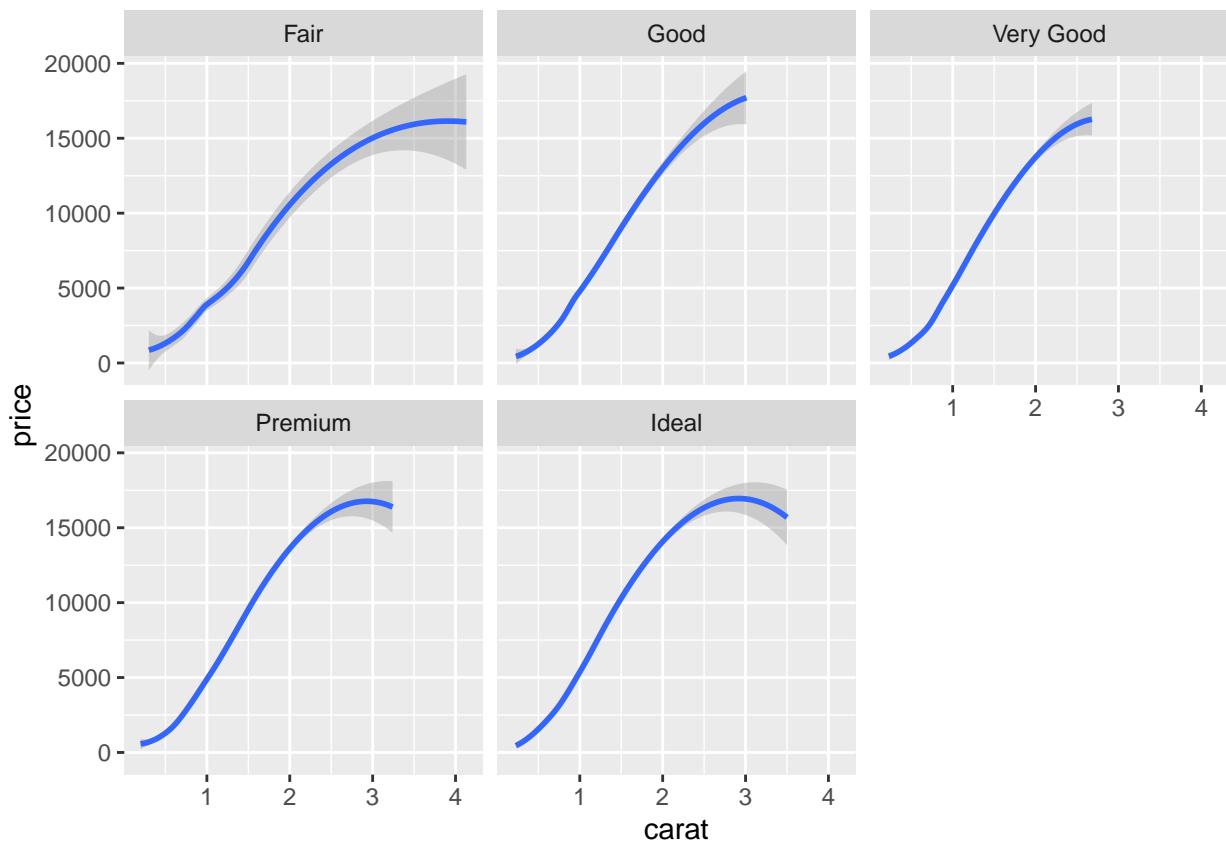
Représentons ici (sur le même graphe) le lissoir `price vs carat` pour chaque modalité de `cut`

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

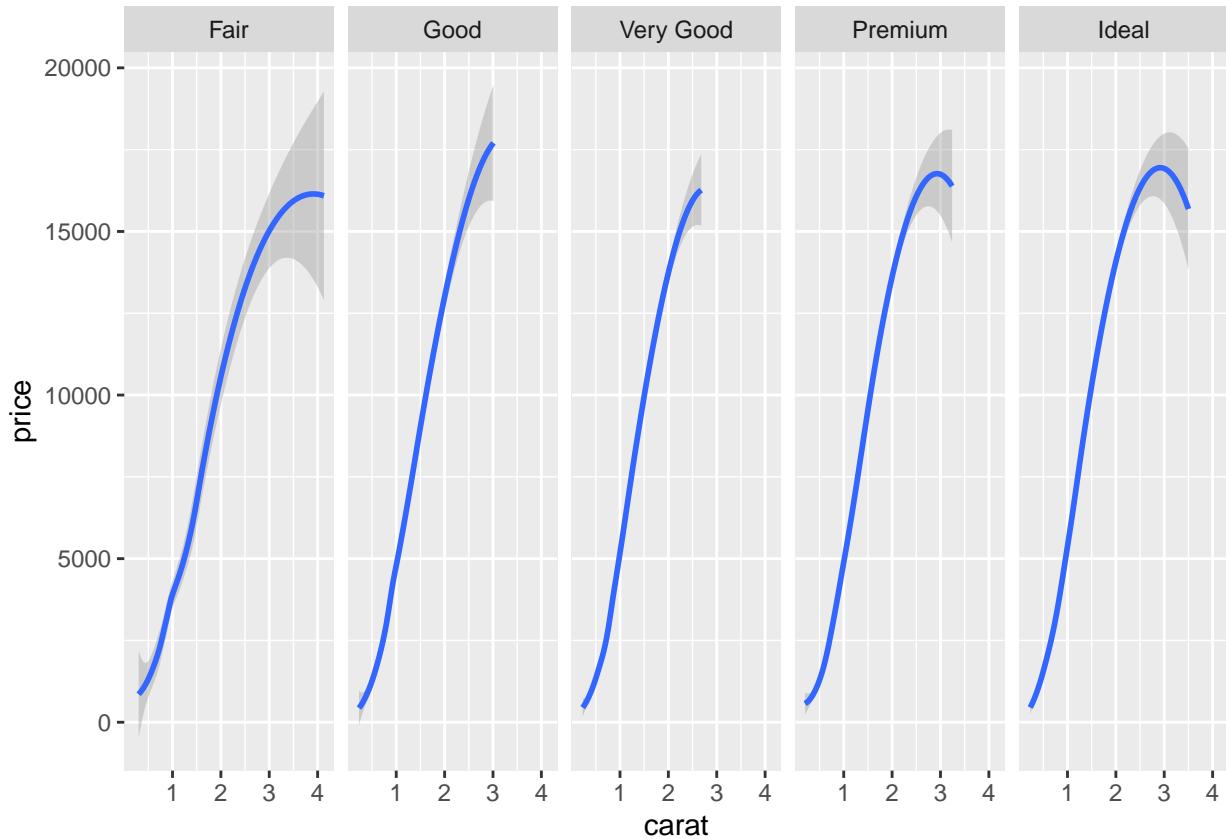


Pour obtenir cette représentation sur plusieurs fenêtres, on utilise

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

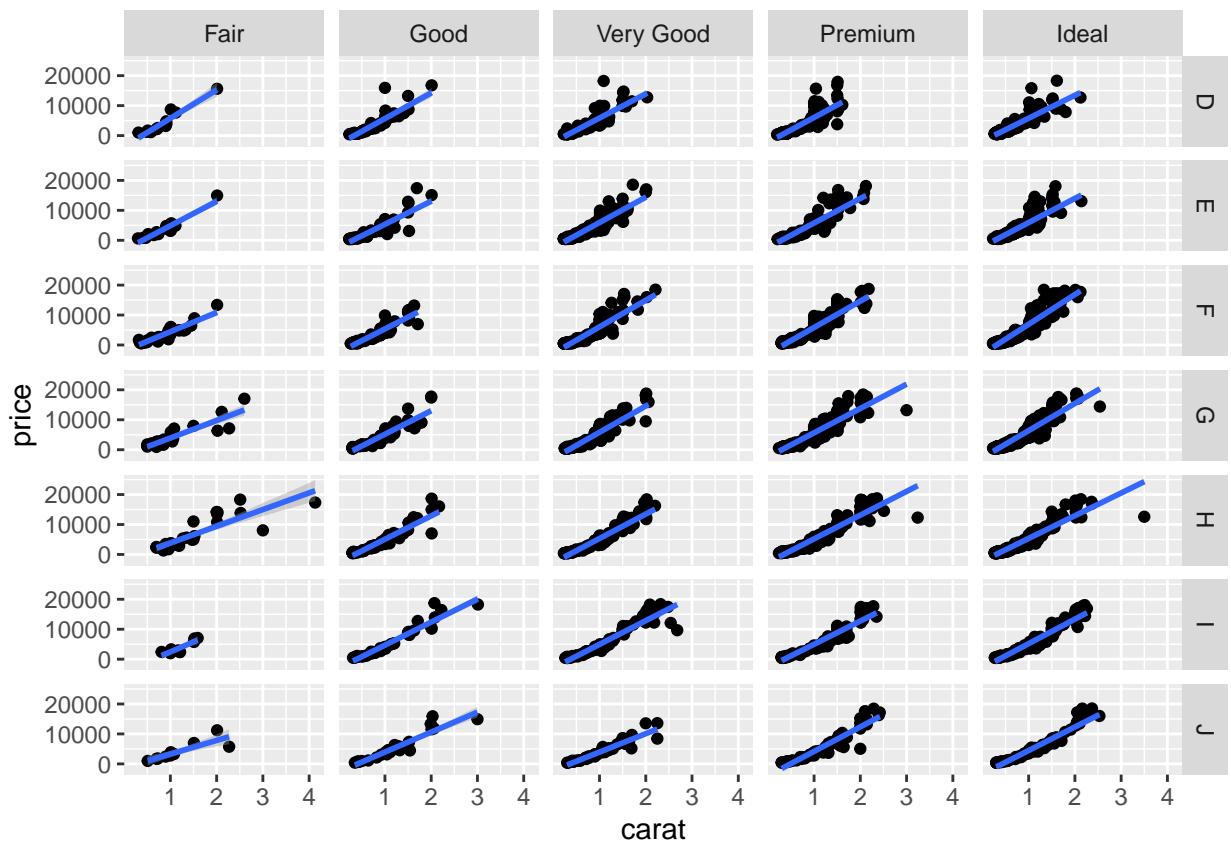


```
ggplot(diamonds2)+aes(x=carat,y=price)+  
  geom_smooth(method="loess") + facet_wrap(~cut,nrow=1)
```

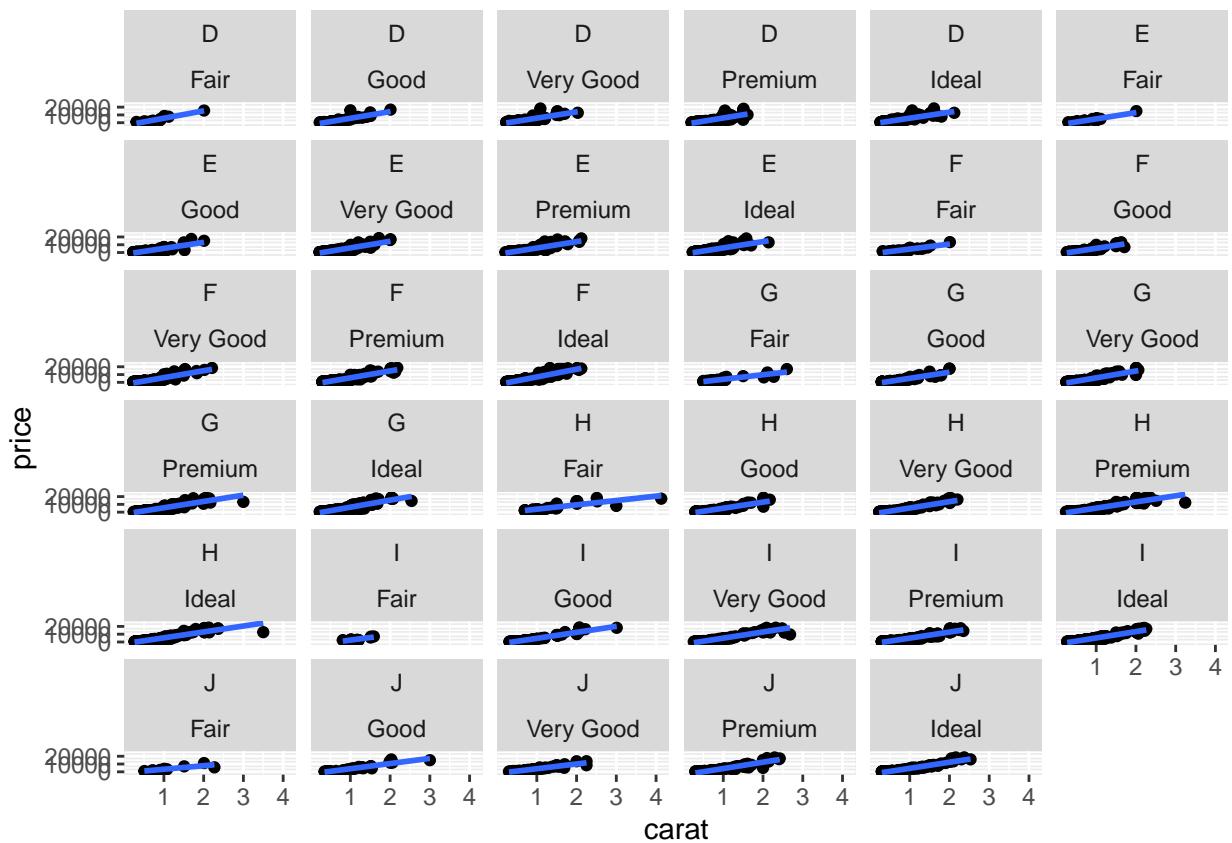


`facet_grid` et `facet_wrap` font des choses proches mais divisent la fenêtre d'une façon différente :

```
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)
```



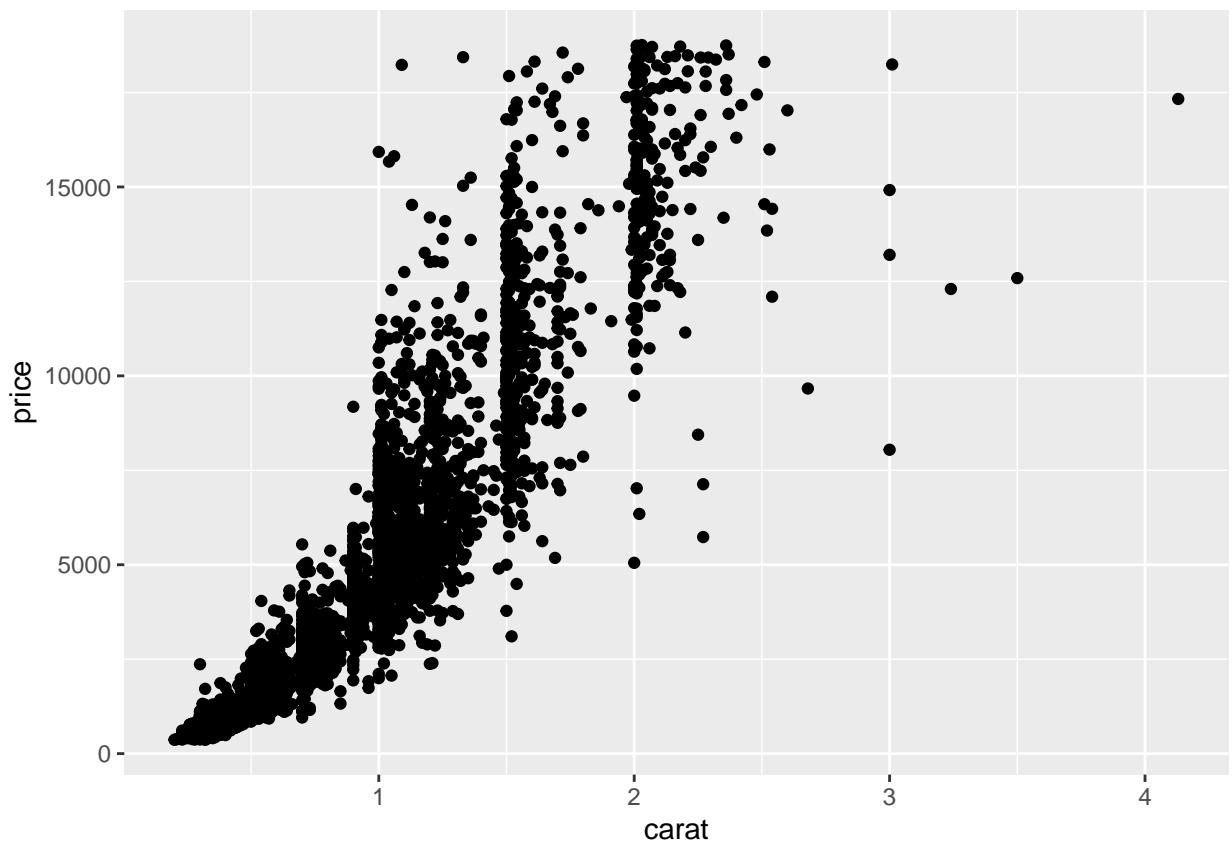
1.3 Compléments

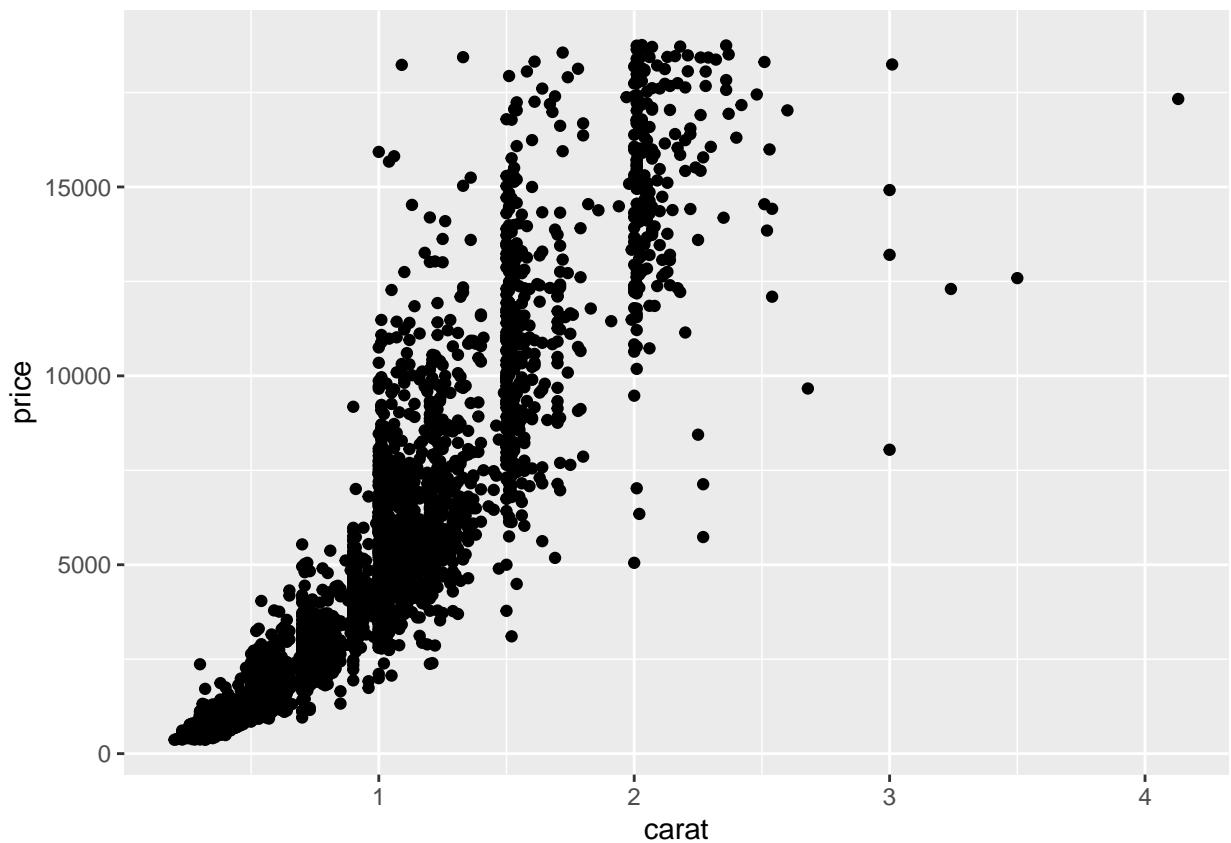
La syntaxe **ggplot** est définie selon le schéma :

