# Chapitre 4 - Les graphiques

Thibault LAURENT

Mise à jour: 14 novembre 2022

## Contents



Ce document a été généré directement depuis **RStudio** en utilisant l'outil Markdown. La version .pdf se trouve ici.

**Résumé**

Fidèle à son principe, le logiciel **R** a recours à l'utilisation de fonctions pour la réalisation de graphiques. Le concept est dans un premier temps d'ouvrir une fenêtre graphique dans laquelle on va représenter un ou plusieurs graphiques. On pourra paramétrer d'une part la fenêtre graphique (marges, couleurs de fond, etc.) et les graphiques eux-mêmes (taille et couleur des traits, type de symbole, couleurs, etc.). Certains logiciels utilisent le principe de couches superposables qu'on peut ajouter ou enlever comme on veut. Ce n'est malheureusement pas le cas de **R** : lorsqu'on ajoute un trait, un point ou une légende dans un graphique, il est indélébile; en d'autres termes on ne peut pas revenir dessus sauf en reprenant le graphique depuis le début. Toutefois, malgré ces quelques inconvénients, on verra que grâce à un nombre important de fonctions prédéfinies, on est à peu près capable de représenter tout ce que l'on peut imaginer de faire avec **R**. On verra dans la dernière partie comment faire des graphiques en utilisant la syntaxe de type **ggplot2**.

**Prérequis**

Avant de commencer, vous devez effectuer les opérations suivantes afin de disposer de tous les éléments nécessaires à l'apprentissage de ce chapitre.

1. Créer un dossier propre à cet E-thème et l'indiquer comme répertoire dans lequel vous allez travailler à l'aide de la fonction `setwd()`.

```
setwd("/home/laurent/Documents/M2/Cours R initiation/chapitre 4")
```

2. Charger le code **R** qui permet de créer le jeu de données présenté dans le chapitre 1 :

```
load(file("http://www.thibault.laurent.free.fr/cours/Ressource/diamants.RData"))
```

Afin d'avoir des représentations graphiques moins lourdes, on va se restreindre à un sous-échantillon de taille 5000. En effet, pour certains graphiques, lorsque le nombre d'observations est important, sa lecture devient difficile.

```
set.seed(123) # on fixe une graine aléatoire pour obtenir à chaque fois le même tirage
diam_ech <- diamants[sample(nrow(diamants), 5000, replace = F), ]
```

3. Installer les packages suivants que nous utiliserons dans ce chapitre :

```
install.packages(c("caschrono", "ggplot2", "gridExtra"))
```

# 1 Les fonctions graphiques

## 1.1 Principe

Il existe trois grandes familles de fonctions dédiées aux graphiques :

- fonctions de haut-niveau : elles permettent d'ouvrir une fenêtre graphique et de réaliser un graphique de type : nuage de points, diagramme en barres, histogramme, etc. En principe, une fonction de haut-niveau appelée toute seule est suffisament bien programmée et paramétrée pour permettre à l'utilisateur d'obtenir une figure "statisfaisante", dans la mesure où cette figure contient tous les éléments nécessaires à sa compréhension (un titre, une légende, des axes gradués, etc.). Les fonctions de haut-niveau que nous allons voir sont : `plot()`, `hist()`, `barplot()`, etc.
- fonctions de bas-niveau : elles permettent d'ajouter des points, des lignes, des polygones, une légende, des étiquettes, etc. à un graphique déjà existant. Les fonctions de bas-niveau que nous allons voir sont : `points()`, `lines()`, `title()`, `axis()`, etc.
- paramétrages : ces fonctions permettent de définir/modifier le "style" des graphiques. Ceci se fera essentiellement avec la fonction `par()`.

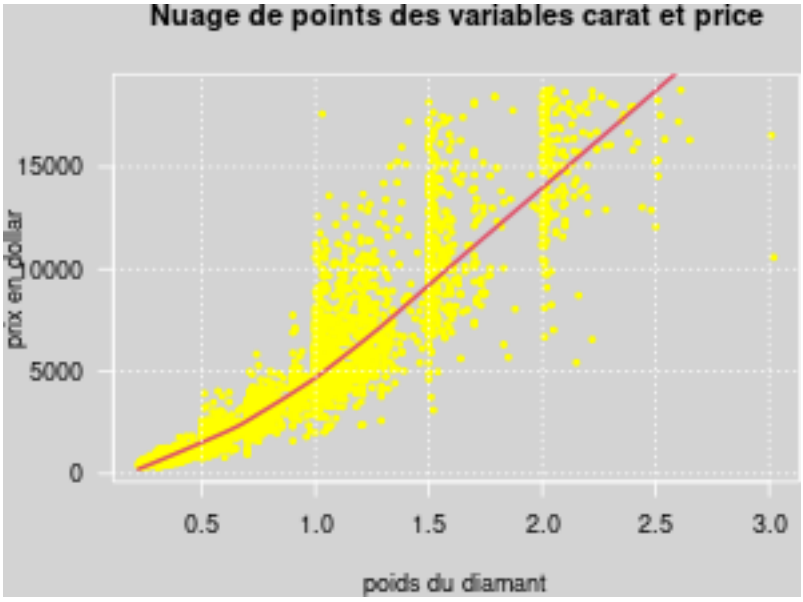
A ces trois grandes familles de fonctions, on pourrait également ajouter la famille des fonctions qui permettent de sauvegarder les fenêtres graphiques au format image bitmap ou vectorielle.

Cette "distinction" ne les rend pas pour autant dissociables. Au contraire, elles sont parfaitement complémentaires. Généralement, lorsqu'elles sont utilisées ensemble, la démarche est la suivante : d'abord, on définit un certain nombre de paramètres graphiques, ensuite, un graphique est construit avec une fonction de haut-niveau, enfin, quelques éléments informatifs (titres, légende, texte, ...) peuvent être ajoutés avec les fonctions de bas niveau.

**Exemple** : on propose un premier graphique pour illustrer notre propos. On va s'intéresser au lien entre la variable `price` (prix en dollar) et `carat` (poids du diamant) du jeu de données `diam_ech`. Pour cela, on va utiliser un nuage de points et représenter les observations avec des cercles remplis en jaune de taille petite. On représentera dans un second temps la courbe de régression non paramétrique de `price` en fonction de `carat`, avec un trait moyennement épais de couleur rouge. Enfin, on souhaite que le fond de ce graphique soit gris, avec un cadre de couleur blanche et une épaisseur de trait moyennement épaisse. Voici le code :

```
# Appel de la fonction par() pour un nouveau paramétrage
# graphique avec comme spécifications :
# - un fond en gris (bg=),
# - un cadre de couleur blanche (fg=)
# - des traits épais dans la figure (lwd=2, au lieu de 1 par défaut)
# NB : l'objet op stocke la valeur des paramètres
# précédant l'appel de la fonction par()
op <- par(bg = "lightgrey", # définit la couleur de fond de la figure
```

```
          fg = "white",     # définit la couleur du cadre + graduations + points/lignes
            pch = 16,         # définit le type de points
          las =  1)         # la graduation sur l'axe des ordonnées est horizontale
with(diam_ech, { # voir explication de with() ci-dessous
# Appel de la fonction de haut-niveau plot(x, y, ...) avec
# - x : le vecteur de numeric en abscisse
# - y : le vecteur de numeric en ordonnée
# et comme options :
# "pch" le symbole des points, "cex" la taille (1 par défaut), "col" la couleur
# "xlab" et "ylab" la légende sur les axes x et y
  plot(carat, price,
     pch = 20, cex = 0.8, col = "yellow",
     xlab = "poids du diamant",
     ylab = "prix en dollar")
# Appel de la fonction de bas-niveau lines() pour ajouter une droite de rég.
# non paramétrique (obtenue avec la fonction lowess())
  lines(lowess(carat, price), col = 2, lwd = 2)
}) # Fin de la fonction with()
# Appel de la fonction de bas-niveau title() pour ajouter un titre
title("Nuage de points des variables carat et price")
# Appel de la fonction de bas-niveau grid() pour une grille
grid(col = "white")
```




```
par(op) # retour au paramétrage graphique précédent stocké dans op
```

**Remarque** : la syntaxe de la fonction `with()` est

```
with(data, expr, ...)
```

L'objet `data` est le nom du jeu de données et `expr` est une expression (potentiellement sur plusieurs lignes à condition de mettre l'expression entre accolades) dans laquelle on peut appeler directement les variables du jeu de données `data` (par exemple `var1`, `var2`, etc.), plutôt que faire `data$var1`, `data$var2`, etc.

## 1.2 Gestion de la fenêtre graphique

### 1.2.1 Sous RStudio

En utilisant **RStudio**, à chaque fois qu'on fait appel à une fonction de haut-niveau, cela créé une nouvelle fenêtre graphique. On fait ensuite défiler les graphiques en utilisant les flèches allant vers la droite ou vers la gauche. L'inconvénient est que lorsqu'on ne fait pas attention à supprimer les fenêtres inutilisées (avec la croix rouge ou le pinceau), on peut se retrouver encombré par un grand nombre de fenêtres ouvertes.

### 1.2.2 Sous R

Dès qu'on fait appel à une fonction de haut-niveau, cela écrase le graphique précédent. C'est pourquoi on vous présente ces trois fonctions facilitant la manipulation des fenêtres graphiques. Attention, ces fonctions ne sont pas utiles dans **RStudio** (comme on l'a vu, l'ouverture, le changement et la fermeture des fenêtres graphiques se font directement depuis le menu de la fenêtre "Plots") :

- Si on souhaite ouvrir une nouvelle fenêtre graphique dans **R**, on utilise la fonction `dev.new()`.
- Lorsqu'on a plusieurs fenêtres graphiques ouvertes, on peut passer de l'une à l'autre en utilisant la fonction `dev.set()` en précisant à l'intérieur de la parenthèse le numéro de la fenêtre (qui commence à 2 et dont le numéro est précisé au-dessus de chaque fenêtre graphique ouverte).
- pour fermer correctement une fenêtre graphique, on utile la fonction `dev.off()` en précisant entre parenthèses le numéro de la fenêtre à fermer.

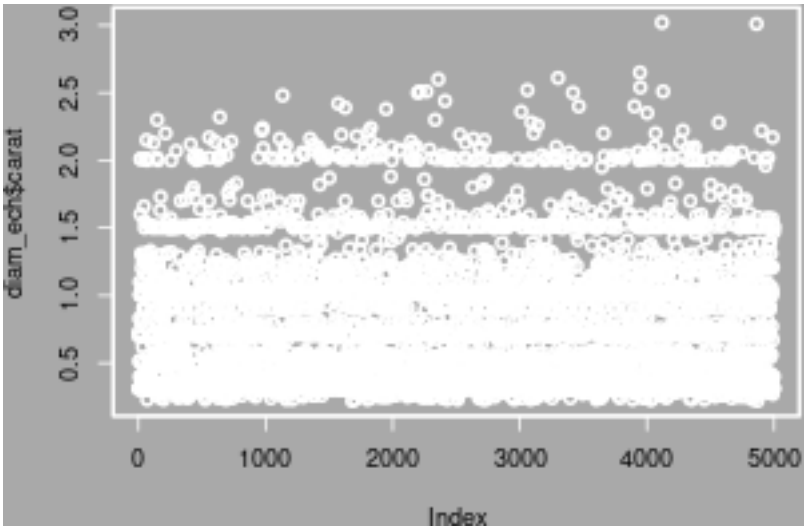
## 1.3 Paramétrage de la fenêtre graphique

On a vu dans l'exemple précédent, qu'il était possible de modifier les paramètres graphiques soit de "façon globale"‘ (les paramètres graphiques seront conservés sur tous les nouveaux graphiques), soit de façon locale (les paramètres graphiques ne seront effectifs que sur le graphique courant) à l'intérieur de chaque fonction de haut et bas niveau.

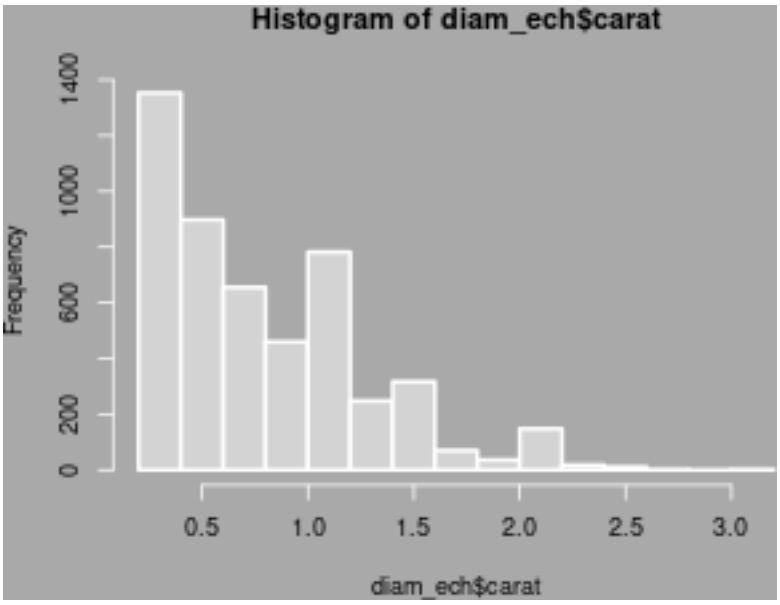
### 1.3.1 Utilisation de la fonction `par()`

La commande `par()` permet de définir de nouveaux paramètres graphiques qui seront automatiquement utilisés dans les fonctions de haut et bas niveau qui seront appelées après l'appel de `par()`. La commande `op<-par(...)` permet de sauvegarder les anciens paramètres graphiques qui ont été modifiés dans l'objet `op`. Il faut savoir qu'à partir du moment où on définit de nouveaux paramètres avec la fonction `par()`, pour ensuite s'en débarasser, il faut soit fermer la fenêtre graphique en cours, soit utiliser la commande `par(op)` à la fin des commandes graphiques. Pour vous en convaincre, exécuter les lignes de code suivantes :

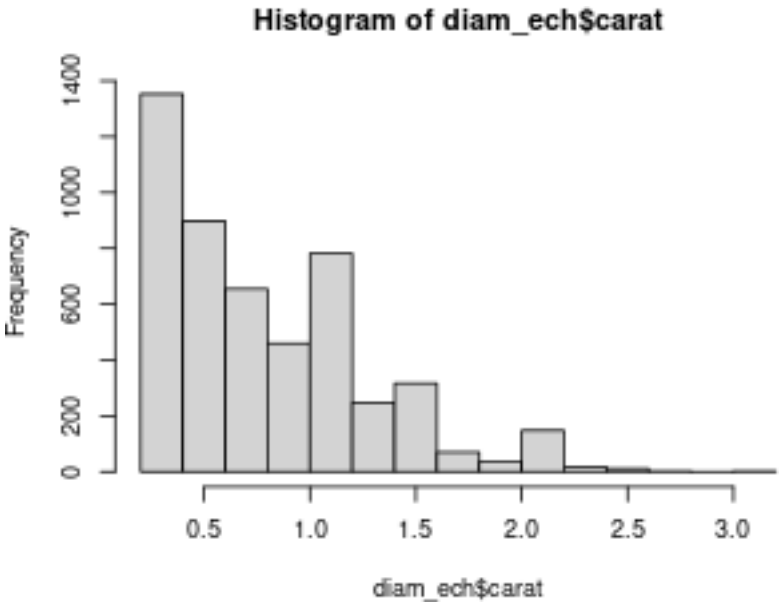
```
# Appel de la fonction par() pour un nouveau paramétrage
op <- par(bg = "darkgrey", fg = "white", lwd = 2)
# Appel de la fonction de haut-niveau plot(x, ...)
# - avec x le vecteur de numeric à représenter en ordonnées. La valeur en abscisse
# sera la position (i.e. l'indice de x) allant de 1 jusqu'à la taille de x
plot(diam_ech$carat)
```




```
# Appel d'une autre fonction de haut-niveau : hist()
# le nouveau paramétrage graphique a été conservé...
hist(diam_ech$carat)
```




```
par(op) # pour retourner au paramétrage graphique initial
hist(diam_ech$carat)
```




**Remarque :** l'option `fg=` définit à la fois la couleur du cadre, des graduations et des points/lignes/barres.

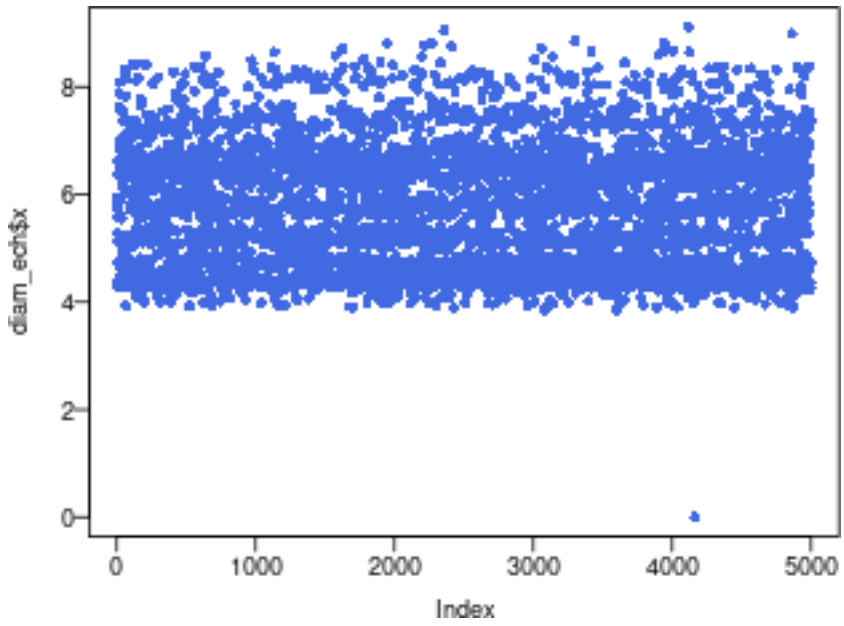
Grâce à la fonction `par()`, on peut modifier de nombreux paramètres graphiques : aussi bien des paramètres concernant la fenêtre graphique elle-même (couleur de fond, couleur du cadre, nombre de divisions de la fenêtre, etc.) que les paramètres concernant le graphique lui-même (couleur des points, taille des traits, taille de la police, couleur des axes, etc.). Quand on regarde le nombre d'options de la fonction `par()`, on en compte plus de 70, ce qui a de quoi faire tourner la tête...