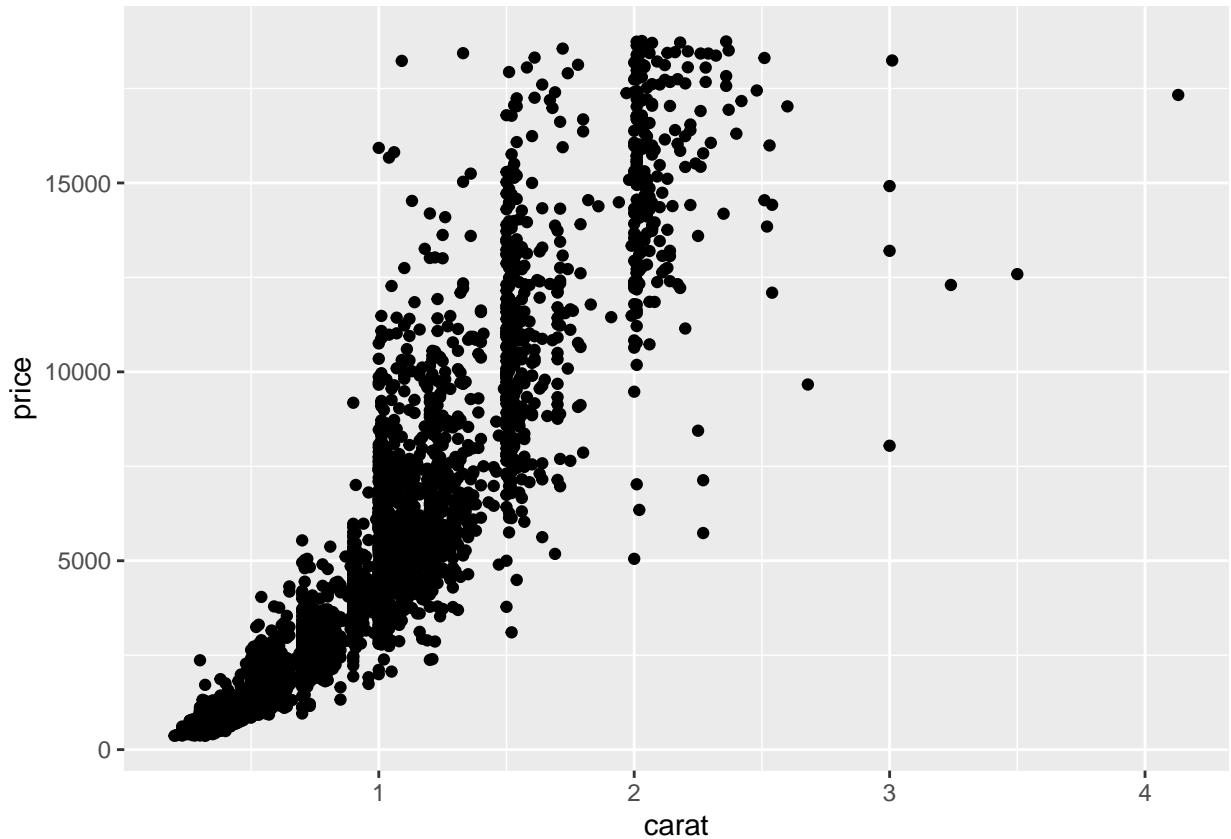
```
ggplot() + aes() + geom_() + scale_()
```

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()
```



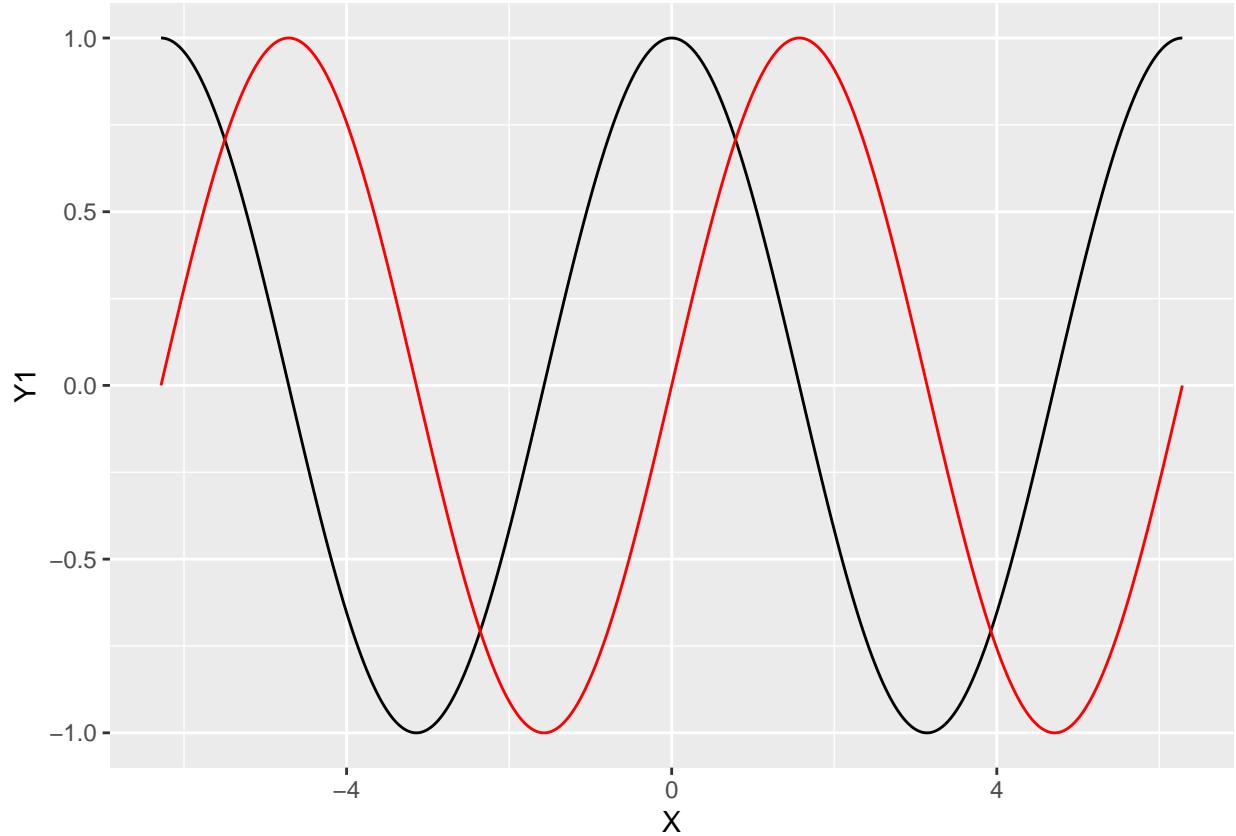




Ceci peut se révéler très utile lorsqu'on utilise des **aes** différents dans les **geom_**.

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

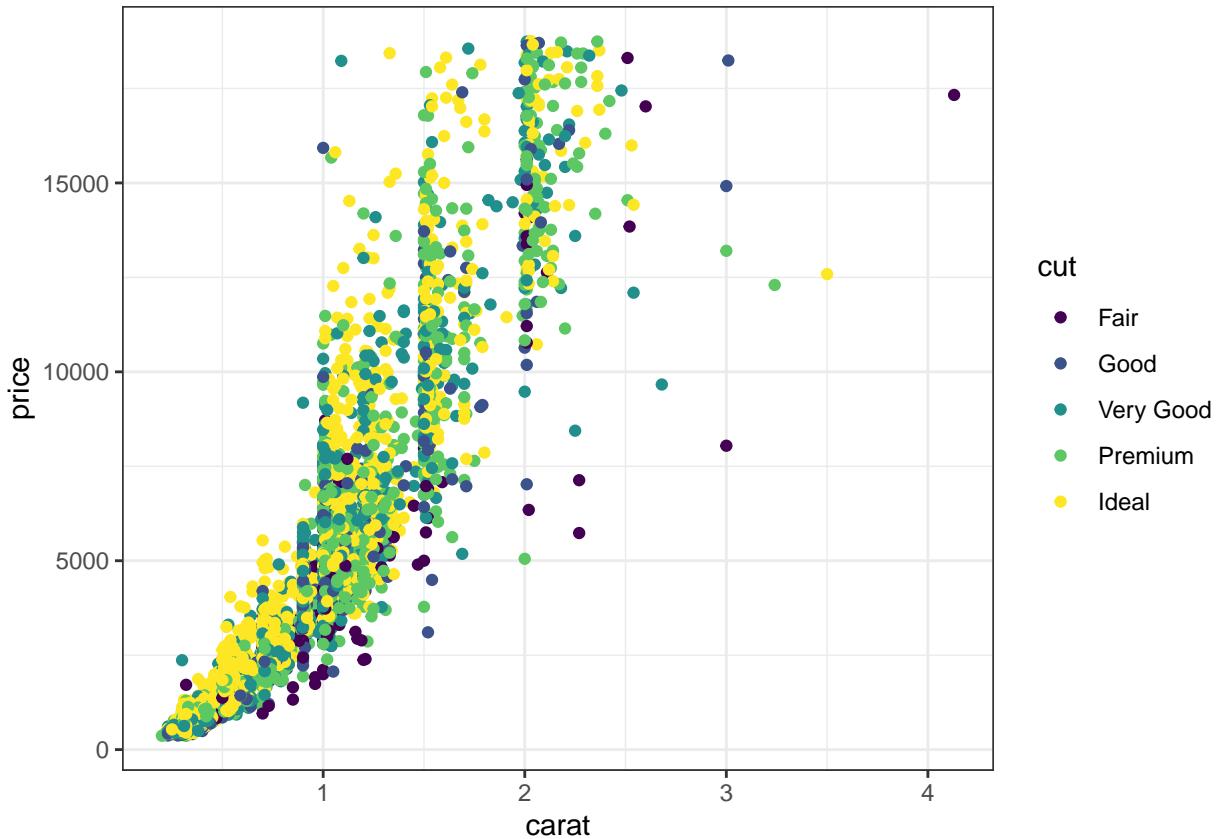
```
X <- seq(-2*pi,2*pi,by=0.001)
Y1 <- cos(X)
Y2 <- sin(X)
donnees1 <- data.frame(X,Y1)
donnees2 <- data.frame(X,Y2)
ggplot(donnees1)+geom_line(aes(x=X,y=Y1))+
  geom_line(data=donnees2,aes(x=X,y=Y2),color="red")
```

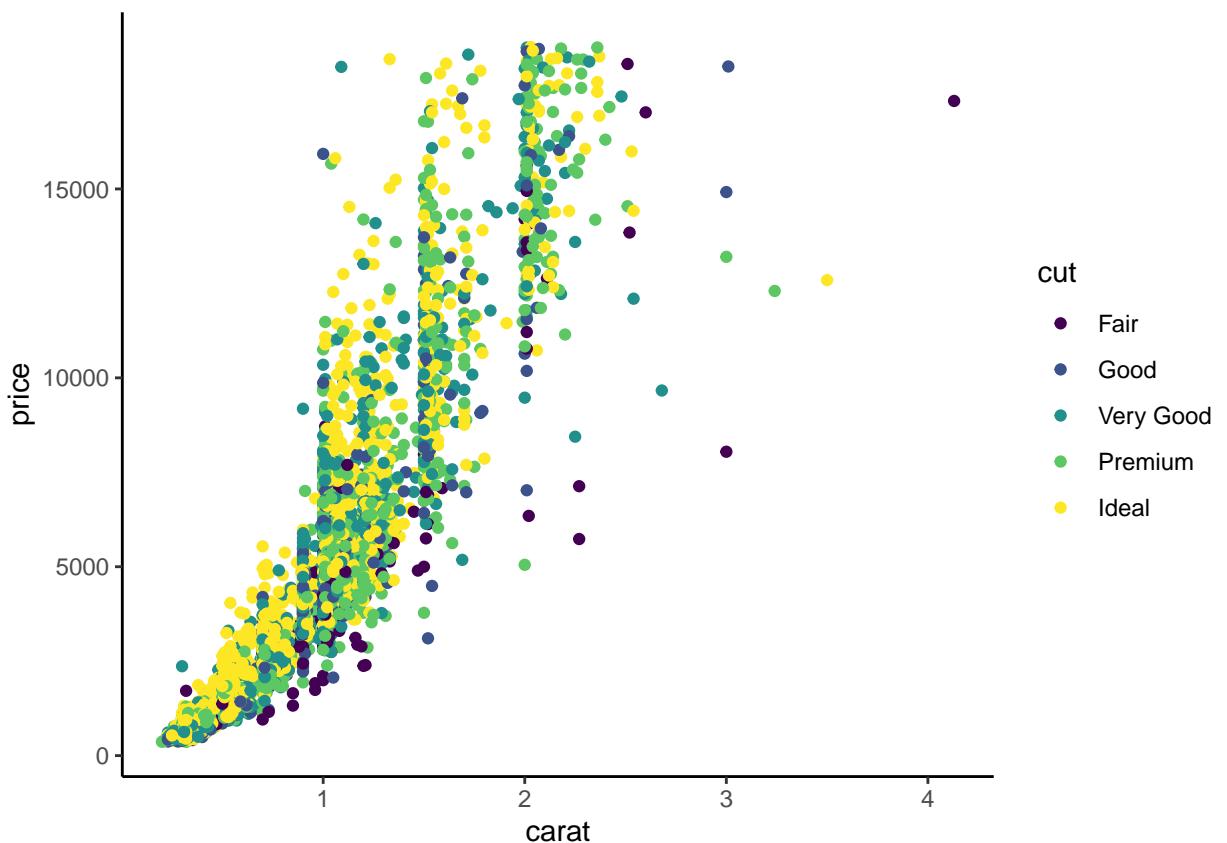


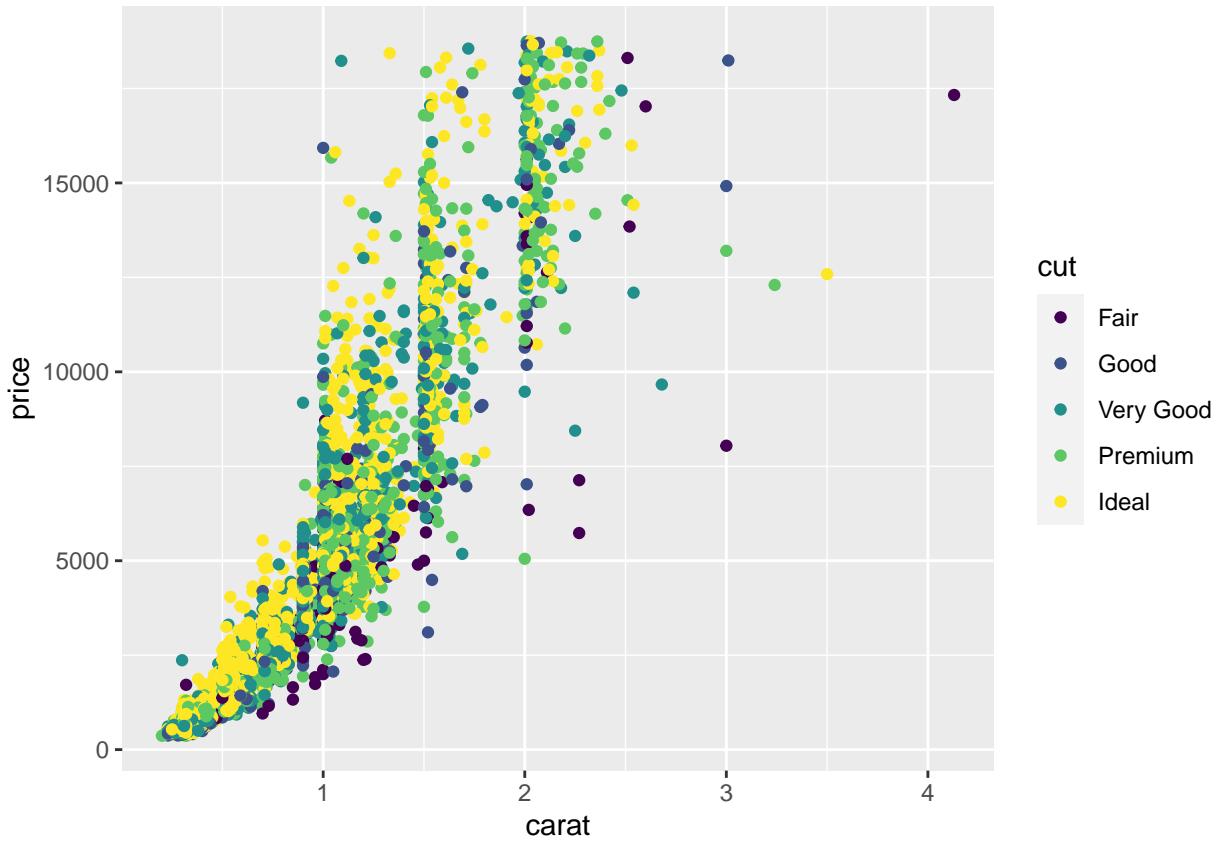
Il existe d'autres fonctions **ggplot** :

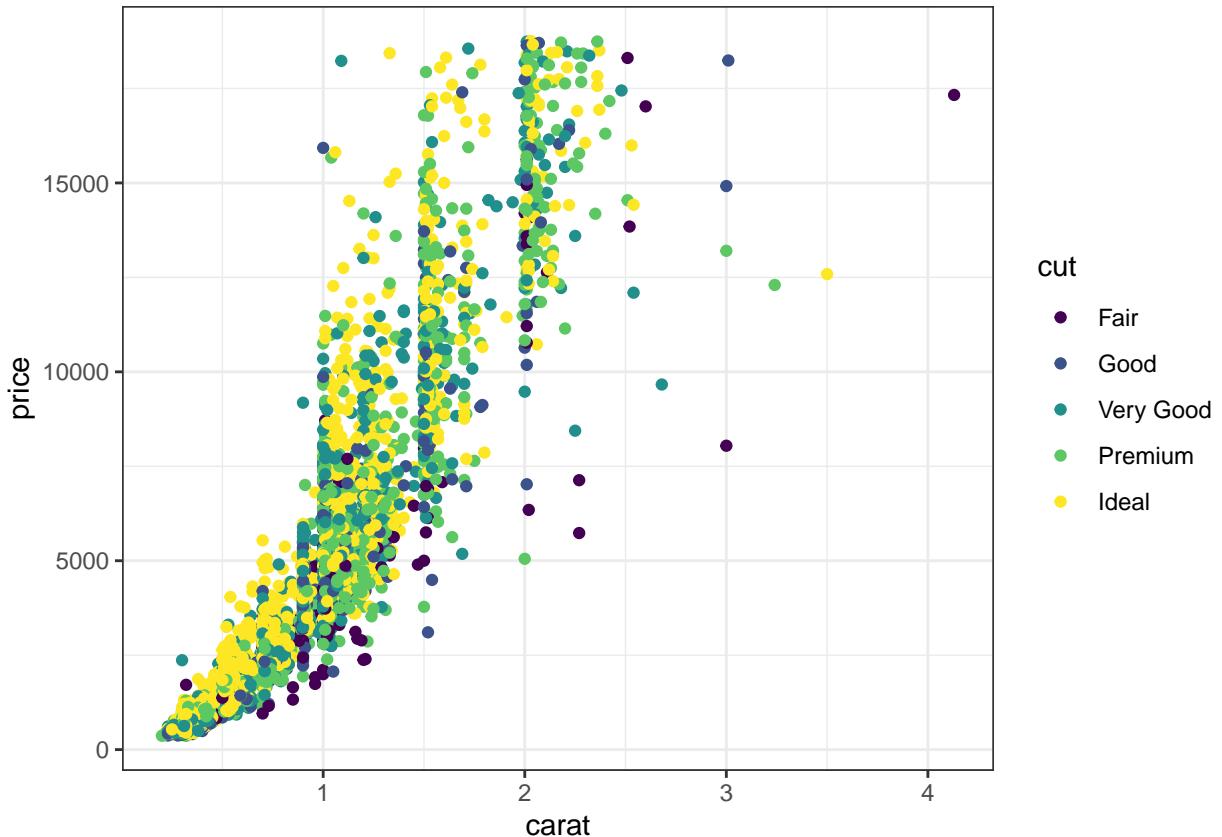
- **ggtitle** pour ajouter un titre.
- **ggsave** pour sauver un graphe.
- **theme_** pour changer le theme du graphe.

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









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

1.4 Quelques exercices supplémentaires

Exercice 1.9 (Fonctions cosinus et sinus). L'objectif est de visualiser les fonctions **sinus** et **cosinus** de plusieurs façons.

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

Exercice 1.10 (Différents graphes). On considère les données `mtcars`

```
data(mtcars)
summary(mtcars)

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

```

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

```

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

Exercice 1.11 (Résidus pour régression simple). On souhaite visualiser les résidus dans un modèle de régression simple.