Lorsqu'on regarde plus en détails l'aide de la fonction `par()`, on constate que la plupart des arguments d'entrée qui sont décrits sont les mêmes arguments que ceux utilisés dans la plupart des fonctions de haut-niveau (`plot()`, `hist()`, `barplot()`, etc.) et bas-niveau (`lines()`, `points()`, etc.). Par ailleurs, lorsqu'on regarde l'aide de la fonction `plot()`, hormis certains arguments d'entrée qui sont propres à la fonction `plot()` (c'est le cas des arguments `type`, `main`, `sub`, `xlab`, `ylab`, `asp`), on nous renvoie vers l'aide de la fonction `par()` pour plus de détails sur tous les paramètres graphiques qu'on peut utiliser.

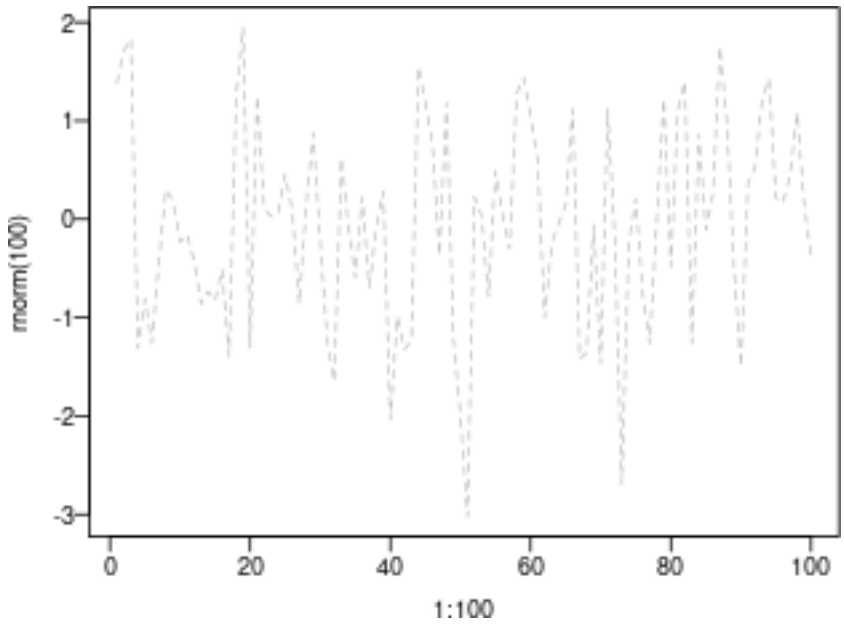
En pratique, on procède ainsi :

- On modifie (si besoin) les paramètres qui sont propres à la fonction `par()`, comme par exemple la largeur des marges (nous verrons dans la section suivante en détails comment cela fonctionne).
- On utilise ensuite les arguments des fonctions de haut-niveau, comme par exemple la taille des points, l'épaisseur des traits, etc.

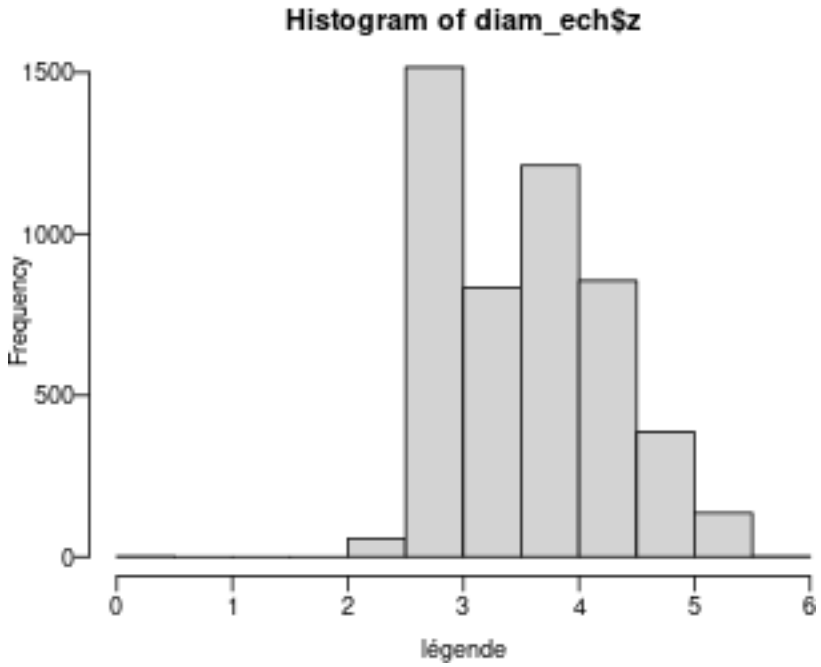
```
# Appel de la fonction par() pour un nouveau paramétrage
op <- par(mar = c(3, 3, 2, 1), # taille des marges
          mgp = c(2, 0.5, 0),  # emplacement des légents/annotations/graduations dans les marges
          las = 1)  # pour changer l'orientation des annotations sur l'axe y
# Appel d'une fonction de haut-niveau
plot(diam_ech$x,
     col = "royalblue",        # couleur des points
     pch = 16)                 # type des points
```




```
# Appel d'une fonction de haut-niveau
# on garde ici le nouveau paramétrage graphique
plot(1:100, rnorm(100),
     type = "l",              # indique de relier les points par traits
     col = "grey",
     lwd = 0.9,               # épaisseur du trait
     lty = 2)                 # type du trait, ici en pointillés
```




```
# Appel d'une fonction de haut-niveau
# on garde ici le nouveau paramétrage graphique
hist(diam_ech$z, xlab = "légende")
```

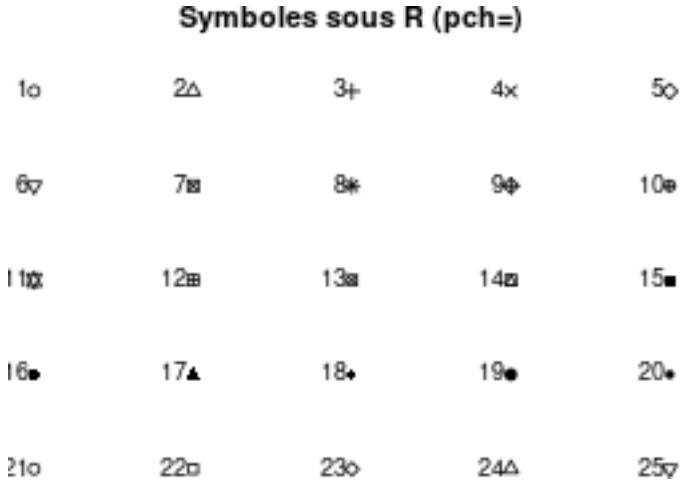



```
par(op) # retour au paramétrage graphique initial
```

### 1.3.2 Les différents symboles pour les points

On peut représenter les points par des symboles différents. C'est l'argument `pch=` (suivi d'un nombre compris entre 1 et 25) qui permet de modifier les symboles et qu'on retrouve par exemple dans la fonction `plot()` ou `points()`. On a représenté ci-dessous les symboles possibles avec le numéro correspondant.

```
plot(rep(1:5, 5), rep(5:1, each = 5), pch = 1:25, axes = FALSE,
     xlab = "", ylab = "", main = "Symboles sous R (pch=)")
text(rep(1:5), rep(5:1, each = 5), as.character(1:25),
     pos = 2, offset = 0.2)
```




**Remarque** : nous avons utilisé un certain nombre de paramètres graphiques ci-dessus. Dans la fonction `plot()`:

- `axes=FALSE` permet de ne pas représenter les axes des abscisses ou des ordonnées,
- `xlab=""` donne une légende sur l'axe des abscisses qui est vide,

- `ylab=""` donne une légende sur l'axe des ordonnées qui est vide,
- `main=` donne un titre au graphique.

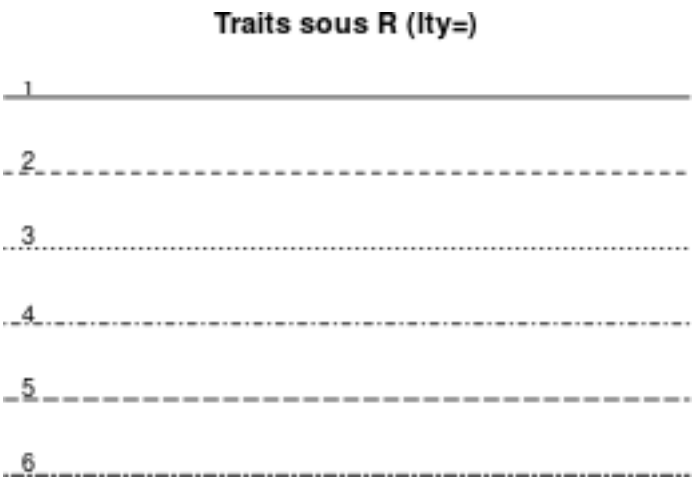
La fonction `text(x, y, labels)` permet de représenter des étiquettes dans un graphique. Les deux premiers arguments `x` et `y` correspondent aux coordonnées du point où sera représentée l'étiquette (l'argument `labels`). Les options que nous avons utilisées sont :

- `pos=2` permet d'afficher l'étiquette à gauche des coordonnées (1=en-dessous, 2=à gauche, 3=au-dessus, 4=à droite)
- `offset` donne la distance entre les coordonnées et l'étiquette

### 1.3.3 Les différents traits pour les lignes

On peut représenter les lignes par des traits différents. C'est l'argument `lty=` (suivi d'un nombre compris entre 1 et 6) qui permet de modifier le type de traits et qu'on retrouve dans les fonctions `plot(, type="l")` ou `lines()` par exemple. On a représenté ci-dessous les traits possibles avec le numéro correspondant.

```
plot.new()
title("Traits sous R (lty=)")
abline(h = rev(seq(0, 1, 0.2)), lty = 1:6)
text(0, rev(seq(0, 1, 0.2)), as.character(1:6), pos = 3,
     offset = 0.1)
```




**Remarque** : nous avons utilisé deux nouvelles fonctions dans l'exemple ci-dessus :

- la fonction `plot.new()` (haut-niveau) ouvre une nouvelle fenêtre graphique dans le cadre $[0,1] \times [0,1]$ sans représenter aucun axe, ni aucune autre information.
- la fonction `abline()` (bas-niveau) permet de représenter soit une droite horizontale (`abline(h=)`), soit une droite verticale (`abline(v=)`), soit une droite de régression linéaire (`abline(a=,b=)` où `a` est la constante et `b` la pente de la droite).

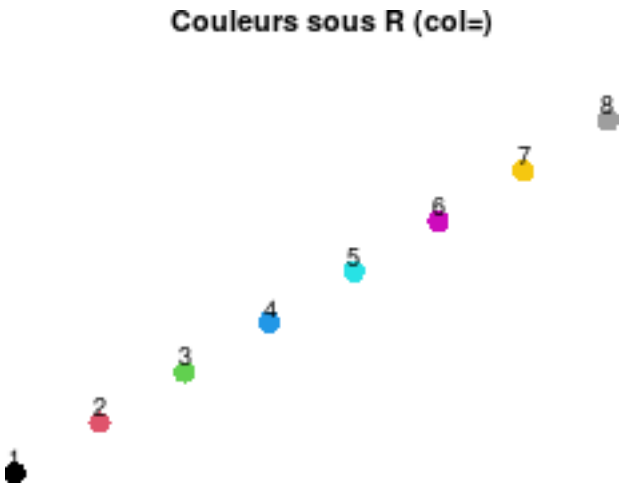
### 1.3.4 Les différentes couleurs

On peut représenter les points ou les lignes par des couleurs différentes. C'est l'argument `col=` qui peut prendre comme valeur :

- un nombre compris entre 1 et 8 pour les couleurs de base

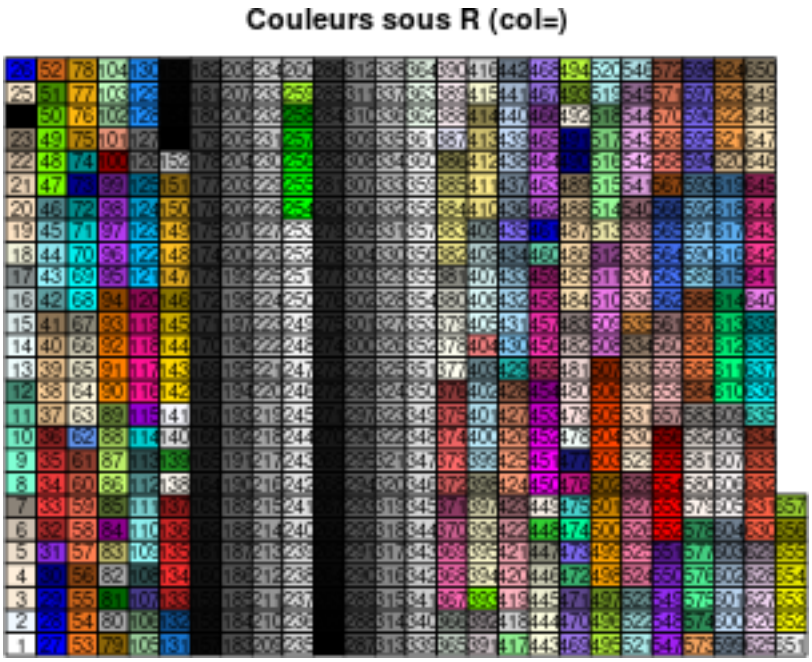
```
plot(1:8, col = 1:8, pch = 16, cex = 2, axes = FALSE,
     xlab = "", ylab = "", main = "Couleurs sous R (col=)",
     xlim = c(1, 8.5), # donne les bornes de l'axe des abscisses
```

```
     ylim = c(1, 8.5)) # donne les bornes de l'axe des ordonnées
text(1:8, as.character(1:8), pos = 3, offset = 0.2)
```




- un nom de couleurs parmi les 657 noms disponibles dans la fonction `colors()`. Pour représenter la palette de couleurs disponibles, exécuter les lignes de codes suivantes :

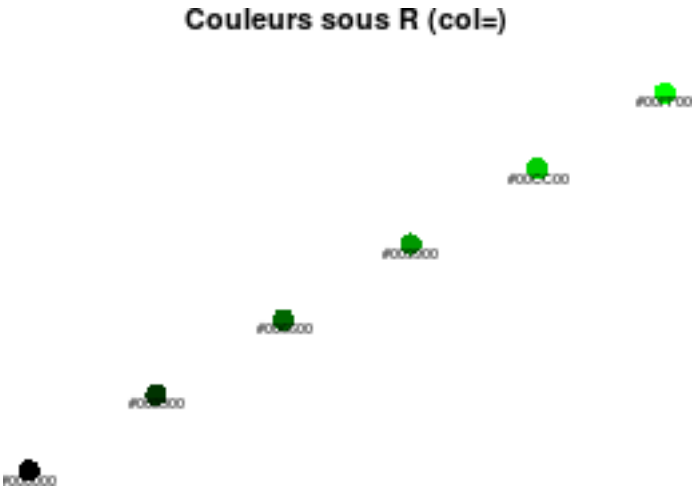
```
op <- par(mar = c(0, 0, 1, 0)) # régle les marges
plot(1:27, type = "n", axes = FALSE,
     xlab = "", ylab = "", main = "Couleurs sous R (col=)")
k <- 0 # k, indice du vecteur des couleurs à parcourir
for(i in 1:26) { # on fait varier les lignes
 for(j in 1:26) { # on fait varier les colonnes
   k <- k + 1
    if (k > 657)
      break # Taille du vecteur à ne pas dépasser
   polygon(c(i, i + 1, i + 1, i, i),
           c(j, j, j + 1, j + 1, j),
          col = colors()[k])
   text(i + 0.5, j + 0.5, as.character(k), cex = 0.8)
 }
}
```




```
par(op)
```

- en utilisant un code alpha-numérique obtenu avec la fonction `rgb()` auquelle on donne comme arguments d'entrée le pourcentage de Rouge, Vert et Bleu de la couleur à représenter. Par exemple :

```
code.coul <- rgb(0, seq(0, 1, 0.2), 0)
plot(1:6, col = code.coul, pch = 16, cex = 2, axes = FALSE,
     xlab = "", ylab = "", main = "Couleurs sous R (col=)")
text(1:6, as.character(code.coul), pos = 1, offset = 0.2, cex = 0.6)
```



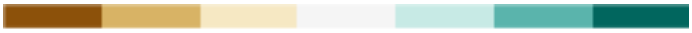

- Enfin, on citera la fonction `brewer.pal()` du package **RColorBrewer** qui retourne des palettes de couleurs déjà préparées, certaines à base de de dégradé. Par exemple, pour créer un dégradé de couleurs sur une teinte de vert :

```
library(RColorBrewer)
mypalette <- brewer.pal(7, "Greens")
image(1:7, 1, as.matrix(1:7), col = mypalette, xlab = "Greens (sequential)",
       ylab = "",xaxt = "n", yaxt = "n", bty = "n")
```

Greens (sequential)

Sinon, pour utiliser une palette appelé “BrBG”

```
mypalette <- brewer.pal(7, "BrBG")
image(1:7, 1, as.matrix(1:7), col = mypalette, xlab = "BrBG",
       ylab = "",xaxt = "n", yaxt = "n", bty = "n")
```