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

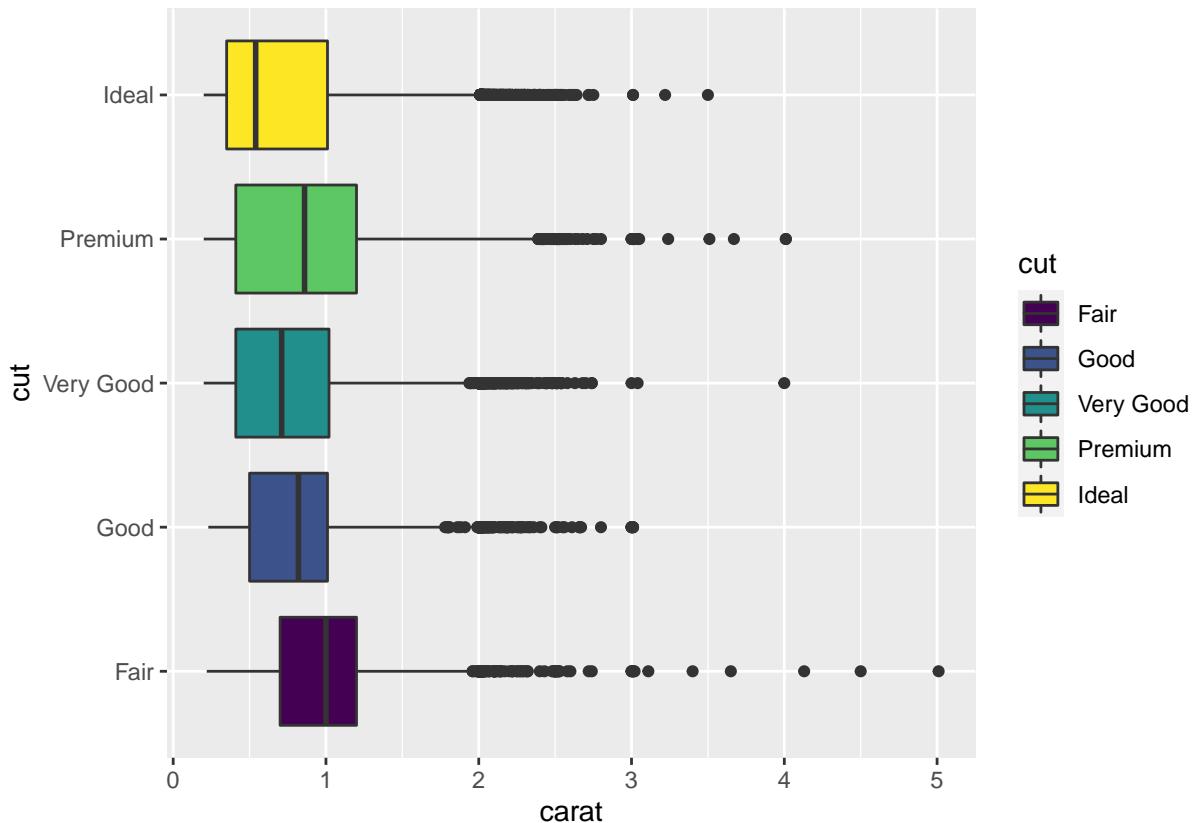
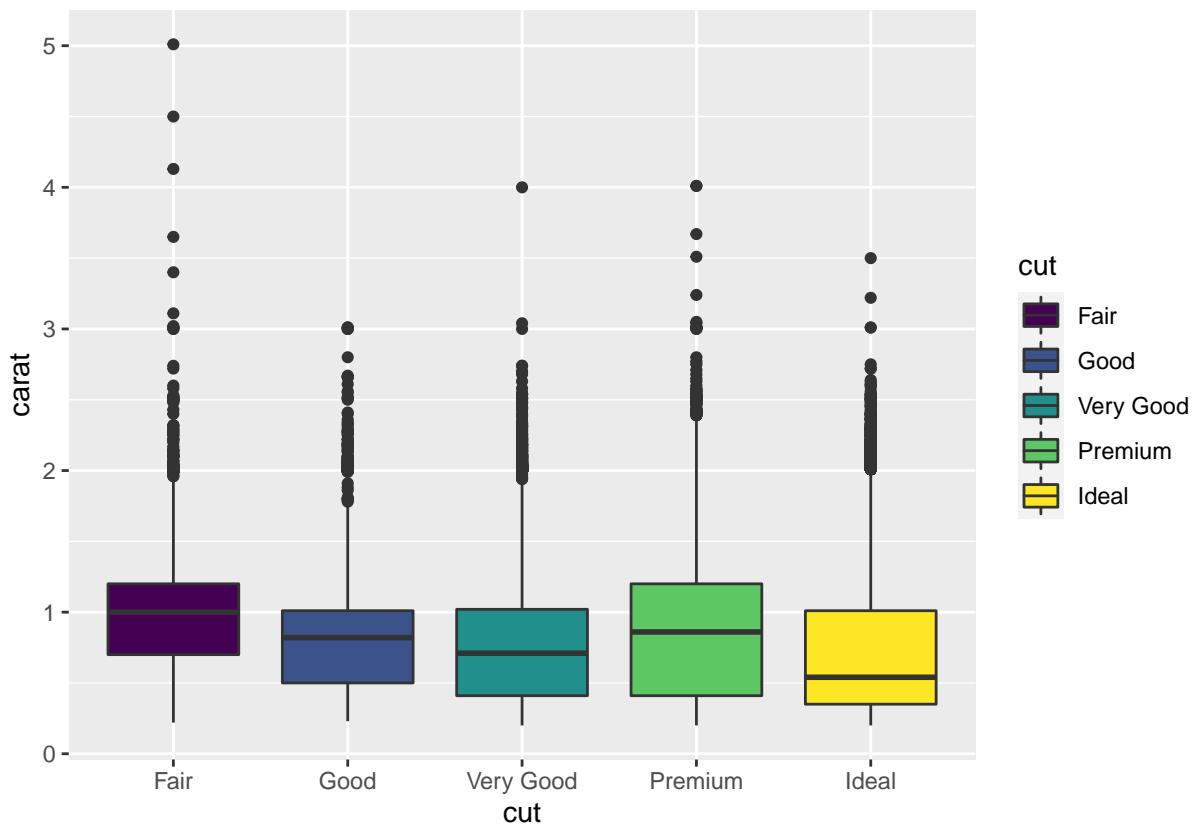
$$y_i = 3 + x_i + \varepsilon_i$$

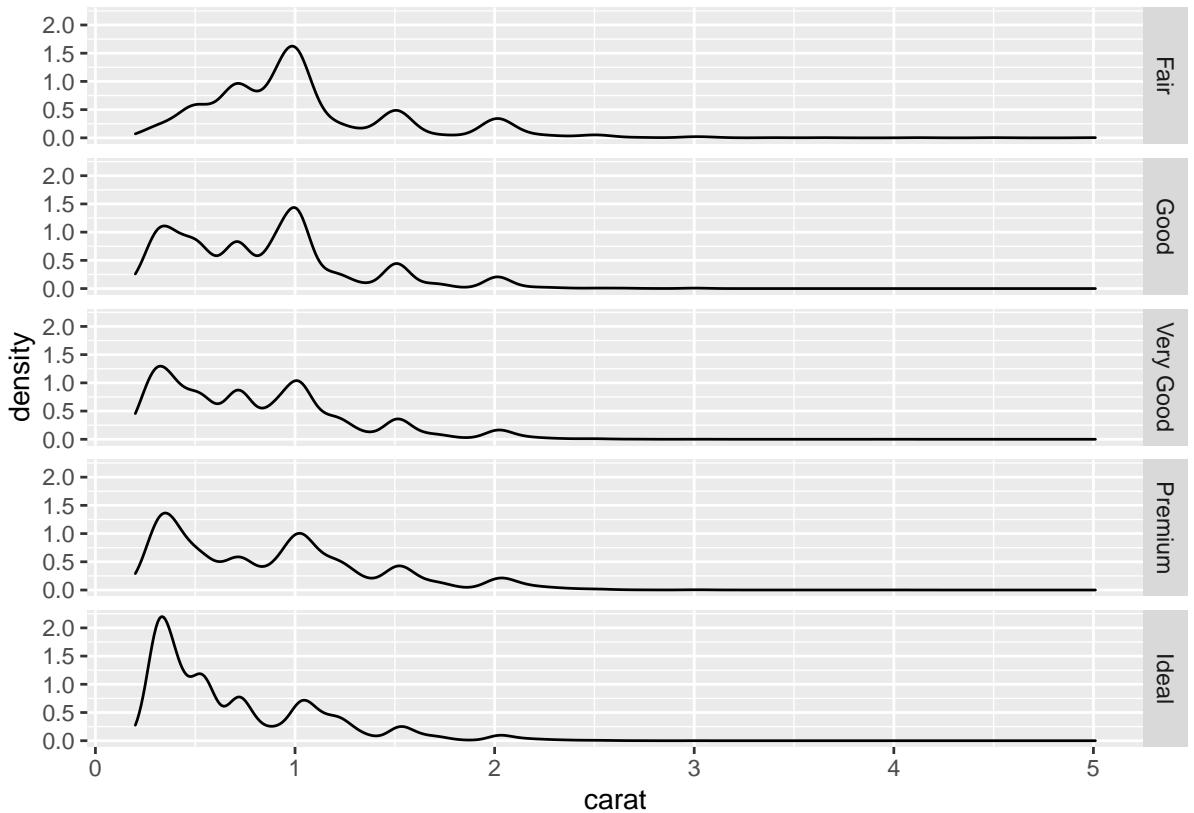
où les x_i sont i.i.d. de loi uniforme sur $[0, 1]$ et les ε_i sont i.i.d. de loi gaussienne $N(0, 0.2^2)$ (utiliser `runif` et `rnorm`).

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

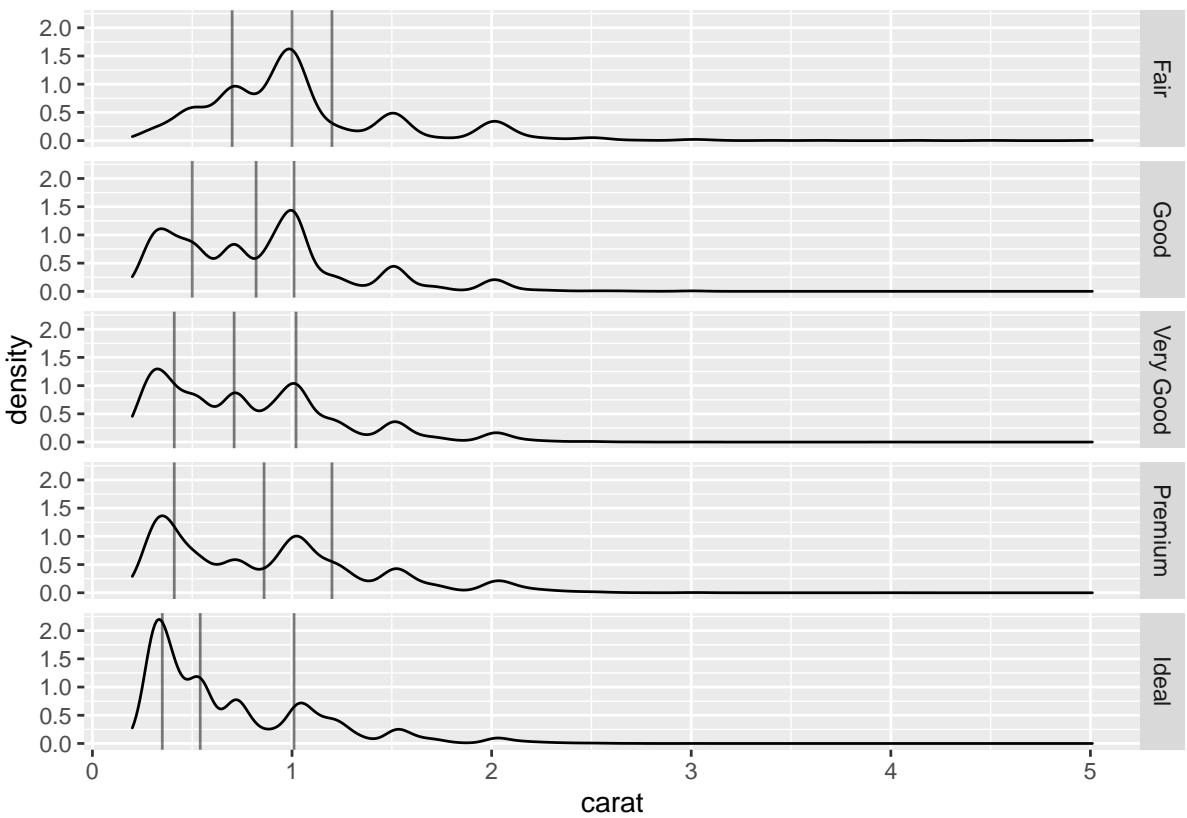
Exercice 1.12 (Challenge). On considère les données **diamonds**.

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

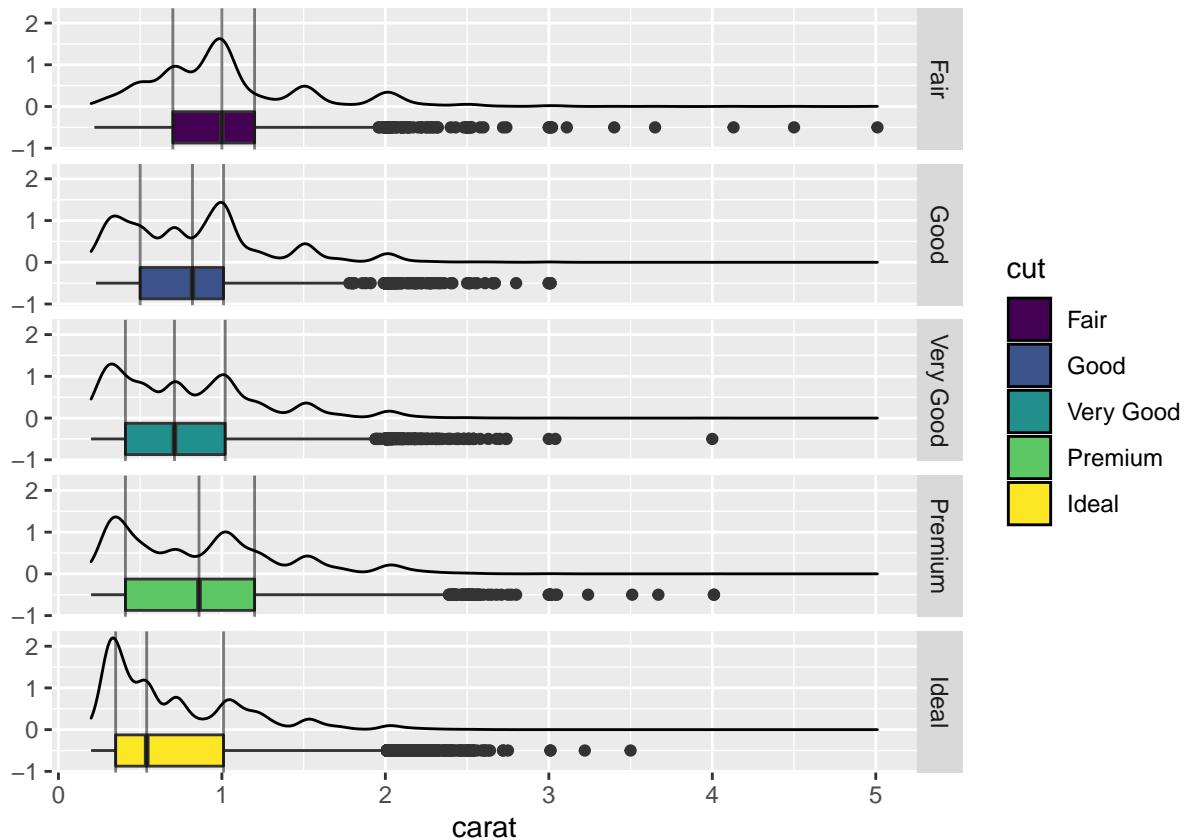




2. Ajouter sur le troisième graphe les quartiles de la variable **carat** pour chaque valeur de **cut**. On utilisera une ligne verticale.



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



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.

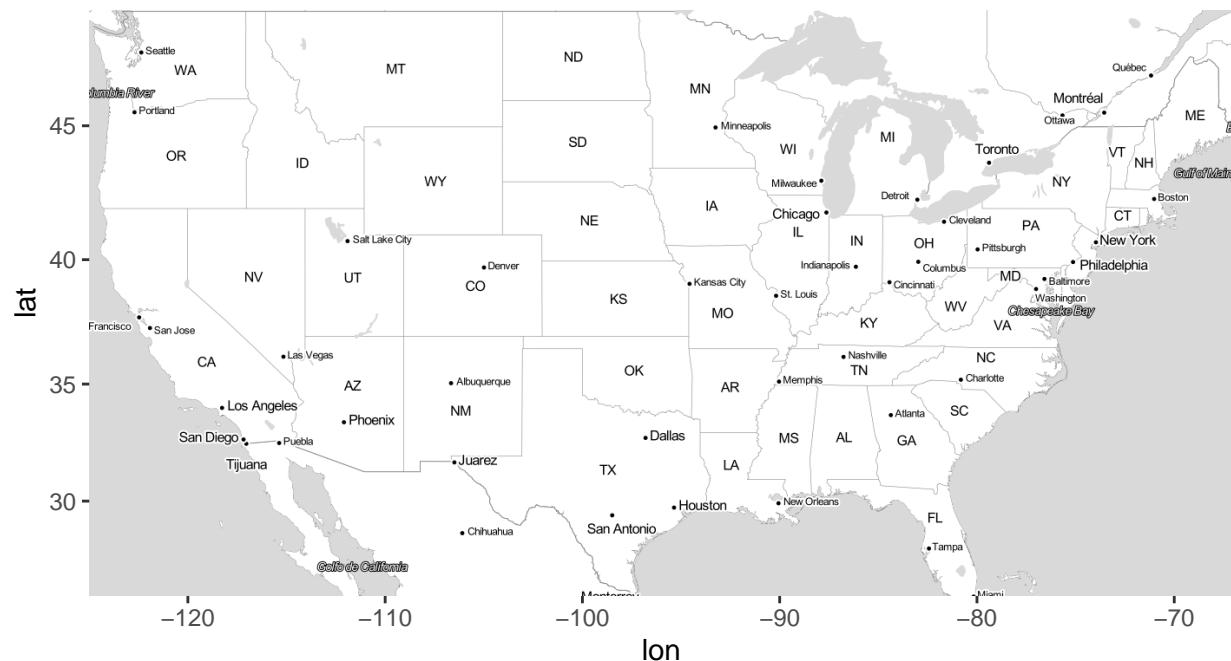
De nombreux packages **R** permettent d'obtenir des cartes. Dans cette partie, on s'intéressera aux packages **ggmap** et **sf** pour les cartes statiques et **leaflet** pour les cartes interactives.

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.

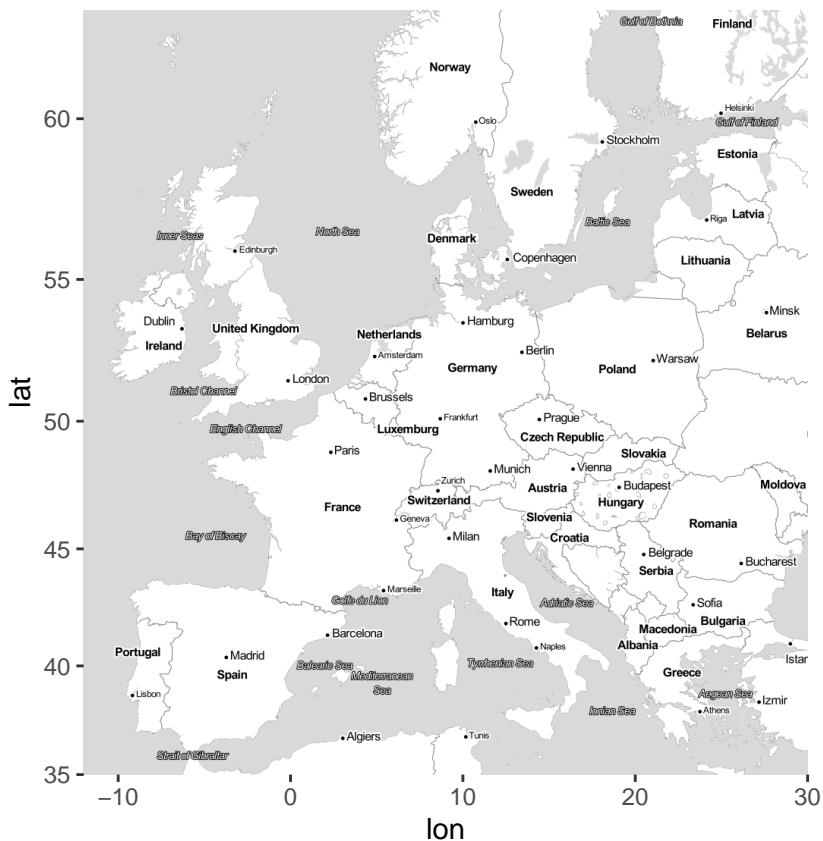
ggmap permet de récupérer facilement des fonds de carte. Par exemple :

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



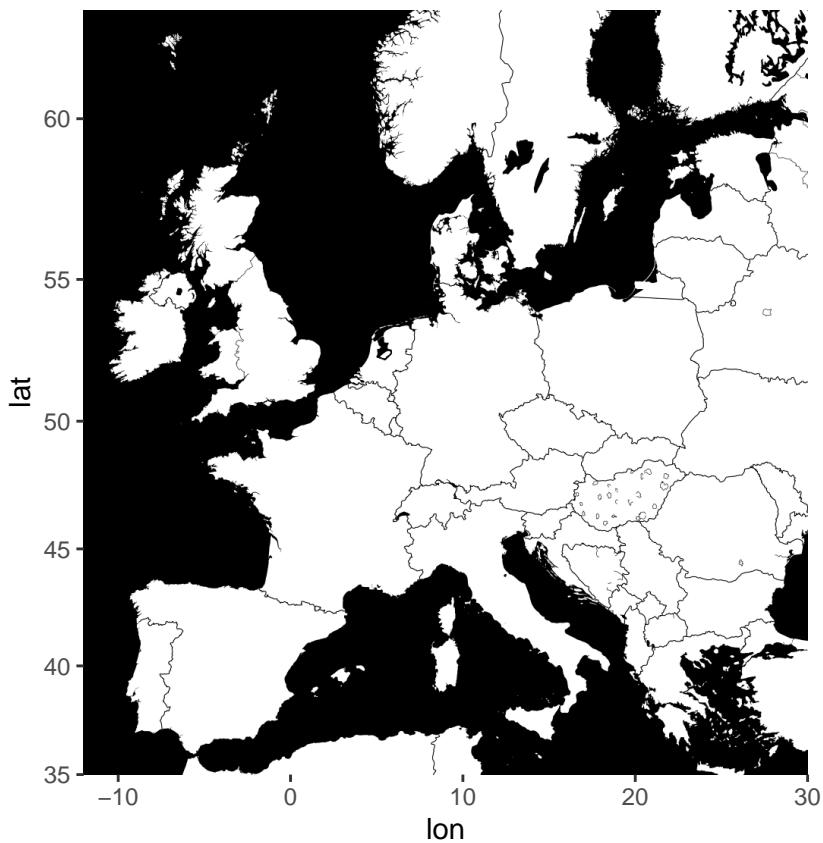
Pour l'Europe on fait

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



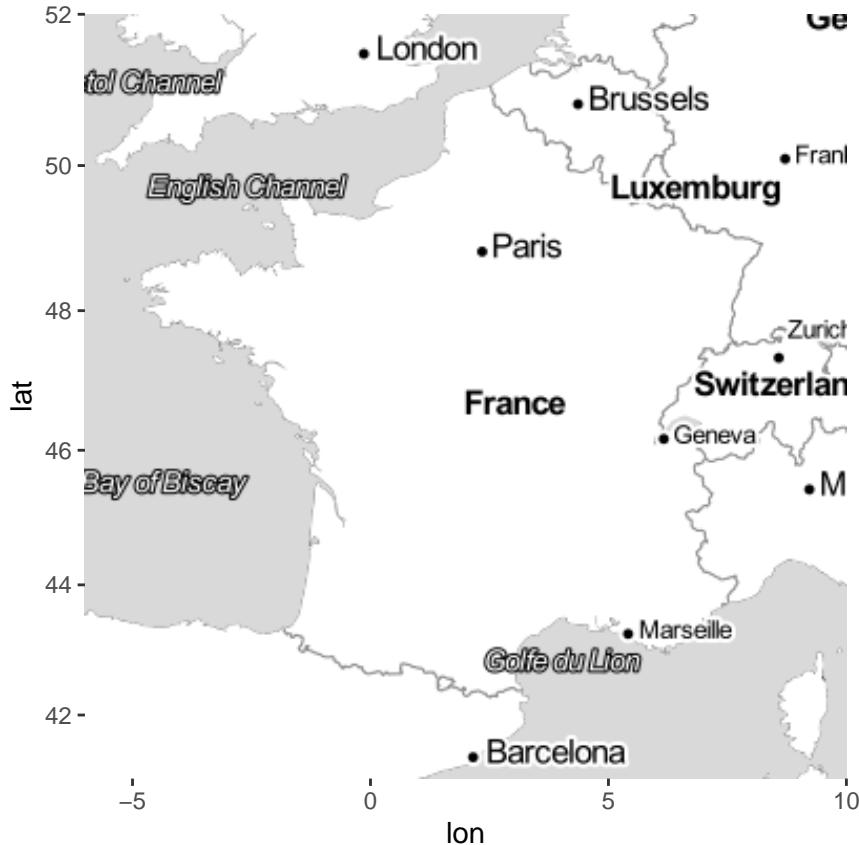
On peut également changer le fond de carte

```
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 2.320041 48.85889
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.
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.

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
Bounding box:  xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
Geodetic CRS:  NAD27
First 10 features:
#> #>   AREA PERIMETER CNTY_ CNTY_ID      NAME  FIPS FIPSNO
#> #> 1 0.114     1.442  1825    1825      Ashe 37009 37009
#> #> 2 0.061     1.231  1827    1827  Alleghany 37005 37005
#> #> 3 0.143     1.630  1828    1828    Surry 37171 37171

```

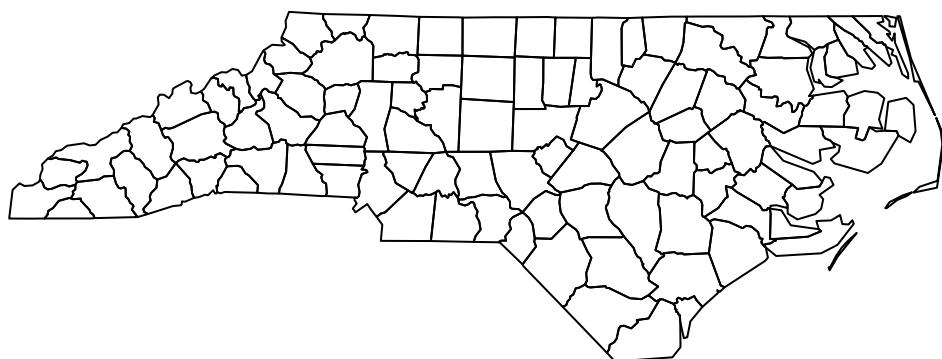
```

4 0.070    2.968  1831    1831    Currituck 37053  37053
5 0.153    2.206  1832    1832    Northampton 37131  37131
6 0.097    1.670  1833    1833    Hertford 37091  37091
7 0.062    1.547  1834    1834    Camden 37029  37029
8 0.091    1.284  1835    1835    Gates 37073  37073
9 0.118    1.421  1836    1836    Warren 37185  37185
10 0.124   1.428  1837    1837    Stokes 37169  37169
CRESS_ID BIR74 SID74 NWBIR74 BIR79 SID79 NWBIR79
1      5 1091    1     10 1364    0     19
2      3 487     0     10 542     3     12
3     86 3188    5    208 3616    6     260
4     27 508     1    123 830     2     145
5     66 1421    9   1066 1606    3    1197
6     46 1452    7    954 1838    5    1237
7     15 286     0    115 350     2     139
8     37 420     0    254 594     2     371
9     93 968     4    748 1190    2     844
10    85 1612    1    160 2038    5     176
geometry
1 MULTIPOLYGON (((-81.47276 3...
2 MULTIPOLYGON (((-81.23989 3...
3 MULTIPOLYGON (((-80.45634 3...
4 MULTIPOLYGON (((-76.00897 3...
5 MULTIPOLYGON (((-77.21767 3...
6 MULTIPOLYGON (((-76.74506 3...
7 MULTIPOLYGON (((-76.00897 3...
8 MULTIPOLYGON (((-76.56251 3...
9 MULTIPOLYGON (((-78.30876 3...
10 MULTIPOLYGON (((-80.02567 3...

```

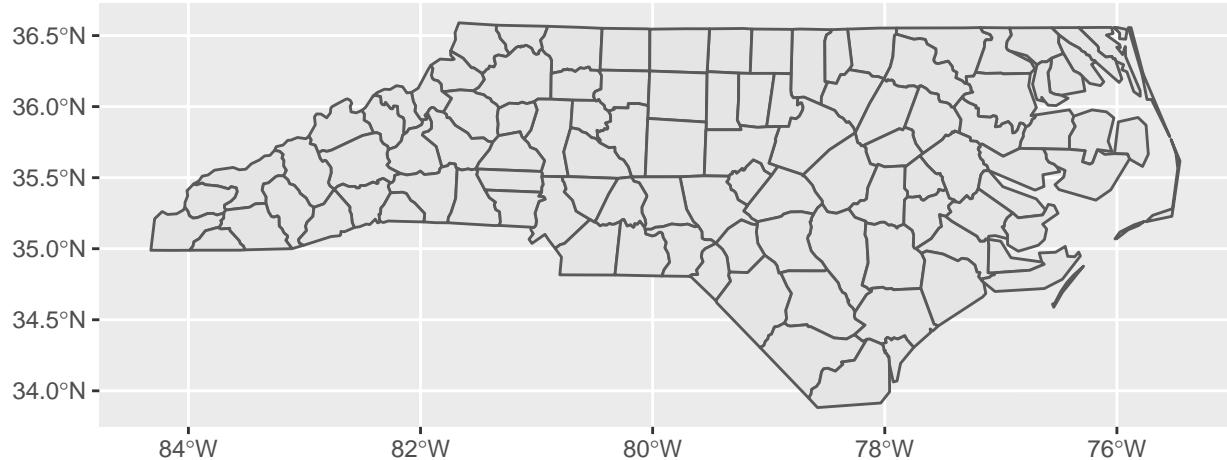
Ces données contiennent des informations sur les morts subites de nourrissons dans des villes de Caroline du Nord. On remarque que l'objet `nc` est au format `sf` et `data.frame`. On peut donc l'utiliser comme un `data.frame` classique. Le format `sf` permet l'ajout d'une colonne particulière (`geometry`) 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))
```



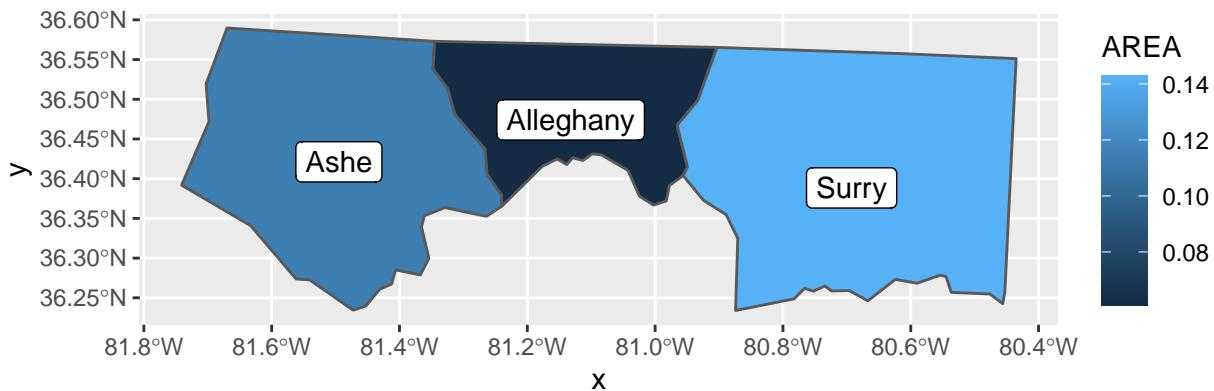
ou en utilisant le verbe `geom_sf` si on veut faire du `ggplot`

```
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() %>% st_cast(to="POINT")  
coord.ville1.nc  
Geometry set for 79 features  
Geometry type: POINT
```

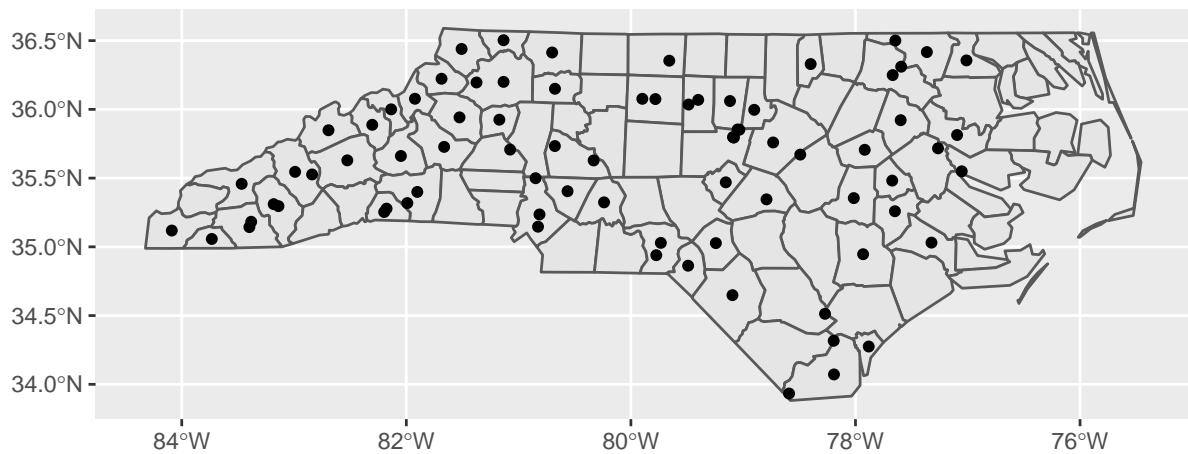
```
Dimension:      XY
Bounding box:  xmin: -84.08862 ymin: 33.93323 xmax: -77.01151 ymax: 36.503
CRS:           NA
First 5 geometries:
```

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)
```

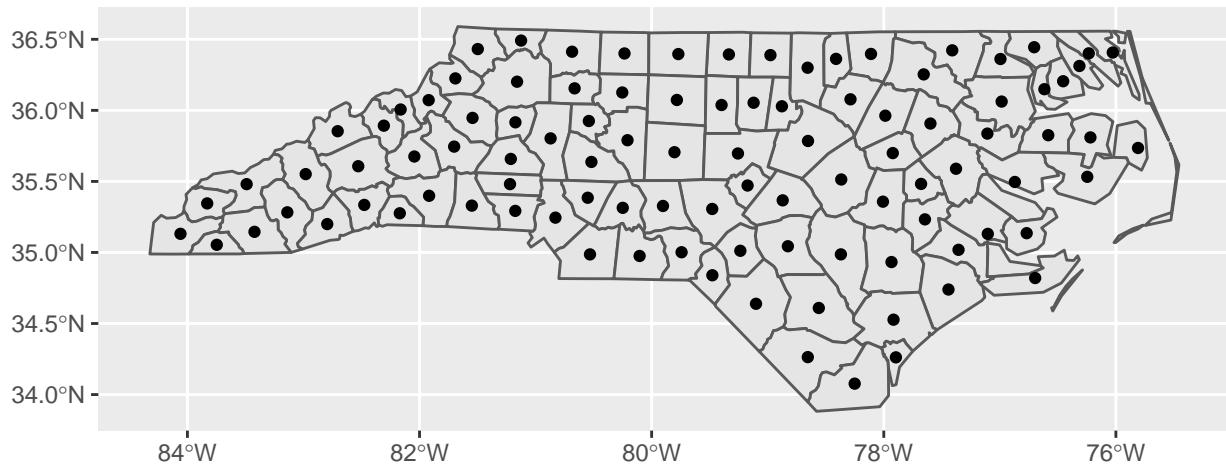


Le package **sf** possède également des fonctions très utiles pour traiter des données cartographiques, on peut citer par exemple :

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

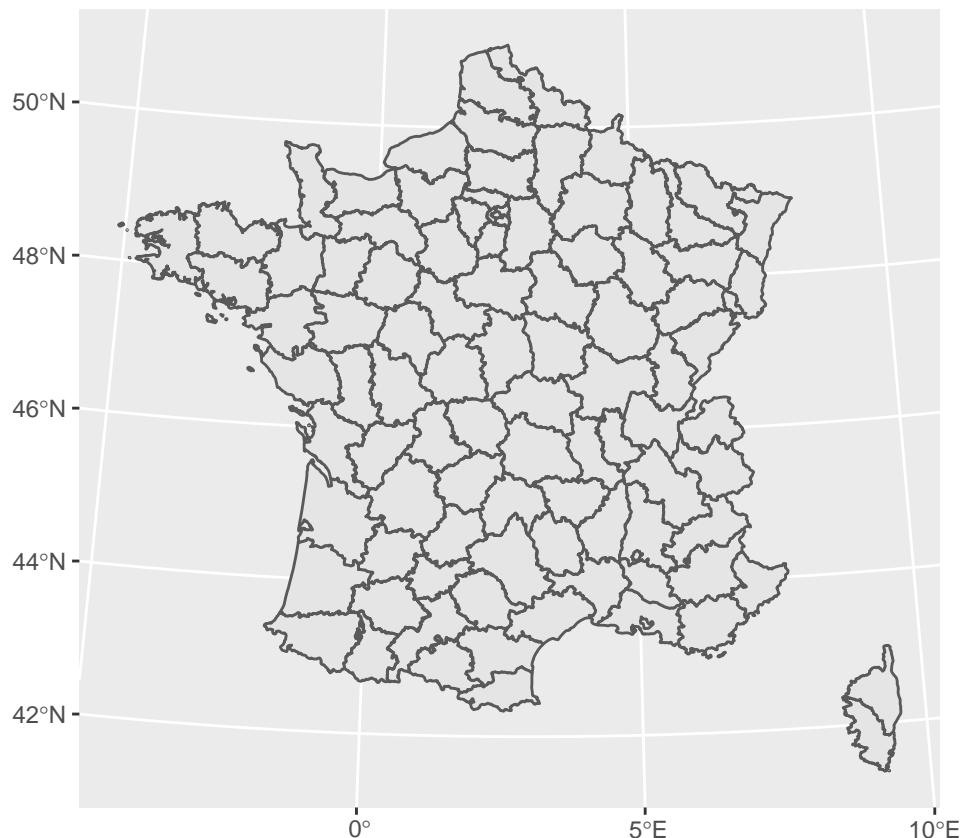
On peut ainsi représenter les centres des villes délimitées par les polygones des données **nc** avec

```
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 département du répertoire **dpt**, peut être importée dans un objet R :

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



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

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.
2. Faire la jointure de cette table avec celle des frontières des départements. On pourra utiliser `inner_join`.
3. Comparer les taux de chômage en 2006 et 2011 (on le fera avec une carte pour les taux en 2006 et une autre pour les taux en 2011).

2.2.1 Challenge 1 : carte des températures avec sf

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 les 15 derniers jours dans le lien téléchargement. On utilisera uniquement les identifiants de la station ainsi que la température observée (colonne `t`).
- la géolocalisation de ces stations dans le lien [documentation](#)

1. Importer les 2 bases nécessaires. On pourra les lire directement sur le site. Convertir les degrés Kelvin en degrés Celsius et faire la jointure de ces bases.
2. Éliminer 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.
3. Créer un datafram 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")
```

4. Représenter les stations sur une carte de france. On pourra mettre un point de couleur différente en fonction de la température.
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 (la température du département i sera celle de la station la plus proche du centroïde de i).