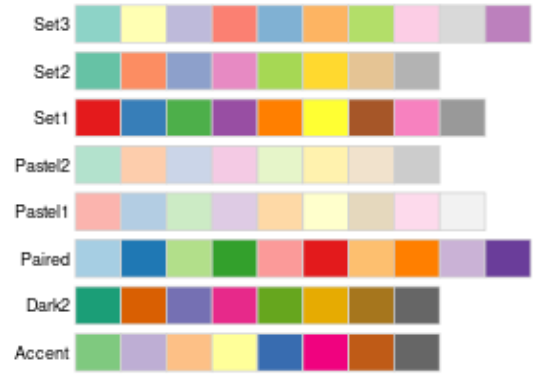


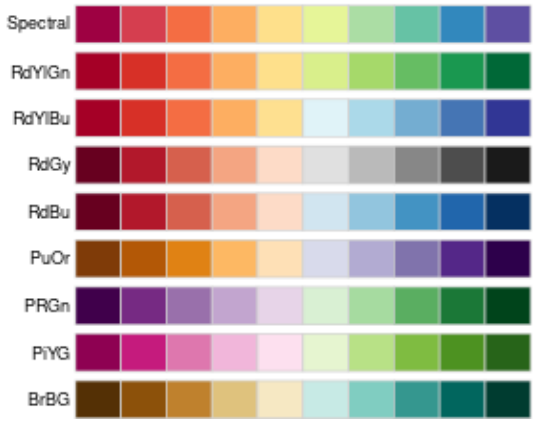
BrBG

On représente ici l’ensemble des palettes disponibles qui sont regroupés en 3 familles : les palettes séquentielles, les palettes qualitatives, les palettes divergentes.

```
display.brewer.all(n=10, exact.n=FALSE)
```


YlOrRd
YlOrBr
YlGnBu
YlGn
Reds
RdPu
Purples
PuRd
PuBuGn
PuBu
OrRd
Oranges
Greys
Greens
GnBu
BuPu
BuGn
Blues

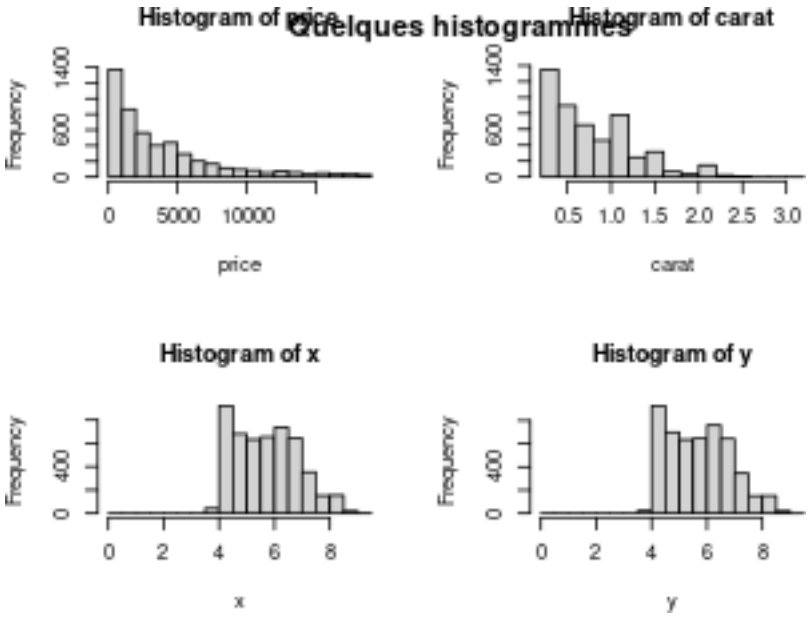

Set3
Set2
Set1
Pastel2
Pastel1
Paired
Dark2
Accent


Spectral
RdYlGn
RdYlBu
RdGy
RdBu
PuOr
PRGn
PiYG
BrBG

### 1.3.5 Partitionner la figure en plusieurs zones

- Une première possibilité consiste à modifier le paramètre graphique `mfrow=c(n,p)` de la fonction `par()` où $n$ est le nombre de lignes et $p$ le nombre de colonnes. Typiquement, pour partitionner en 4 la fenêtre graphique (2 lignes et 2 colonnes), on fait :

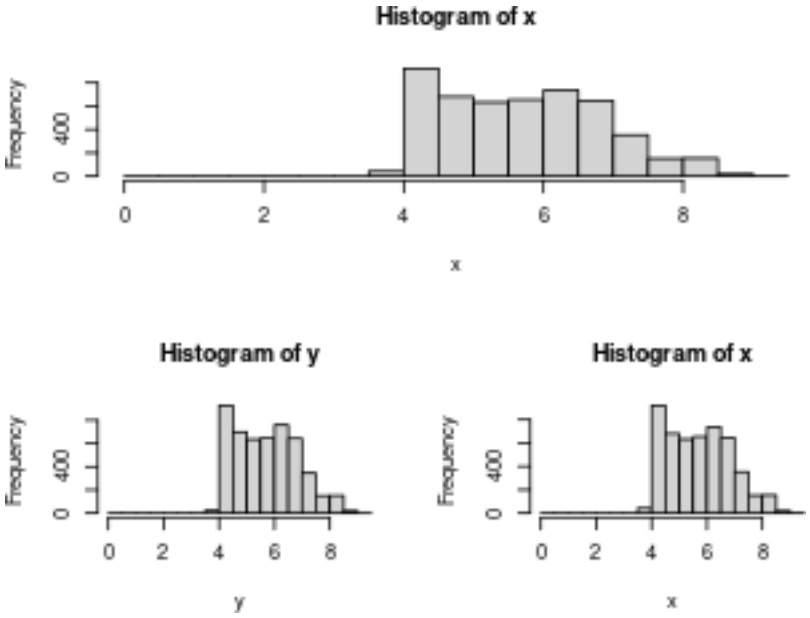
```
op <- par(mfrow = c(2, 2))
with(diam_ech,{
  hist(price)
  hist(carat)
  hist(x)
  hist(y)
})
par(op)
title("Quelques histogrammes")
```




**Remarque** : pour donner un titre à l'ensemble du graphique, on a utilisé la fonction `title()` juste après avoir réinitialiser les paramètres par défaut. En revanche, on ne peut pas être trop satisfait de l'endroit où il est situé et on verra juste après comment on peut remédier à cela.

- Une seconde possibilité consiste à utiliser la fonction `layout()` qui a pour 1er argument une matrice à $n$ lignes et $p$ colonnes, remplies de valeurs entières consécutives allant de 1 jusqu'à un chiffre inférieur ou égal à $np$. Les valeurs entières de la matrice correspondent aux numéros des sous-fenêtres, l'idée étant de s'arranger pour que les cases portant le même numéro forment des blocs contigus. Par exemple :

```
mat <- matrix(c(1, 1, 2, 3), 2, 2, byrow = TRUE)
layout(mat)
  with(diam_ech,{
  hist(x)
  hist(y)
  hist(x)
})
```




permet une partition intéressante, mais si on avait pris la matrice suivante :

```
mat <- matrix(c(1, 2, 3, 1), 2, 2, byrow = TRUE)
```

cela aurait créé un problème dans la représentation graphique...

**Remarque** : pour plus d'informations sur le partitionnement de la fenêtre graphique, le lecteur pourra consulter cette page web.

### 1.3.6 Modifier la taille des marges dans la fenêtre graphique

Lorsqu'on ne représente qu'un seul graphique dans une figure, pour modifier la taille des marges, il faut modifier un des paramètres graphiques suivants de la fonction `par()` : `mar=` ou `mai=`. Ces deux paramètres sont équivalents, mais sont exprimés dans deux mesures différentes. Il suffit donc de choisir un des deux.