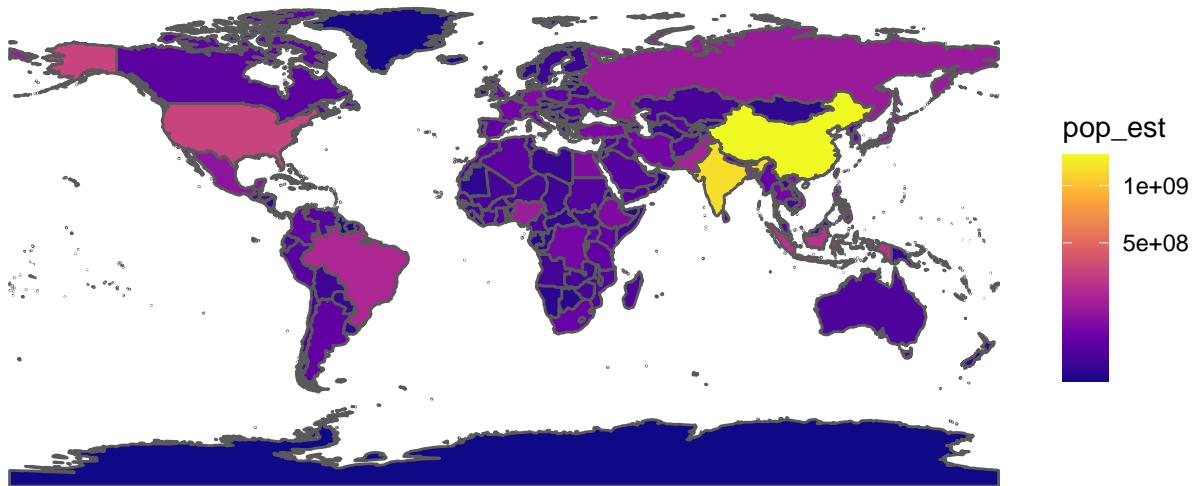
6. Colorier les départements en fonction de la température prédite dans le département. On pourra faire varier le dégradé de couleur du jaune (pour les faibles températures) au rouge (pour les fortes).

2.2.2 Trouver des cartes au format shapefile

Le plus souvent on ne va pas construire les fonds de carte au format shapefile “à la main” et il est bien entendu important de récupérer ces fonds de carte au préalable. La méthode la plus courante consiste à taper les bons mots clefs sur un moteur de recherche... On pourra par exemple utiliser :

— des packages R, par exemple `rnatuarlearth` :

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



On peut aussi visualiser la terre comme une sphère :

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



Voir <https://www.r-spatial.org/r/2018/10/25/ggplot2-sf.html> pour plus de détails.

— le **web**, par exemple le site [data.gouv](#) :

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

Attention, la taille des objets peut être très (trop) grande

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

et la construction de la carte peut dans ce cas prendre beaucoup de temps... On peut réduire la taille avec ce type d'outils

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



2.3 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 partir de couches qui se superposent. Un fond de carte s'obtient avec les fonctions `leaflet` et `addTiles`

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

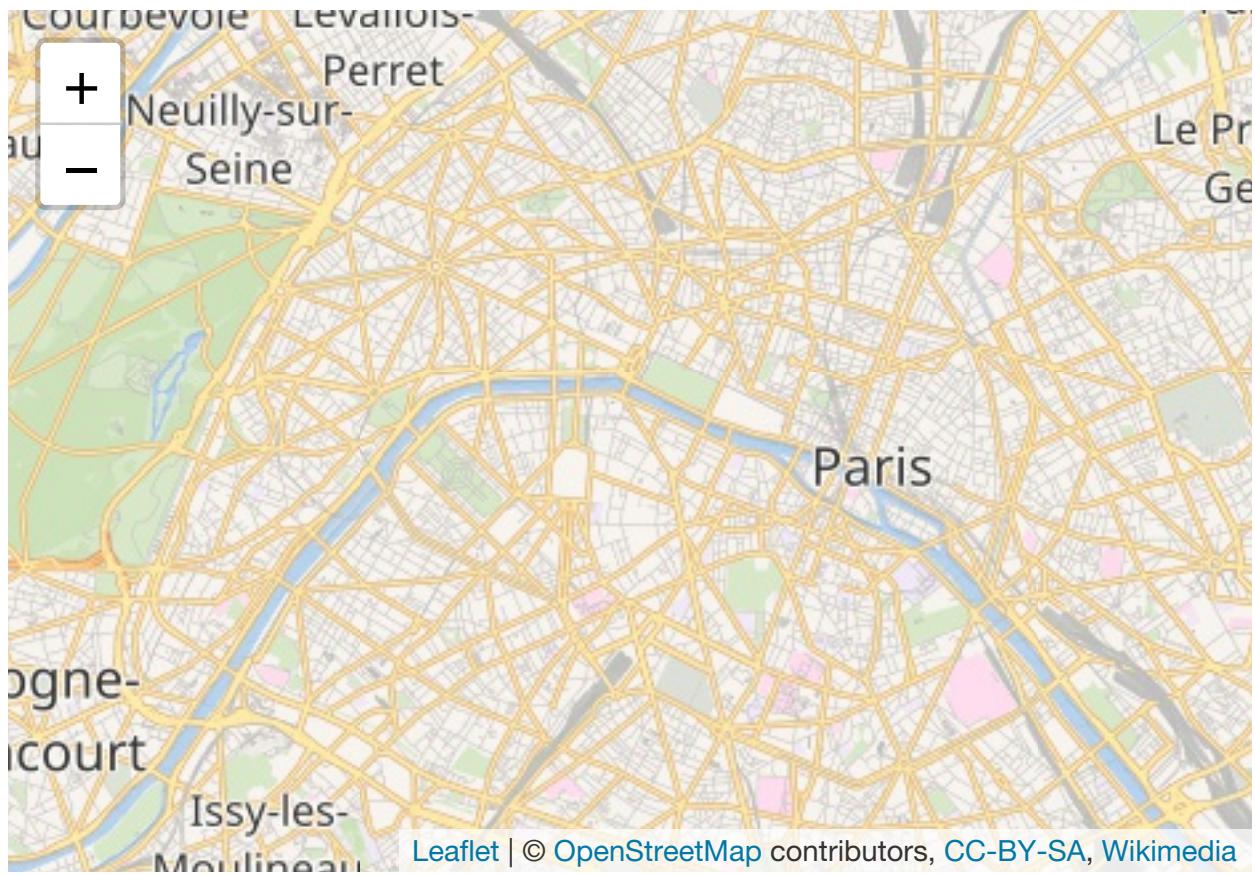


On dispose de plusieurs styles de fonds de cartes (quelques exemples [ici](#)) :

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



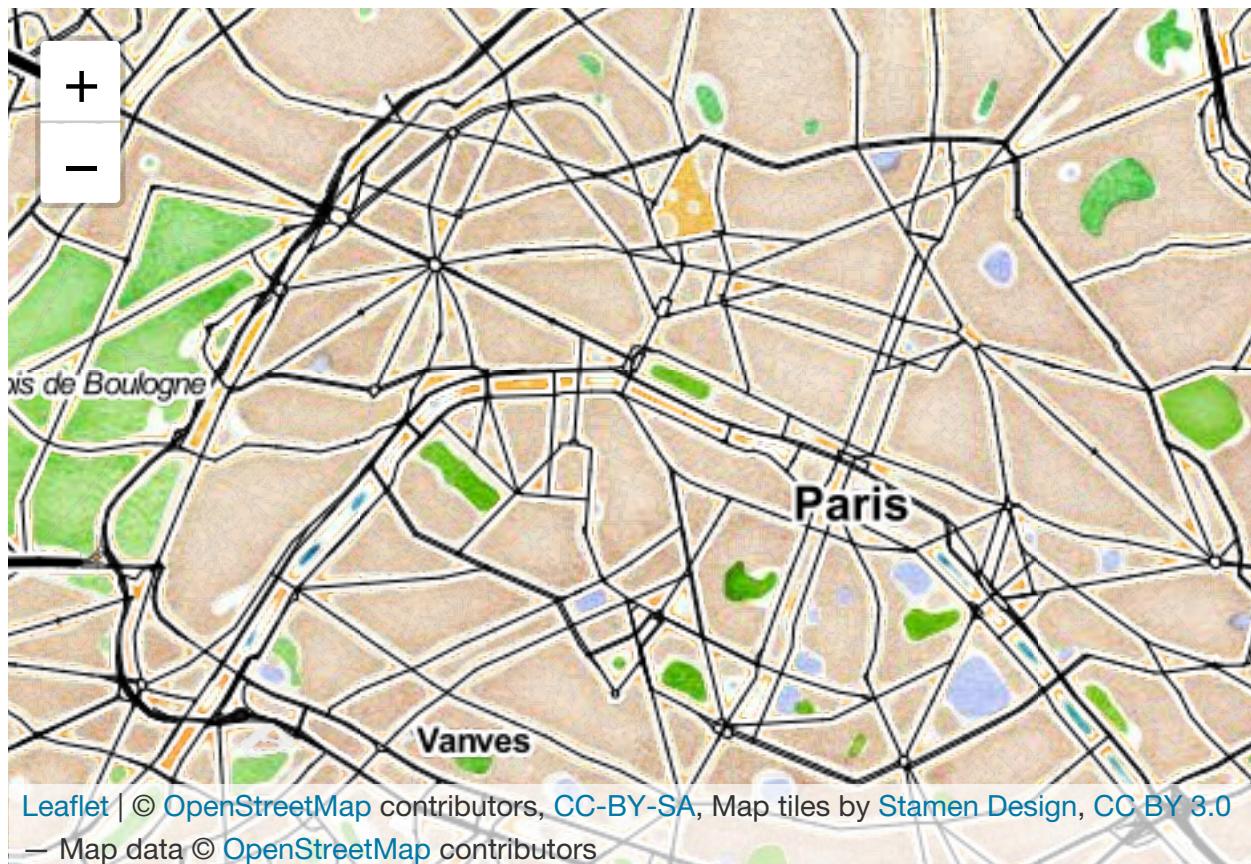
```
m2 %>% addProviderTiles("Wikimedia")
```



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

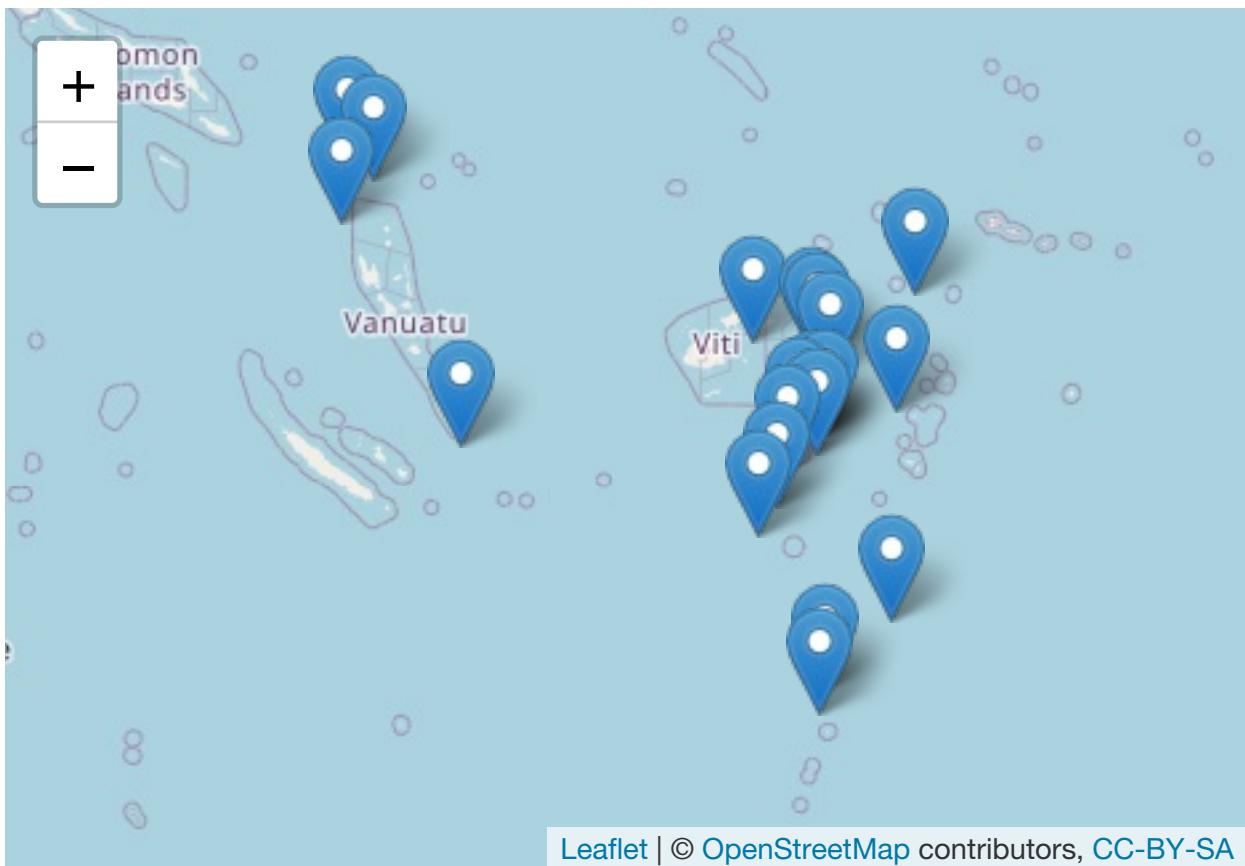


```
m2 %>%  
addProviderTiles("Stamen.Watercolor") %>%  
addProviderTiles("Stamen.TonerHybrid")
```



Il est souvent 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))
```

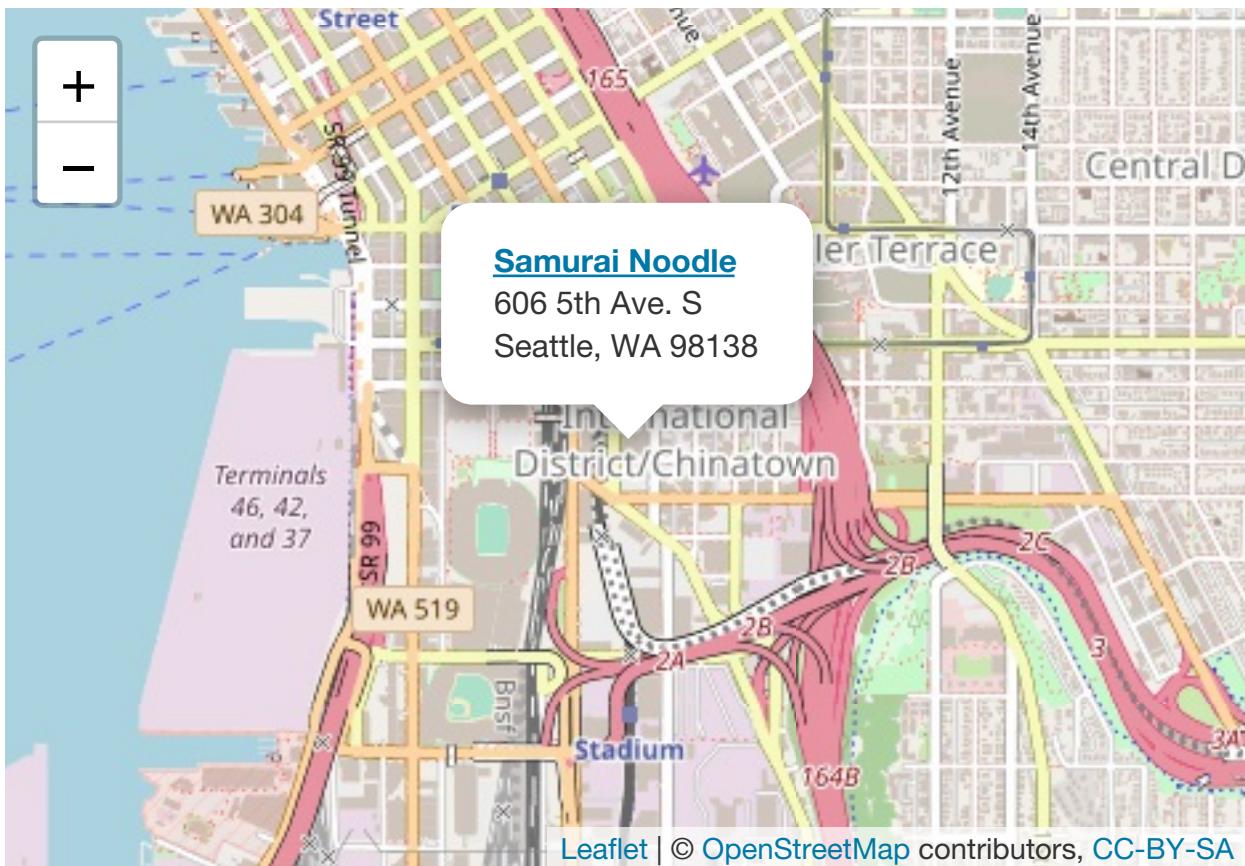


On remarque que l'on utilise ici un **tilde** pour spécifier qu'on utilise des variables dans un **dataframe**.

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.

2.3.1 Challenge 2 : Visualisation des stations velib à Paris

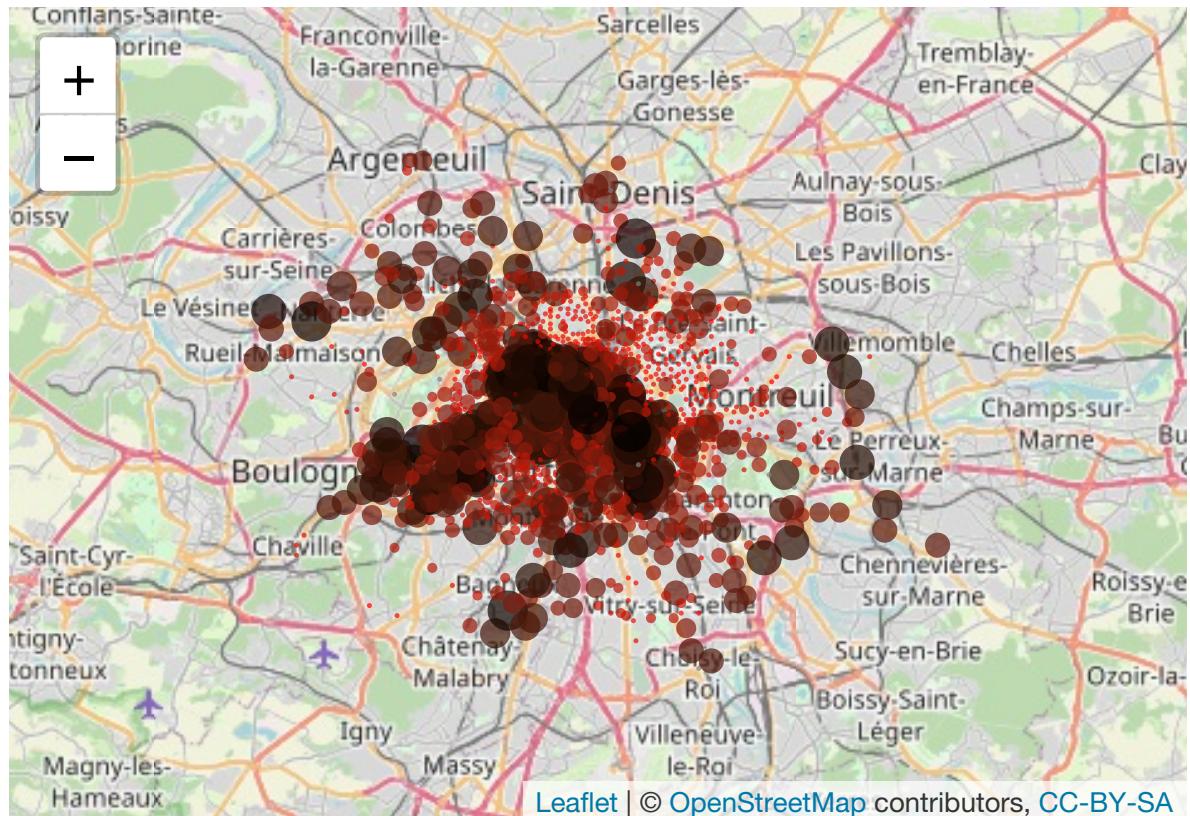
Plusieurs villes dans le monde ont accepté de mettre en ligne les données sur l’occupation des stations velib. Ces données sont facilement accessibles et mises à jour en temps réel. On dispose généralement de la taille et la localisation des stations, la proportion de vélos disponibles... Il est possible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

- sur les données [Decaux](#)
 - sur [Open data Paris](#)
 - sur [vlstats](#) pour des données mensuelles ou historiques ou encore sur Velib pour obtenir des fichiers qui sont rafraîchis régulièrement.
1. Récupérer les données actuelles de velib disponibles pour la ville de Paris : <https://opendata.paris.fr/explore/dataset/velib-disponibilite-en-temps-reel/information/>. On pourra utiliser la fonction `read_delim` avec l’option `delim=";"`.
 2. Décrire les variables du jeu de données.
 3. Créer une variable `latitude` et une variable `longitude` à partir de la variable `Coordonnées géographiques`. On pourra utiliser la fonction `separate` du package `tidyverse`.
 4. Visualiser les positions des stations sur une carte leaflet.
 5. Ajouter un popup qui permet de connaître le nombre de vélos disponibles (électriques+mécanique) quand on clique sur la station (on pourra utiliser l’option `popup` dans la fonction `addCircleMarkers`).
 6. Ajouter le nom de la station dans le popup.
 7. Faire de même en utilisant des couleurs différentes en fonction de la proportion de vélos disponibles dans la station. On pourra utiliser les palettes de couleur

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

```
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="")))

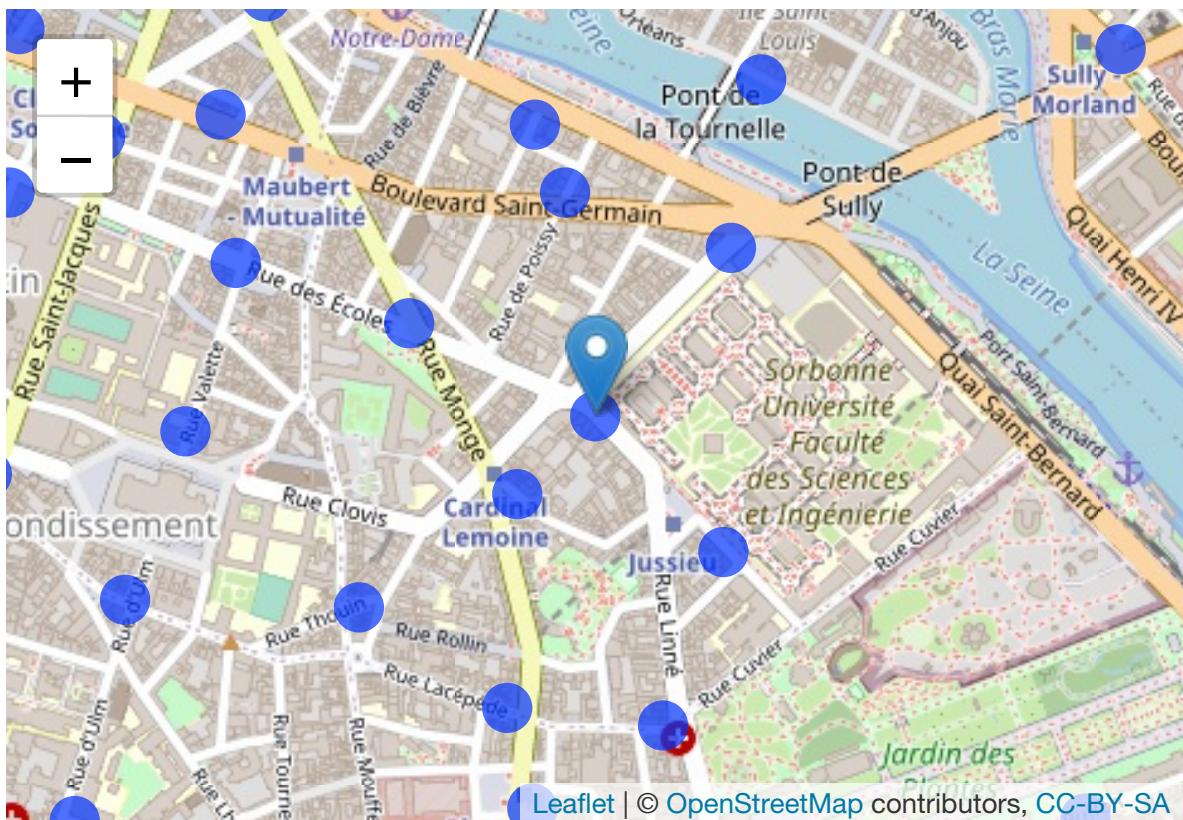
map.velib5
```



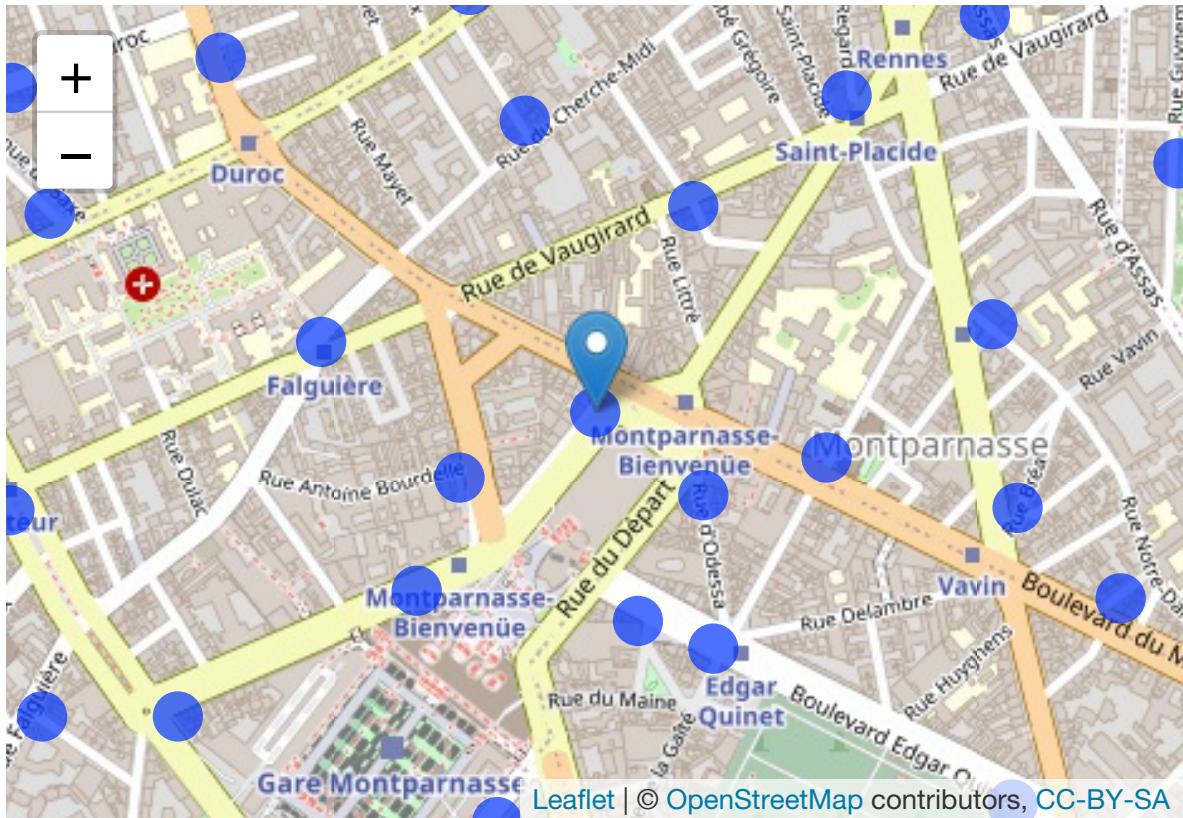
- Créer une fonction `local.station` qui permette de visualiser quelques stations autour d'une station choisie.

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.

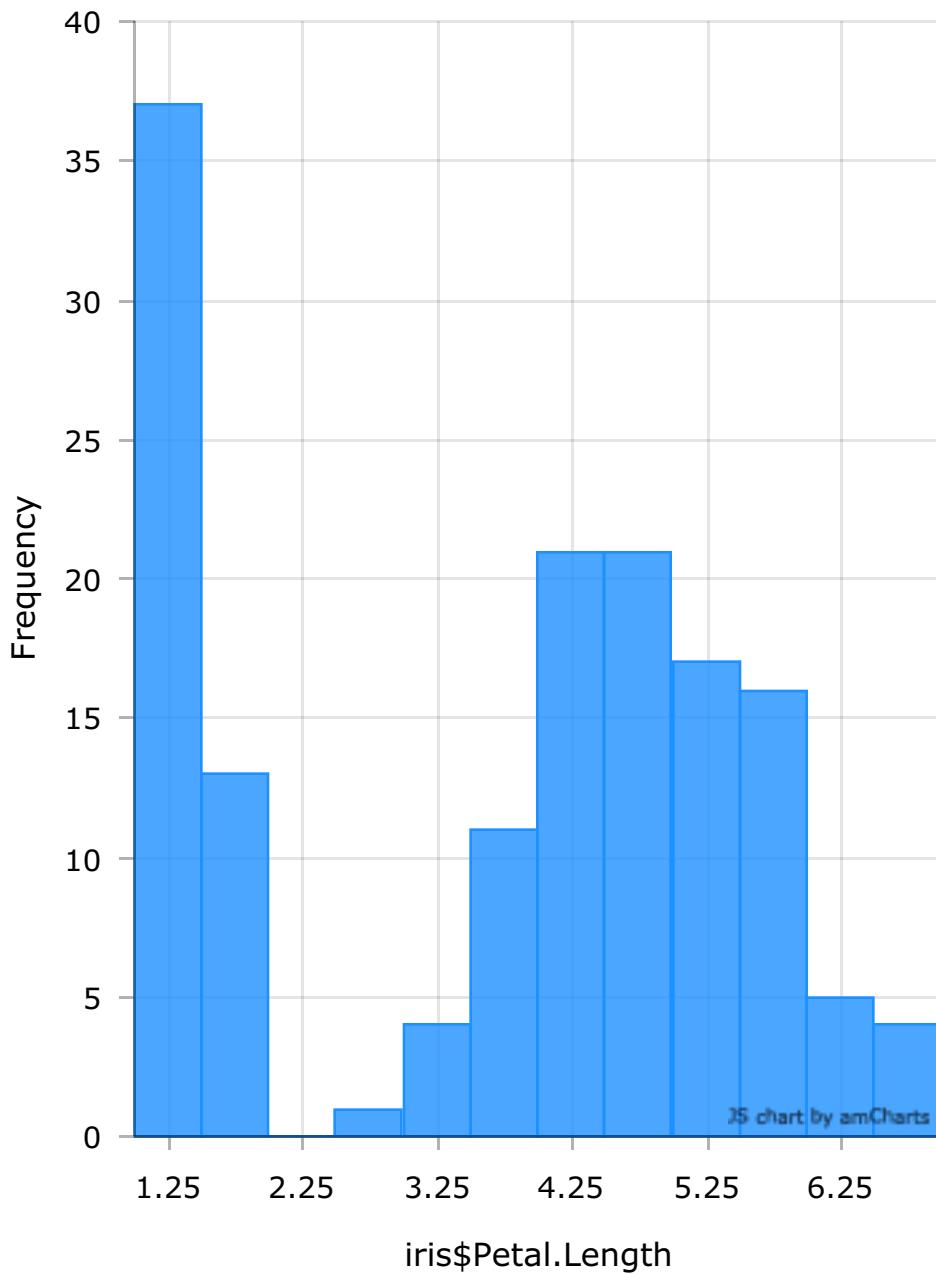
3 Quelques outils de visualisation dynamique/interactive

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.

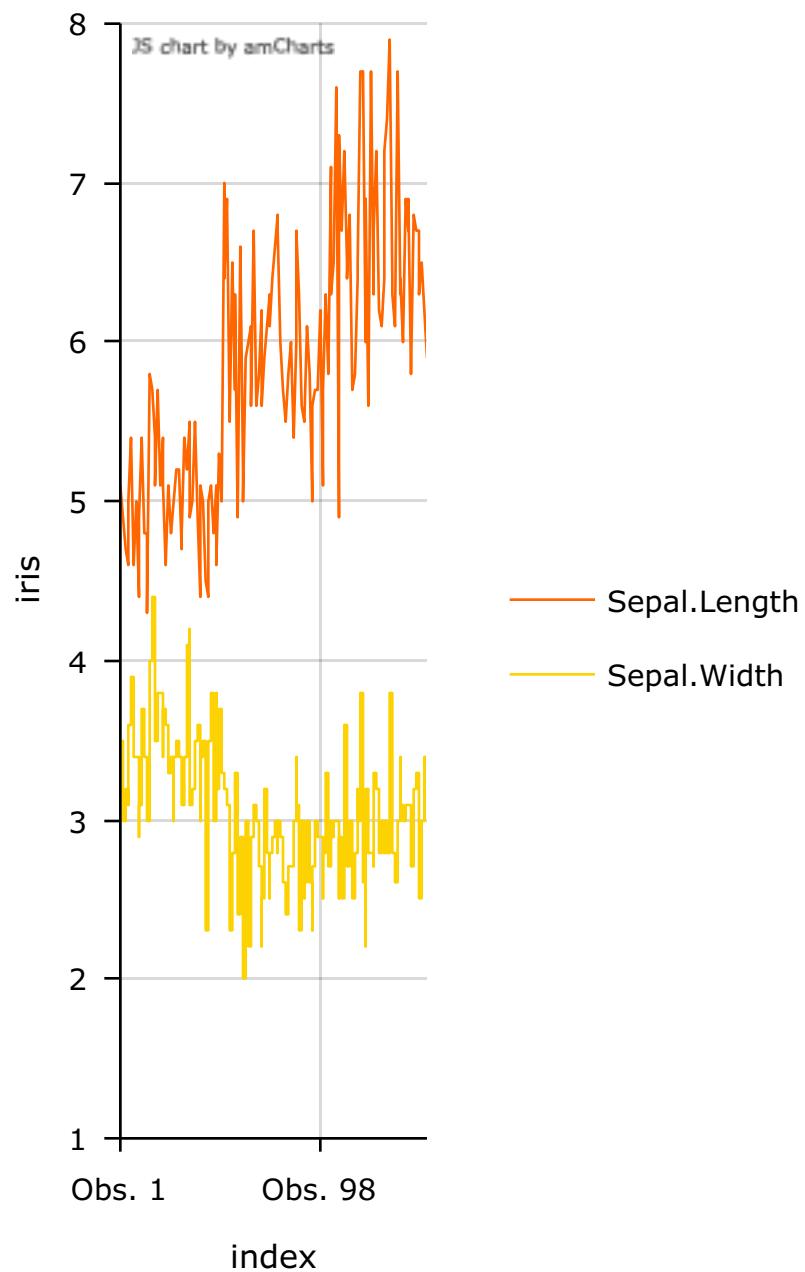
3.1 Représentations classiques avec rAmCharts et plotly

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 :

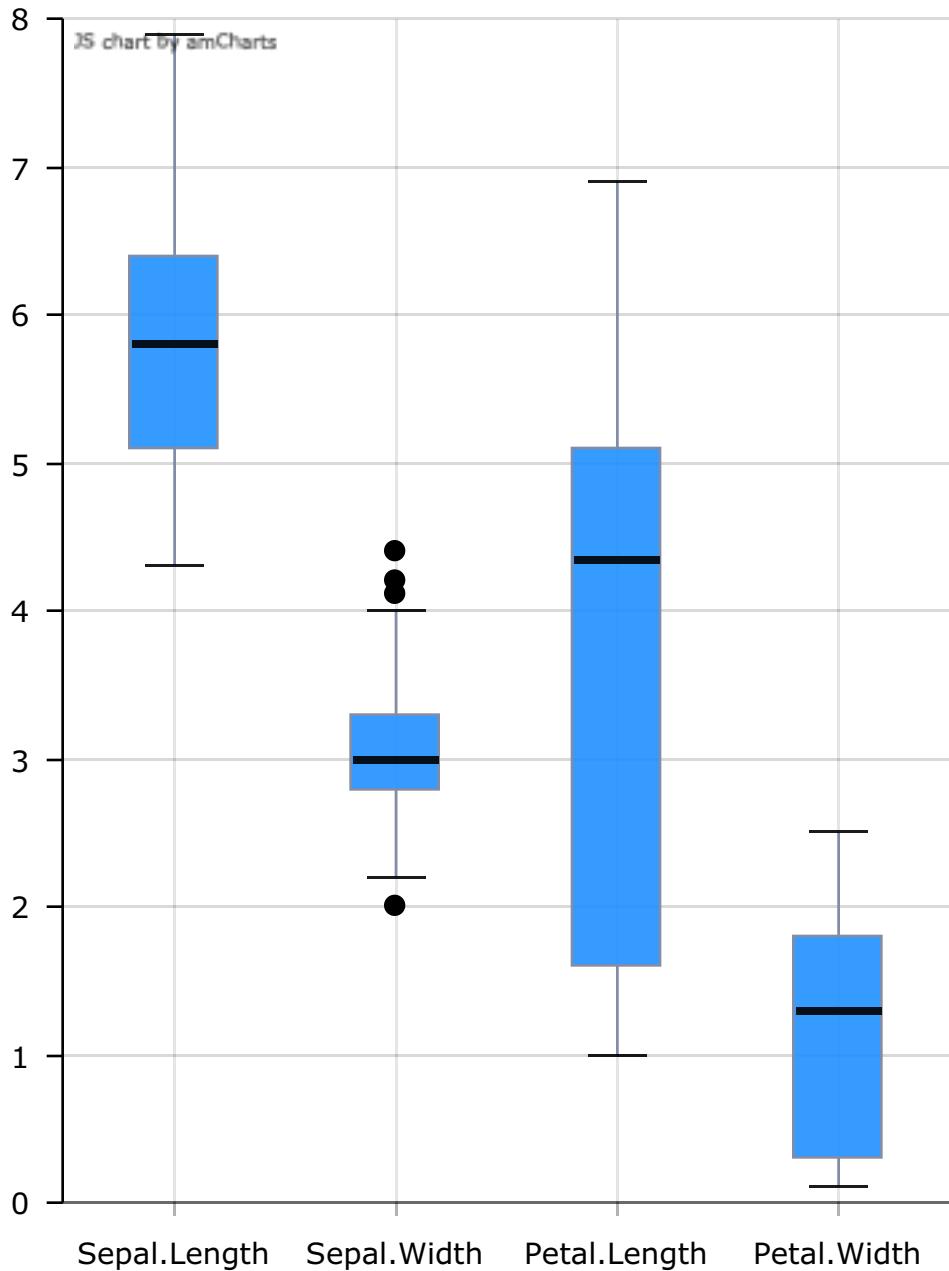
```
library(rAmCharts)
amHist(iris$Petal.Length)
```