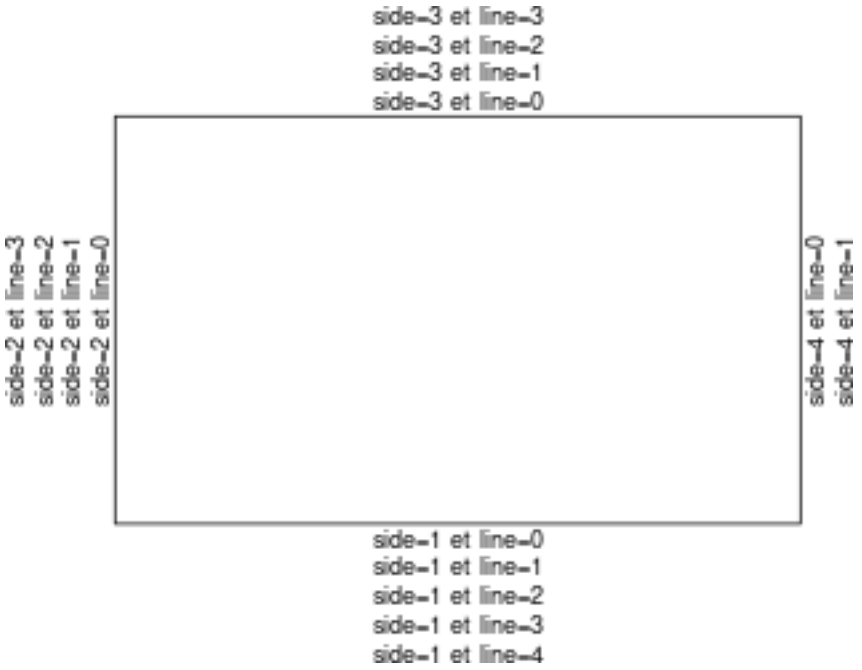
- `mar=` est exprimé en nombre de lignes. Pour le modifier, on écrit `mar=c(bottom, left, top, right)` où `c(bottom, left, top, right)` est un vecteur de taille 4, où le premier élément indique la marge en-dessous (bottom en anglais) du cadre graphique, le deuxième élément la marge à gauche du cadre graphique, le troisième élément la marge au-dessus du cadre graphique et le 4ème élément la marge à droite du cadre graphique. Par défaut, le paramétrage est égal :

```
par("mar")
```

```
## [1] 5.1 4.1 4.1 2.1
```

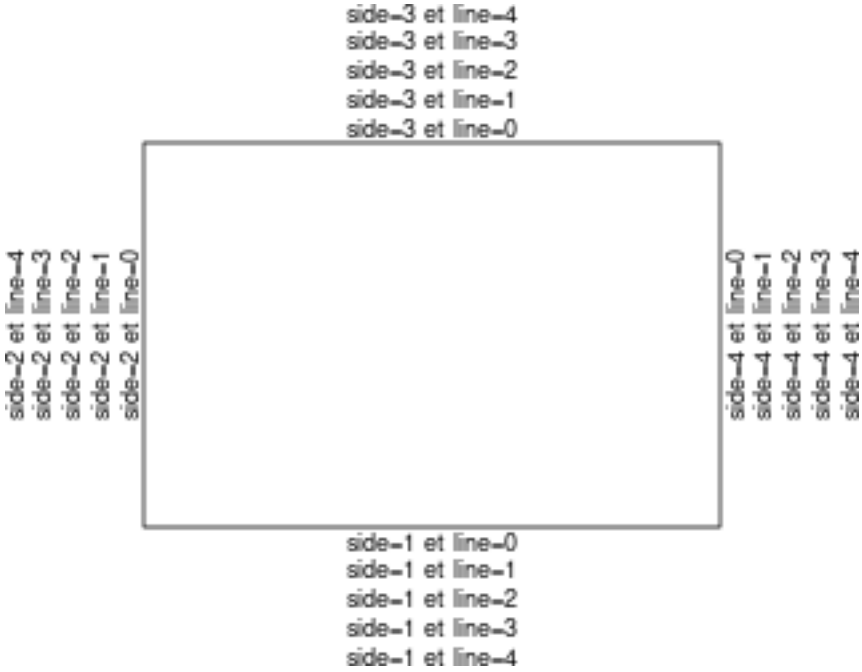
Dans l'exemple ci-dessous, on a rempli à l'aide de la fonction `mtext()` et des paramètres `side=` (1 pour au-dessous, 2 pour à gauche, 3 pour au-dessus et 4 pour à droite du cadre) et `line=` les différents espaces réservés pour les marges. On voit donc bien que les valeurs `c(5.1, 4.1, 4.1, 2.1)` correspondent au nombre de lignes autorisées autour du cadre graphique (5 lignes réservées en-dessous, 4 lignes réservées à gauche, 4 lignes au-dessus et 1 ligne à droite).

```
plot(1:10, ann = FALSE, type = "n", xaxt = "n", yaxt = "n")
for(j in 1:4)
  for(i in 0:4)
    mtext(paste0("side=", j, " et line=", i), side = j, line = i)
```




Si on modifie le paramètre `mar=`, on impacte donc les espaces réservés à ces légendes :

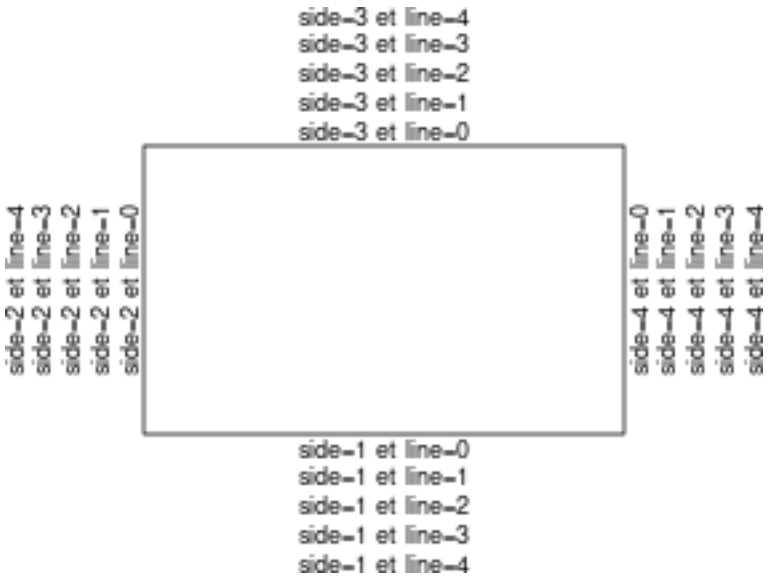
```
op <- par(mar = c(5, 5, 5, 5))
plot(1:10, ann = FALSE, type = "n", xaxt = "n", yaxt = "n")
for(j in 1:4)
  for(i in 0:4)
    mtext(paste0("side=", j, " et line=", i), side = j, line = i)
```




```
par(op)
```

- `mai=c(bottom, left, top, right)` est exprimé quant à lui en "inches". En gros, 0.2 inches est l'équivalent d'1 ligne. Autrement dit, pour reproduire le graphique précédent, on aurait pu faire :

```r
op <- par(mai = c(1, 1, 1, 1))
plot(1:10, ann = FALSE, type = "n", xaxt = "n", yaxt = "n")
for(j in 1:4)
  for(i in 0:4)
    mtext(paste0("side=", j, " et line=", i), side = j, line = i)
```




```r
par(op)
```

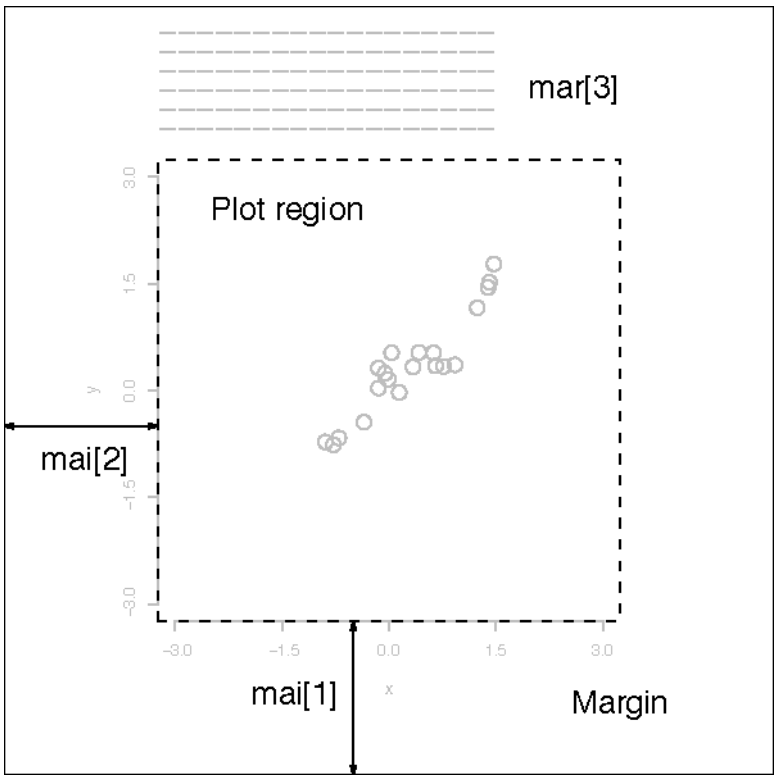
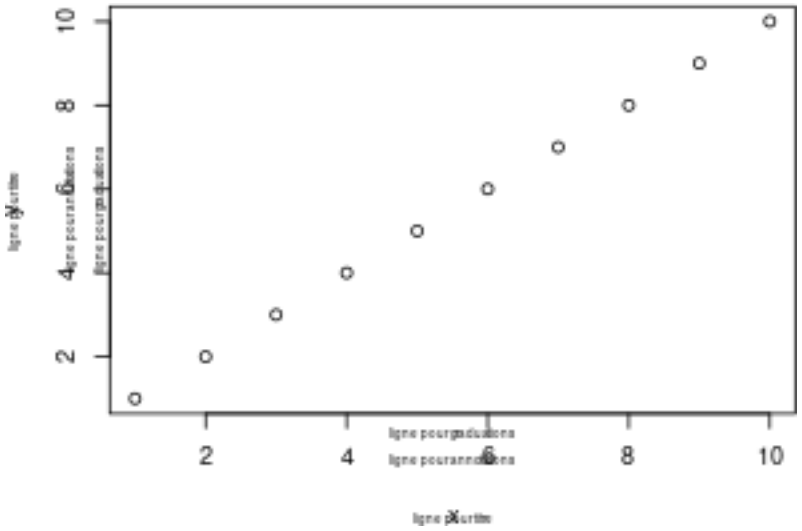