```
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 de tracer un nuage de points en dimension 2 et d'y ajouter la droite de régression. On commence par générer le nuage et 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)

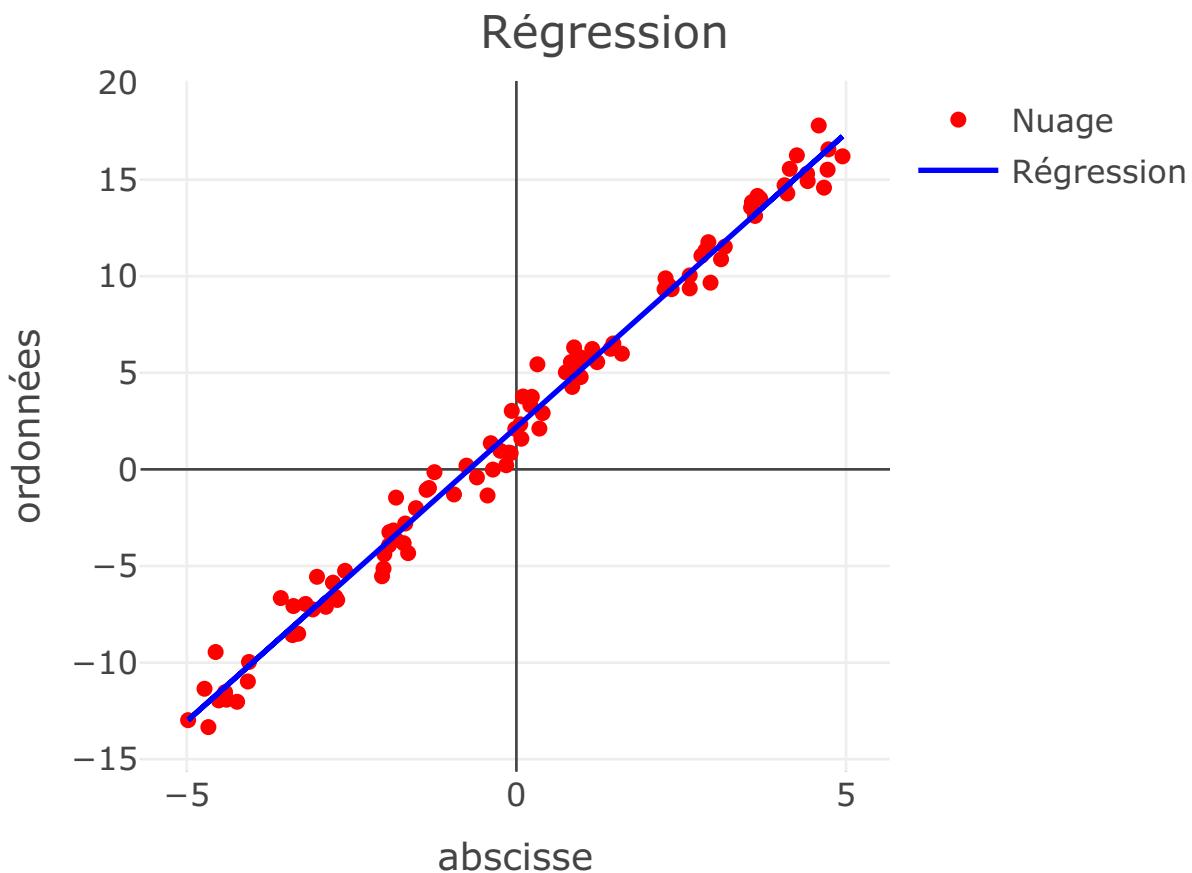
```

On effectue maintenant le tracé

```

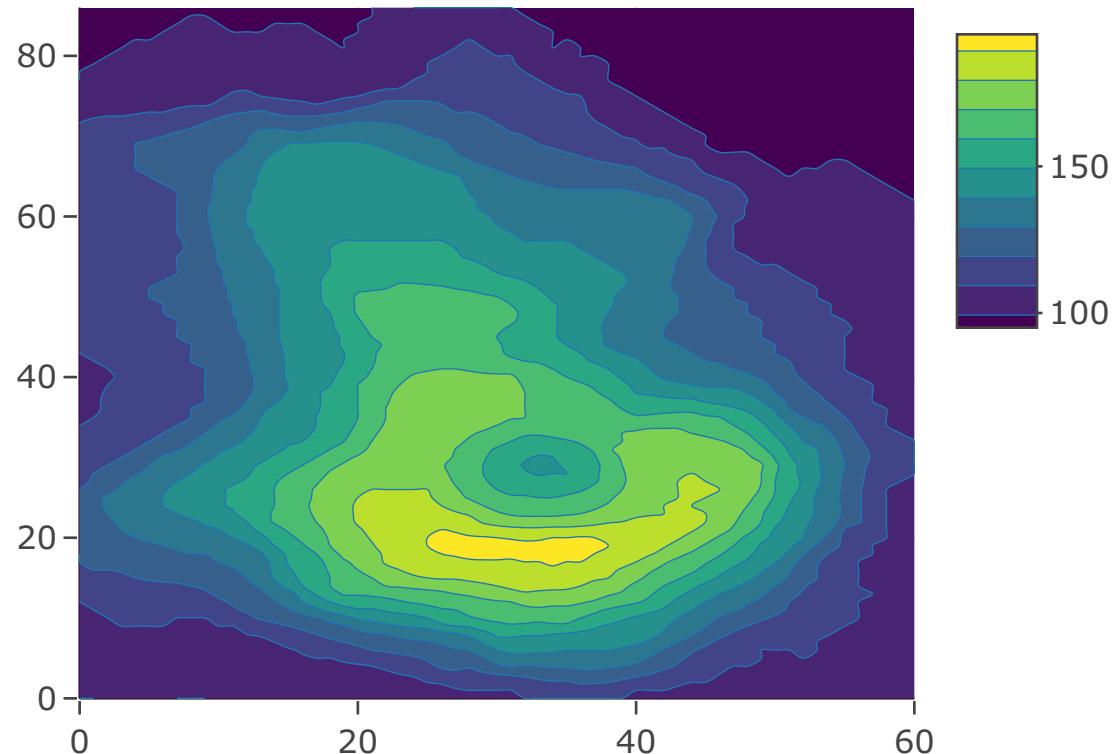
D %>% plot_ly(x=~X,y=~Y) %>%
  add_markers(type="scatter",mode="markers",
              marker=list(color="red"),name="Nuage") %>%
  add_trace(y=fitted(model),type="scatter",mode='lines',
             name="Régression",line=list(color="blue")) %>%
  layout(title="Régression",xaxis=list(title="abscisse"),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

```

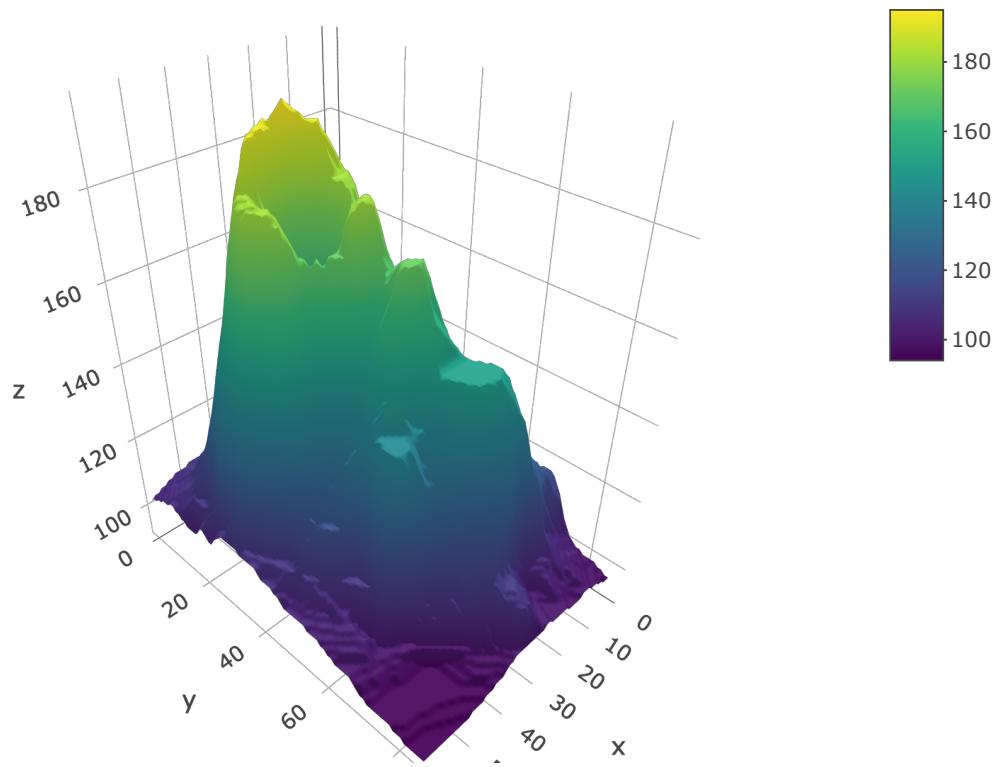


Contrairement à **ggplot**, **plotly** permet de faire de la 3D. Par exemple

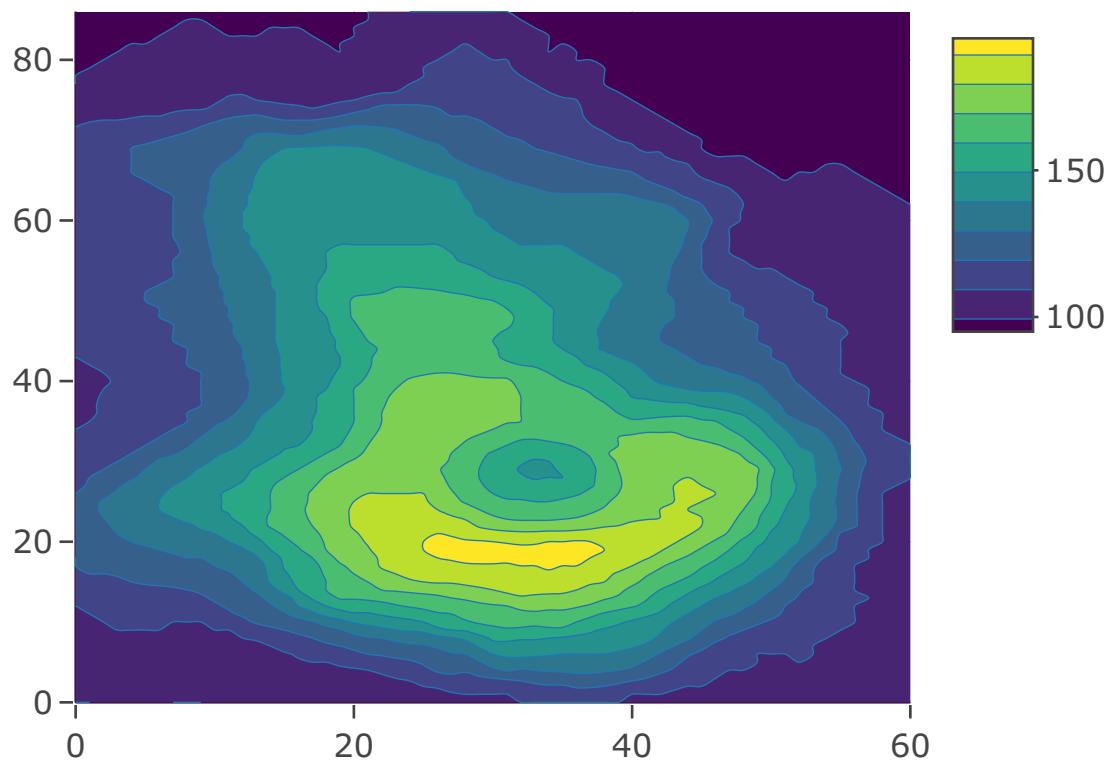
```
plot_ly(z = volcano, type = "contour")
```



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



```
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)
```



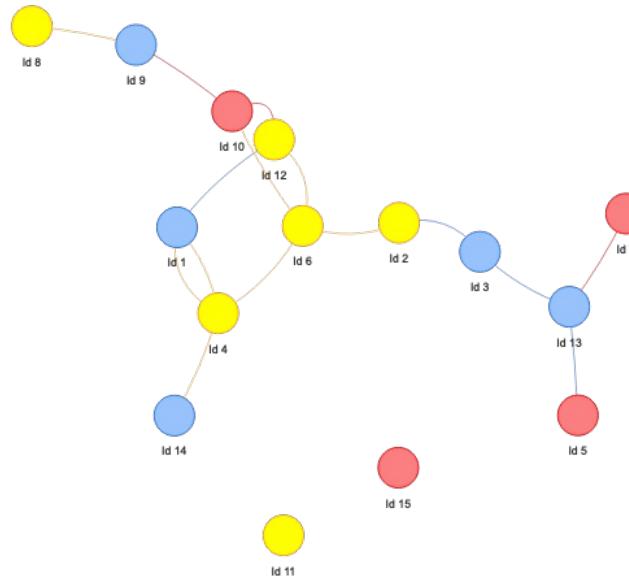
Exercice 3.1 (Graphes basiques avec ‘rAmCharts’ et ‘plotly’). Pour le jeu de données **iris** on effectuera les graphes suivants en **rAmCharts** et **plotly**.

1. Nuage de points représentant les longueurs et largeurs de Sépales. On utilisera une couleur différente en fonction de l’espèce.
2. Boxplot permettant de visualiser la distribution de la variable **Petal.Length** en fonction de l’espèce.

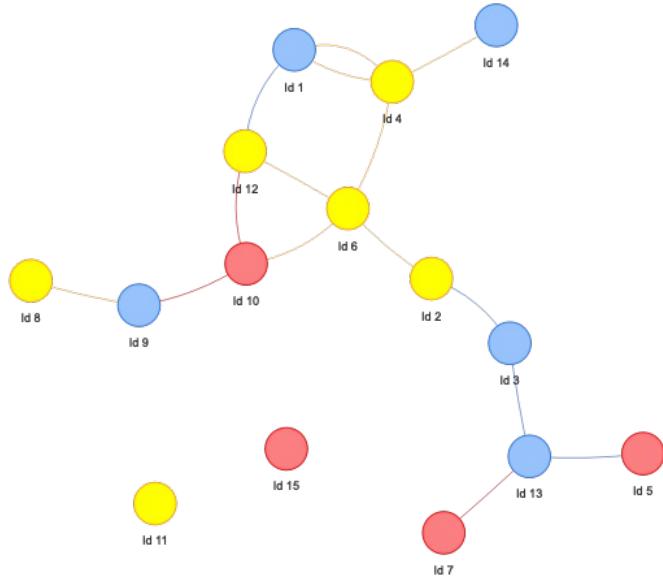
3.2 Graphes pour visualiser des réseaux avec visNetwork

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 représentés 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 nœuds et connexions 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)
```

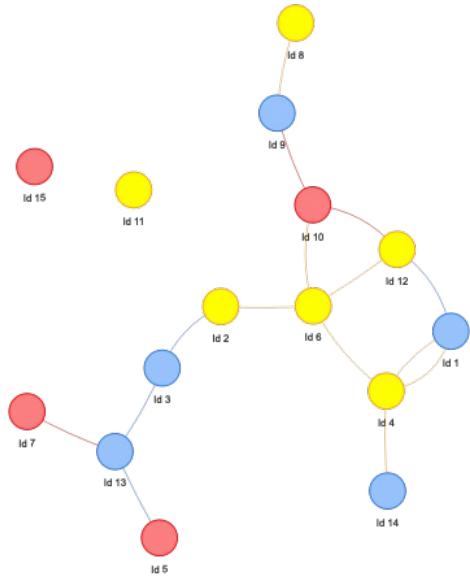


```
visNetwork(nodes, edges) %>% visOptions(highlightNearest = TRUE)
```



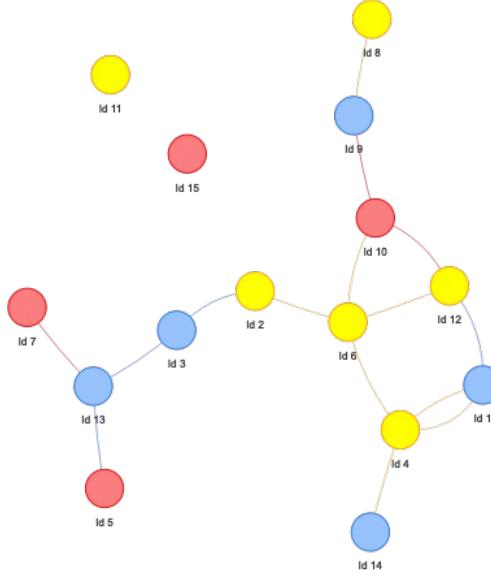
```
visNetwork(nodes, edges) %>% visOptions(highlightNearest = TRUE,
nodesIdSelection = TRUE)
```

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("data/Dataset1-Media-Example-NODES.csv", header=T, as.is=T)
links <- read.csv("data/Dataset1-Media-Example-EDGES.csv", header=T, as.is=T)
head(nodes)
  id          media media.type type.label
1 s01        NY Times      1  Newspaper
2 s02  Washington Post    1  Newspaper
3 s03 Wall Street Journal 1  Newspaper
4 s04        USA Today    1  Newspaper
5 s05        LA Times    1  Newspaper
6 s06  New York Post     1  Newspaper
audience.size
1      20
2      25
3      30
4      32
5      20
6      50
head(links)
  from   to weight      type
1  s01  s02     10 hyperlink
2  s01  s02     12 hyperlink
3  s01  s03     22 hyperlink
4  s01  s04     21 hyperlink
5  s04  s11     22   mention
6  s05  s15     21   mention
```

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

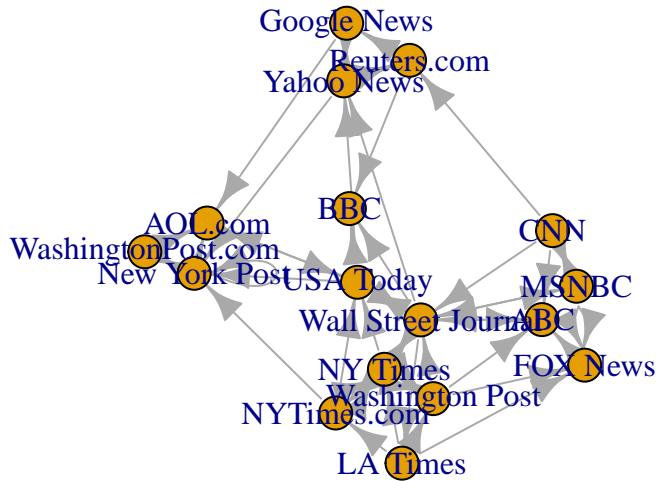
```

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

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

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

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 **max03**) 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`.
- b. Représenter l'histogramme de la variable `maxO3`, on fera le graphe `ggplot` et `rAmCharts` et `plotly` (en utilisant `ggplotly` par exemple).
- 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 lisseur sur le graphe.