Figure 1: Les paramètres mai/mar

### 1.3.7 Modifier les graduations/étiquettes/titres des axes $x/y$

Par défaut, on a vu que les marges étaient définies de la sorte : 5 lignes réservées en-dessous, 4 lignes réservées à gauche, 4 lignes au-dessus et 1 ligne à droite. Lorsqu'on dessine un graphique, il y a des informations qui s'affichent automatiquement; c'est le cas notamment des graduations/étiquettes/titres associés aux axes des $x$ et $y$. Dans le graphique ci-dessous, on constate que les graduations des $x$ et des $y$ sont situées sur la ligne 0 (première ligne en partant du cadre); les étiquettes sont situées sur la ligne 1 (deuxième ligne en partant du cadre) et les légendes sont situées sur la ligne 3 (quatrième ligne en partant du cadre).

```
plot(1:10, xlab = "x", ylab = "y")
mtext("ligne pour graduations", side = 1, line = 0, cex = 0.5)
mtext("ligne pour graduations", side = 2, line = 0, cex = 0.5)
mtext("ligne pour annotations", side = 1, line = 1, cex = 0.5)
mtext("ligne pour annotations", side = 2, line = 1, cex = 0.5)
mtext("ligne pour titre", side = 1, line = 3, cex = 0.5)
mtext("ligne pour titre", side = 2, line = 3, cex = 0.5)
```



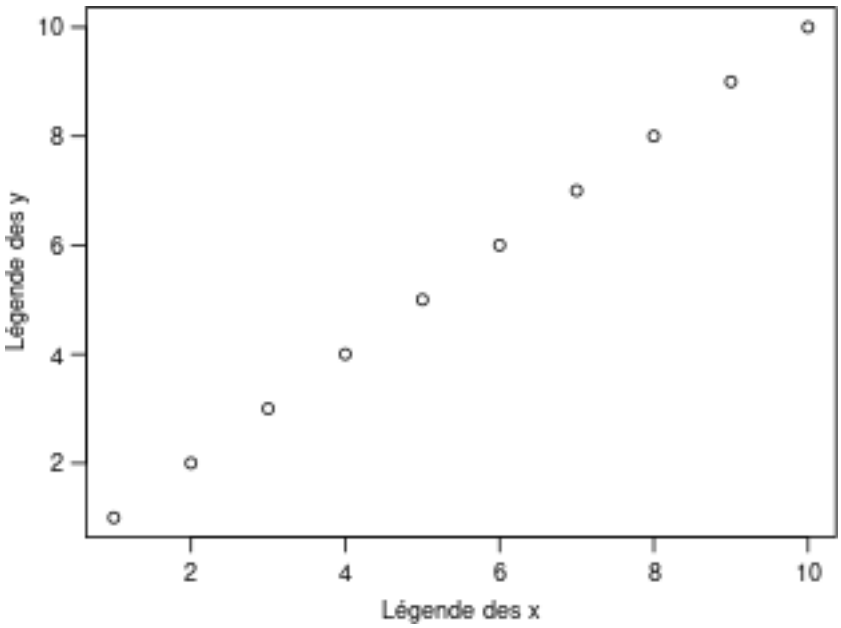

Cette information est donnée par l'argument `mgp` :

```
par("mgp")
```

```
## [1] 3 1 0
```

Pour changer ces dispositions, il faut donc modifier ce paramètre. Dans l'exemple ci-dessous, on a également modifier les marges :

```
op <- par(mar = c(3, 3, 2, 1),
          mgp = c(2, 0.75, 0),
          las = 1)  # pour changer l'orientation des annotations sur l'axe y
plot(1:10, xlab = "Légende des x", ylab = "Légende des y")
```

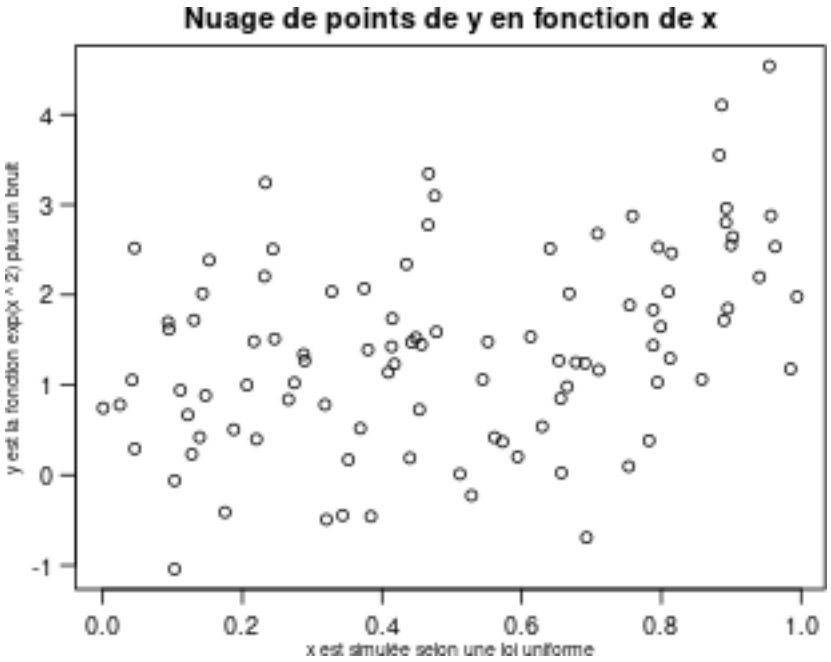



```
par(op)
```

**Exercice 4.1**

Trouver le code qui permet d'obtenir le graphique suivant où `x` et `y` sont définis ainsi:

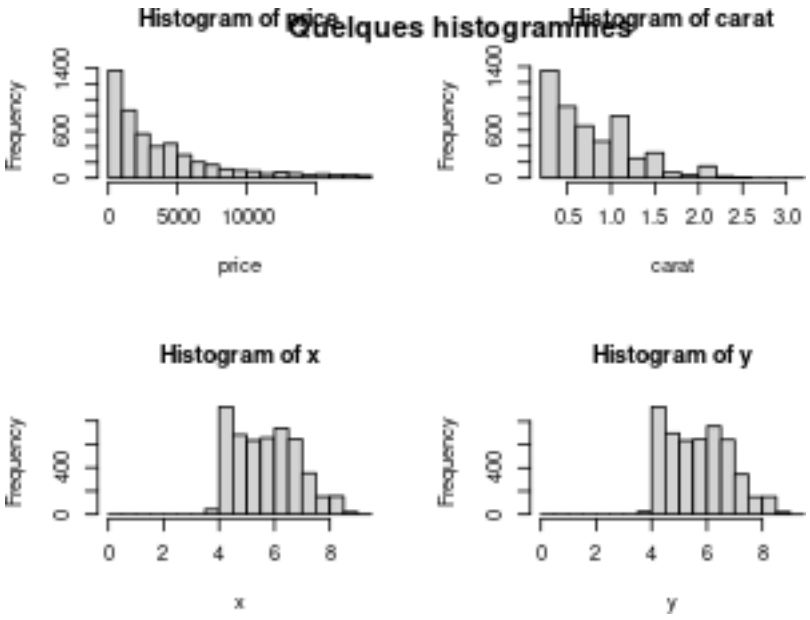
```
set.seed(123)
x <- runif(100, 0, 1)
y <- exp(x ^ 2) + rnorm(100)
```




### 1.3.8 Modifier les marges dans une figure avec plusieurs graphiques

Revenons à l'exemple vu précédemment où on a représenté 4 graphiques dans une même figure.

```
op <- par(mfrow = c(2, 2))
with(diam_ech,{
  hist(price)
  hist(carat)
  hist(x)
  hist(y)
})
par(op)
title("Quelques histogrammes")
```




On remarque dans la figure ci-dessus que le titre principal est situé sur l'emplacement des deux graphiques du dessus. Pour éviter cela, il faudrait ajouter des marges autour de l'emplacement réservé aux graphiques. Pour faire cela, il faut utiliser un deux paramètres suivants : `oma=` (exprimé en nombre de ligne) ou `omi=` (exprimé en "inches"). Par défaut le paramètrage vaut :

```
par("oma")
```

```
## [1] 0 0 0 0
```

Autrement dit, il n'y a par défaut aucune marge autour de la zone dédié au(x) graphiques.