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

- a. 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.
- b. Ajouter un nouvel onglet qui permet de visualiser le `summary` du modèle linéaire complet. On pourra utiliser la fonction `datatable` du package `DT`. **Indication** : ce nouvel onglet peut se créer avec
- c. Ajouter un nouvel onglet qui permet de visualiser un modèle linéaire simple avec la variable explicative de votre choix. On pourra afficher dans cet onglet le `summary` du modèle ainsi que le nuage de points et la droite de régression.
- d. **Pour aller plus loin** : ajouter un dernier onglet qui permette à l'utilisateur de choisir la variable explicative du modèle simple. **Indications** : on pourra utiliser les commandes `Shiny`
 - Choix de la variable

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

- Mise à jour du résumé

```
mod1 <- reactive({
  XX <- paste(input$variable1,collapse="+")
  form <- paste("maxO3~",XX,sep="") %>% formula()
  lm(form,data=df)
})
#Df correspond au jeu de données
renderDataTable({
  mod.sum1 <- summary(mod1())$coefficients %>% round(3) %>% as.data.frame()
  DT::datatable(mod.sum1,options = list(dom = 't'))
})
```

- Mise à jour du graph interactif

```
renderPlotly({
  (ggplot(df)+aes(x=!!as.name(input$variable1),y=maxO3)+
    geom_point()+geom_smooth(method="lm")) %>% ggplotly()
})
```

Enfin il ne faudra pas oublier d'ajouter

```
runtime: shiny
```

dans l'entête.

- e. Ajouter un dernier 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/>

4 Applications web avec Shiny

Shiny est un package R qui permet la création simple d'applications web interactives depuis R. Cette partie provient essentiellement du tutoriel de Benoît Thieurmel disponible ici : https://github.com/datastorm-open/tuto_shiny_rennes.

4.1 Une première application

Créer un répertoire pour l'application avec **RStudio**

```
File -> New Project -> New Directory -> Shiny Web Application
```

Choisir une application **Multiple File**.

Si cette option n'est pas disponible (ça peut dépendre des versions de Rstudio), on pourra utiliser

```
File -> New File -> Shiny Web App -> Multiple File
```

Deux fichier sont automatiquement générés : **ui.R** et **server.R**. Lancer l'application en cliquant sur le bouton **Run App**.

- Changer le titre de l'application. On pourra l'appeler **My first application**.
- Mettre à jour et vérifier que le titre a bien été changé.

4.2 Input - output

On garde la même application. On ne s'intéressera pas à la structure dans cette partie, on veut simplement ajouter

- des nouveaux **inputs** dans le **sidebarPanel**, après le **sliderInput**. On n'oubliera pas de séparer les inputs par des virgules ;
- des nouveaux **outputs** dans le **mainPanel**, après le **plotOutput**. Là encore, on n'oubliera pas de séparer les outputs par des virgules.

Pour résumer on souhaite une colonne avec tous les inputs et une autre avec tous les outputs.

1. Ajouter dans **ui.R** une entrée qui permette de changer la couleur de l'histogramme. On pourra utiliser

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

2. Ajouter une sortie qui permette de visualiser le **summary** du jeu de données **faithful**. On pourra utiliser

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

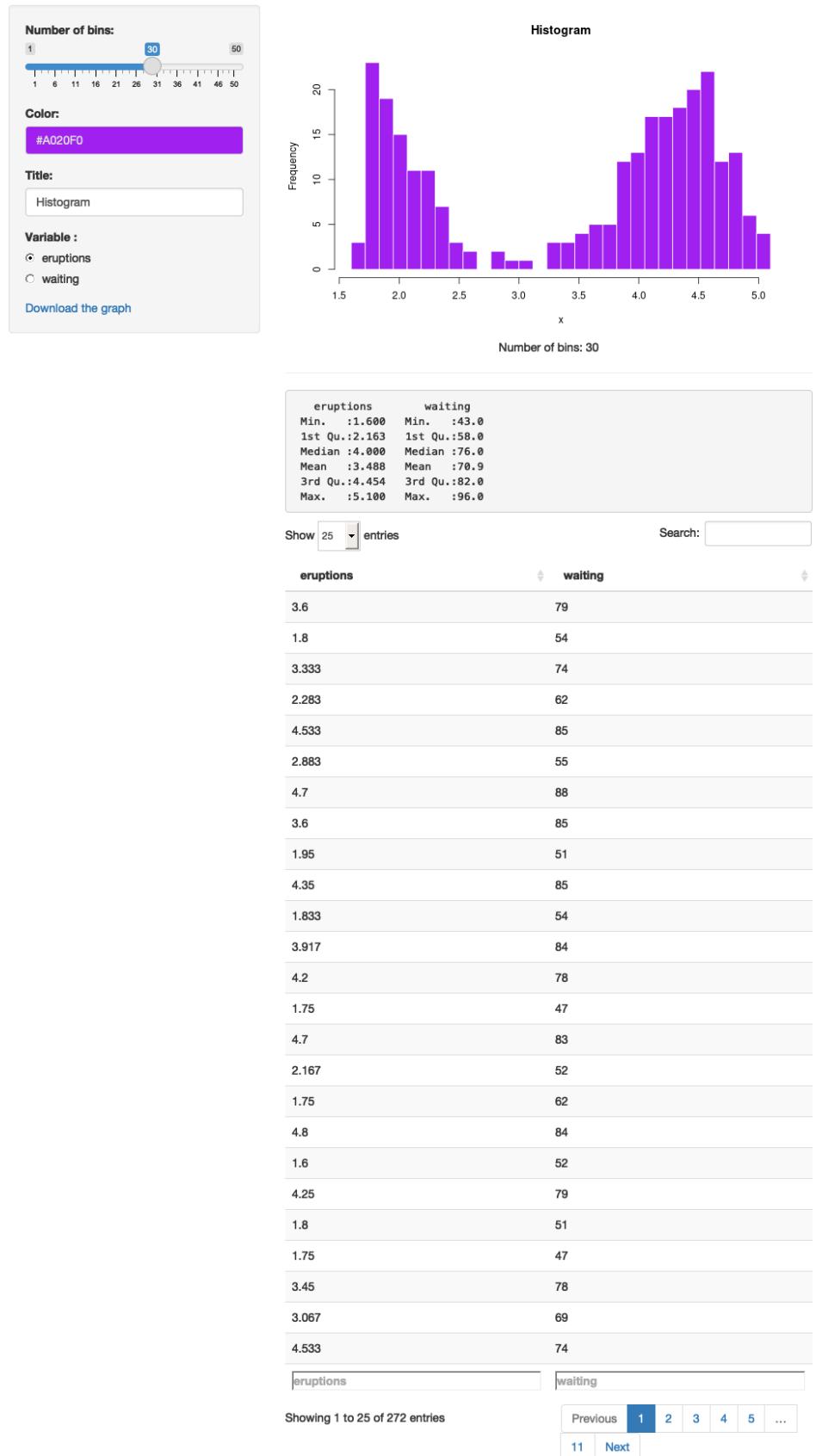
# server.R
output$... <- renderPrint({
  summary(...)
})
```

Exercice 4.1 (Ajouter des inputs/outputs). Ajouter des entrées/sorties à votre application pour

1. proposer à l'utilisateur de choisir un titre pour l'histogramme (utiliser `textInput` dans l'**ui** et l'option `main` dans **hist**) ;
2. choisir la variable de `faithful` à représenter dans l'histogramme avec un `radioButtons` ayant pour choix `colnames(faithful)` ;
3. visualiser le jeu de données entier (`renderDataTable & dataTableOutput`) ;
4. ajouter un `text` sous l'histogramme qui indique le nombre de classes (`renderText` et `paste` dans **server**, `textOutput` dans **ui**) ;
5. remplacer le `selectInput` du choix de la couleur par un `colourInput` (utiliser la package `colourpicker`) ;
6. exporter le graphe (`downloadButton & jpeg`).

L'application demandée pourra ressembler à

My first application



Elle est également disponible ici <https://input-output-rouviere-shiny.apps.math.cnrs.fr/>.

4.3 Structurer l'application

On considère l'application **app_structure** disponible [ici](#). C'est quasiment la même que précédemment avec un **navbarPage** qui définit

- un onglet *Data* pour visualiser les données (table + summary)
- un onglet *Visualisation* : inputs + histogramme.

Exercice 4.2 (Structurer son application). On conserve l'application précédente.

1. Dans l'onglet **Data** utiliser **navlistPanel** pour séparer le **summary** et la table **table** en deux onglets :

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

2. Dans l'onglet **Visualization** changer **sidebarLayout - sidebarPanel - mainPanel** par un **fluidRow** à 2 colonnes :

- 1/4 : pour le **sidebarPanel**
- 3/4 : pour le **mainPanel**.

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

*Indication : utiliser **wellPanel** pour la colonne de gauche.*

3. Ajouter un boxplot dans l'onglet visualisation (même variable et même couleur). On pourra également utiliser **tabsetPanel** pour avoir deux onglets pour l'histogramme et le boxplot.

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

L'application demandée pourra ressembler à

Second app Data Visualisation

Table

Summary

Show 25 entries

Search:

eruptions	waiting
3.6	79
1.8	54
3.333	74
2.283	62
4.533	85
2.883	55
4.7	88
3.6	85
1.95	51
4.35	85
1.833	54
3.917	84
4.2	78
1.75	47
4.7	83
2.167	52
1.75	62
4.8	84
1.6	52
4.25	79
1.8	51
1.75	47
3.45	78
3.067	69
4.533	74
eruptions	waiting

Showing 1 to 25 of 272 entries

Previous **1** 2 3 4 5 ... 11 Next

Elle est également disponible ici <https://structure-rouviere-shiny.apps.math.cnrs.fr/>.

Pour aller plus loin : faire la même application avec shinydashboard (<https://rstudio.github.io/shinydashboard/>).

4.4 Ajout de graphes interactifs

Dans l'application précédente, remplacer l'histogramme et la boxplot par des graphes javascript réalisés avec **rAmCharts**. On pourra utiliser

```
# server.R
output$distPlot <- renderAmCharts(...)

# ui.R
amChartsOutput("...")
```

L'application demandée pourra ressembler à

Premiers pas avec shiny Data Visualisation

Table Summary

Show 25 entries Search:

eruptions	waiting
3.6	79
1.8	54
3.333	74
2.283	62
4.533	85
2.883	55
4.7	88
3.6	85
1.95	51
4.35	85
1.833	54
3.917	84
4.2	78
1.75	47
4.7	83
2.167	52
1.75	62
4.8	84
1.6	52
4.25	79
1.8	51
1.75	47
3.45	78
3.067	69
4.533	74

eruptions waiting

Showing 1 to 25 of 272 entries Previous **1** 2 3 4 5 ... 11 Next

Elle est également disponible ici <https://interactifs-rouviere-shiny-2.apps.math.cnrs.fr/>.

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

Garder la même application et

- ajouter un `ActionButton` combiné à un `isolate` pour mettre à jour l'application uniquement lorsque l'utilisateur clique sur le bouton.

- Utiliser `observeEvent` pour forcer l'apparition de l'histogramme lorsqu'on met à jour l'application. On pourra utiliser

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

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

# and finaly
observeEvent(input$go, {
  updateTabsetPanel(session, inputId = "viz", selected = "Histogram")
})
```

- Utiliser `reactive` pour stocker la variable sélectionnée

```
# Example of reactive
data <- reactive({
  ...
})

output$plot <- renderPlot({
  x <- data()
  ...
})
```

- Ajouter un titre en bleu sur le jeu de données. On pourra utiliser `h1`

```
h1("Dataset", style = "color : #0099ff;text-align:center")
```

- Ajouter un troisième onglet pour présenter un résumé de votre Uinersité, avec un logo de l'institution et un lien vers son site web.
- **Pour aller plus loin** : changer le thème de l'application avec un fichier de style `.css`. On pourra par exemple utiliser bootswatch <http://bootswatch.com/3>.

L'application finale pourra ressembler à

Table

Summary

Show 25 entries

Search:

eruptions	waiting
3.6	79
1.8	54
3.333	74
2.283	62
4.533	85
2.883	55
4.7	88
3.6	85
1.95	51
4.35	85
1.833	54
3.917	84
4.2	78
1.75	47
4.7	83
2.167	52
1.75	62
4.8	84
1.6	52
4.25	79
1.8	51
1.75	47
3.45	78
3.067	69
4.533	74

Elle est également disponible ici <https://plus-loin-rouviere-shiny-2.apps.math.cnrs.fr/>.

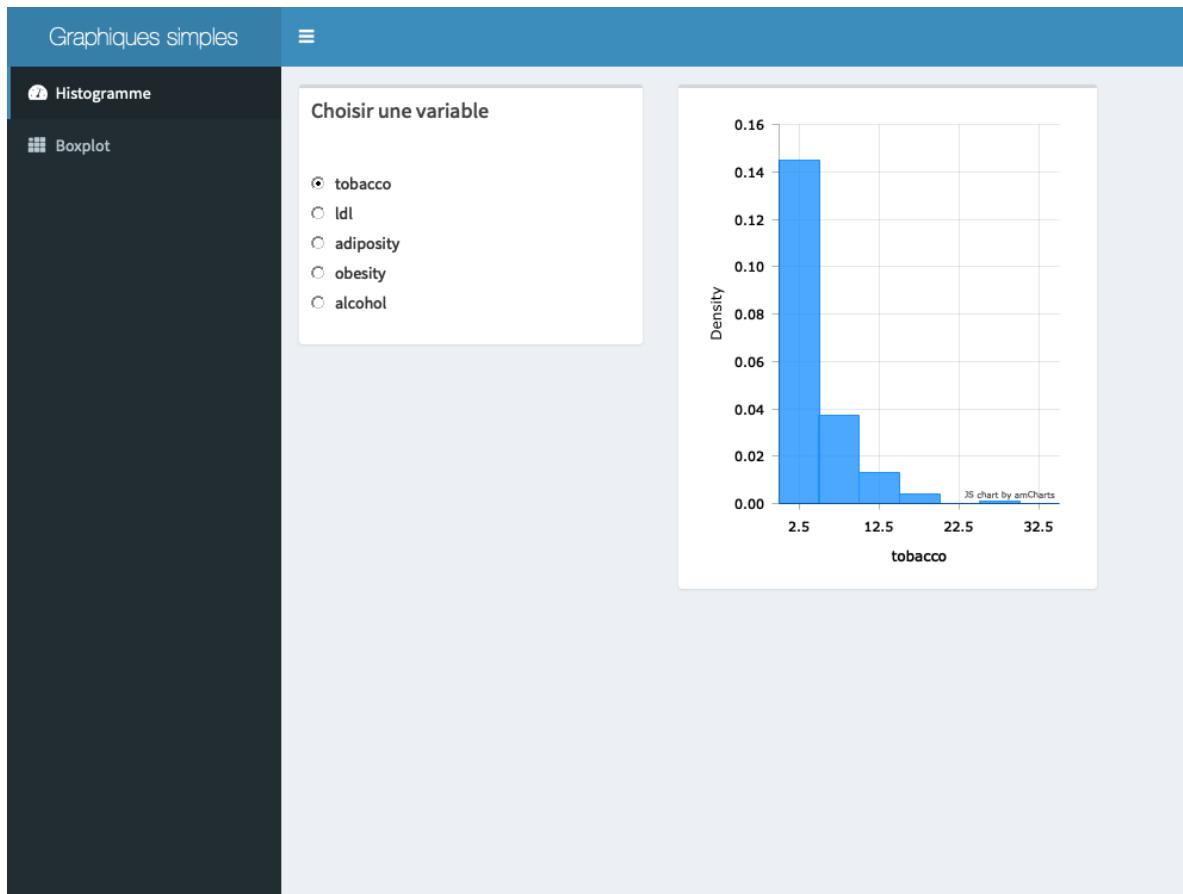
4.6 Exercices complémentaires

Exercice 4.3 (Une application simple descriptive). On considère le jeu de données *SAheart* du package *bestglm*.

1. A l'aide du package **rAmCharts**, représenter les histogrammes des variables quantitatives du jeu de données ainsi que les boxplots de ces variables en fonction de la variable **chd**.
2. Créer une application shiny avec **shinydashboard** qui permette de
 - choisir une variable parmi les variables quantitatives du jeu de données. On pourra utiliser **radioButtons** avec l'argument

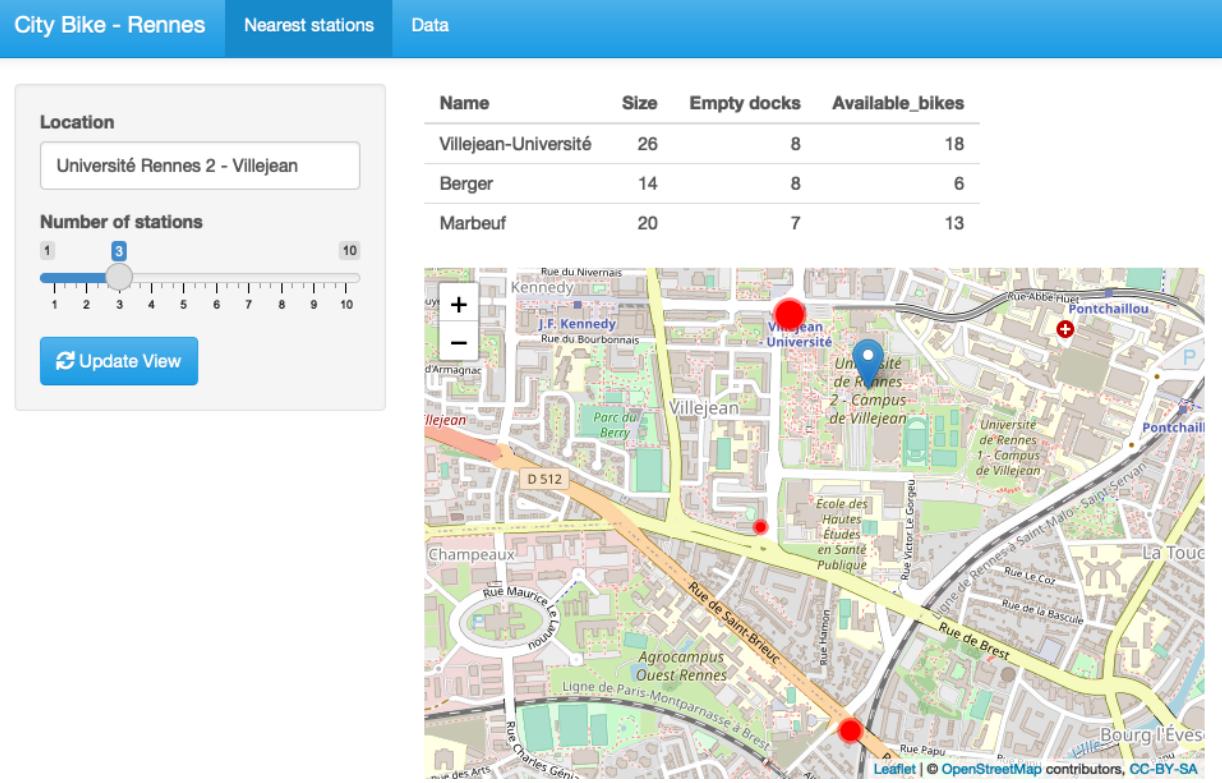
```
choices=names(SAheart)[sapply(SAheart, class)=="numeric"]
```

- visualiser l'histogramme, puis le boxplot en fonction de *chd* de la variable sélectionnée. Ces graphiques devront être faits avec **rAmCharts**. On pourra utiliser **amChartsOutput**. L'application demandée pourra ressembler à



Elle est disponible ici https://lrouviere.shinyapps.io/DESC_APP.

Exercice 4.4 (Stations velib à Rennes). Réaliser une application qui permette de visualiser les stations velib à Rennes. Elle pourra être du même genre que celle-ci :



On peut avoir une meilleure vision ici : <https://lrouviere.shinyapps.io/velib/>. On récupérera les données sur le site de Rennes métropole : <https://data.rennesmetropole.fr/explore/dataset/etat-des-stations-le-velo-star-en-temps-reel/export/>