**Application** : pour modifier la figure ci-dessus et laisser de la marge au titre global, on va également rétrécir les marges à l'intérieur de chaque graphique (paramètre `mar` vu ci-dessus). Pour écrire du texte dans les marges extérieures au cadre graphique, on utilise encore la fonction `mtext()` avec l'argument `outer=TRUE` pour indiquer qu'on se place en-dehors de l'emplacement réservé aux graphiques.

```
op <- par(mfrow = c(2, 2), oma = c(1, 1, 3, 0), mar = c(2, 3, 3, 1),
          mgp = c(2, 1, 0))
with(diam_ech,{
  hist(price)
  hist(carat)
  hist(x)
  hist(y)
})
mtext("Référence : données diamonds",
      side = 1, line = 0, cex = 0.5, col = "blue", outer = TRUE)
```

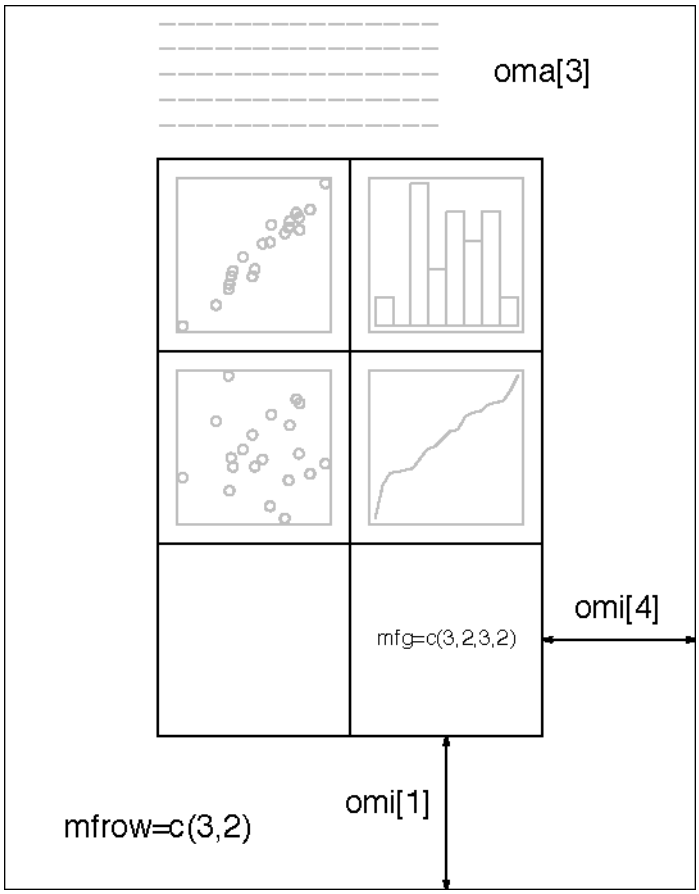
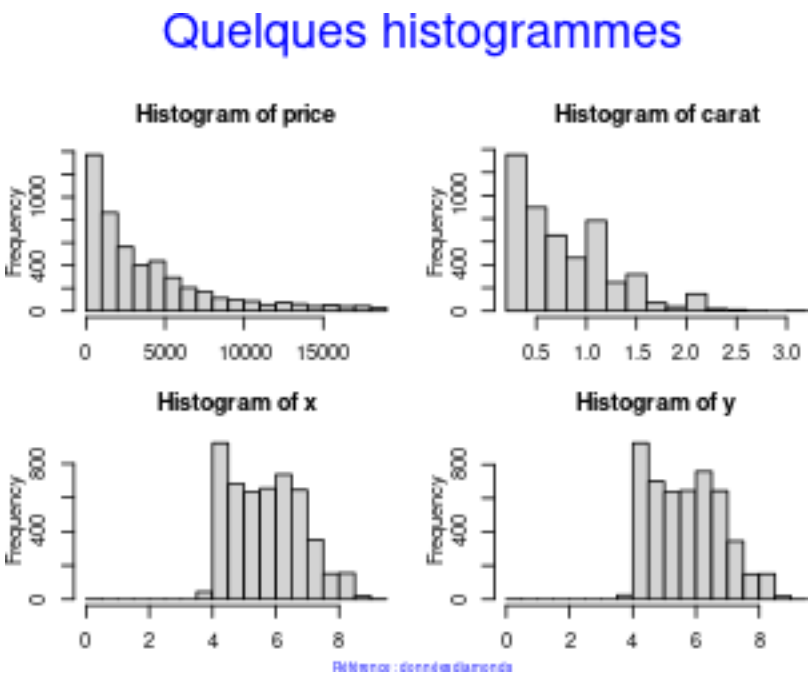



Figure 2: Les paramètres oma/omi

```
mtext("Quelques histogrammes",
      side = 3, line = 1, cex = 2, col = "blue", outer = TRUE)
```




```
par(op)
```

### 1.3.9 Obtenir un style graphique proche de celui de ggplot2

Nous verrons dans la section suivante comment faire un graphique en utilisant la syntaxe **ggplot2**. On va essayer ici de reproduire un graphique qui aurait le même style que celui d'un graphique **ggplot2** en utilisant les fonctions de base et paramètres graphiques vus jusqu'à présent. Pour cela, on va devoir utiliser différentes fonctions et paramètres. Le cheminement est celui-ci :

- la fonction `par()` permet de définir un certain nombre de paramètres sur la figure, notamment les marges.
- on appelle la fonction `plot()` mais on ne représente aucun point ni aucune légende. Le but est de définir automatiquement les limites de la fenêtre graphique.
- on colorie en gris clair, à l'aide de la fonction `rect()`, l'intérieur de la figure. Pour déterminer les limites, on utilise la commande `par()$usr`.
- à l'aide de la fonction `abline()`, on trace des lignes horizontales (argument `h=`) et verticales (argument `v=`) qui sont supposées aider la lecture du graphique.
- la fonction `points()` permet de représenter les points
- la fonction `axis()` permet de représenter les graduations et annotations.

```
op <- par(mar = c(2.25, 2.25, 0.5, 0.5), # marge
          mgp = c(1.25, 0.75, 0),        # emplacement annot/étiquettes/légendes
          cex.axis = 0.75, # tailles des anotations
          cex.lab = 0.85, # tailles des légendes
          pch = 16,       # type de point utilisé
          las = 1,      # affiche les étiquettes |_ aux axes
          bty = "n"    # pas de box
          )
```

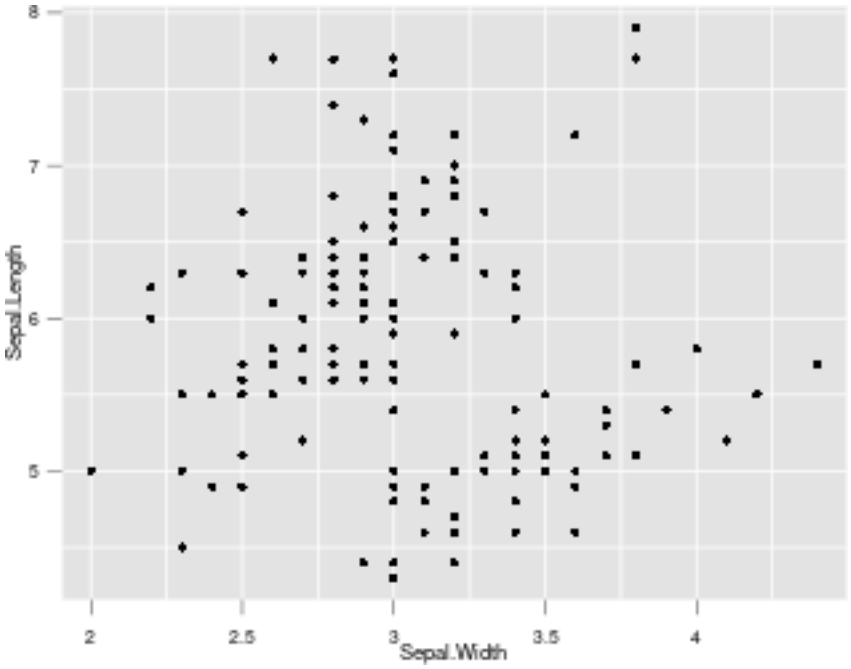
```r
plot(Sepal.Length ~ Sepal.Width,
     data = iris,
     axes = F,  # ne représente pas les axes des x et y
     type = "n") # ne représente aucun point

rect(par()$usr[1], par()$usr[3],
     par()$usr[2], par()$usr[4],
     col = "grey89", border = "white")

abline(h = seq(4.5, 8, 0.5), col = "white")
abline(v = seq(2, 4.5, 0.25), col = "white")

points(Sepal.Length ~ Sepal.Width,
     data = iris,
     cex = 0.8)   # taille des points

axis(side = 1,
     at = seq(2, 4, 0.5),  # ou mettre les graduations
     labels = seq(2, 4, 0.5), # quelles étiquettes
     lwd = 0,  # supprime la ligne de l'axe des abscisses
     lwd.ticks = 0.5)  # donne l'épaisseur des graduations
axis(side = 2,
     at = seq(5, 8, 1),
     labels = seq(5, 8, 1),
     lwd = 0,
     lwd.ticks = 0.5)
```



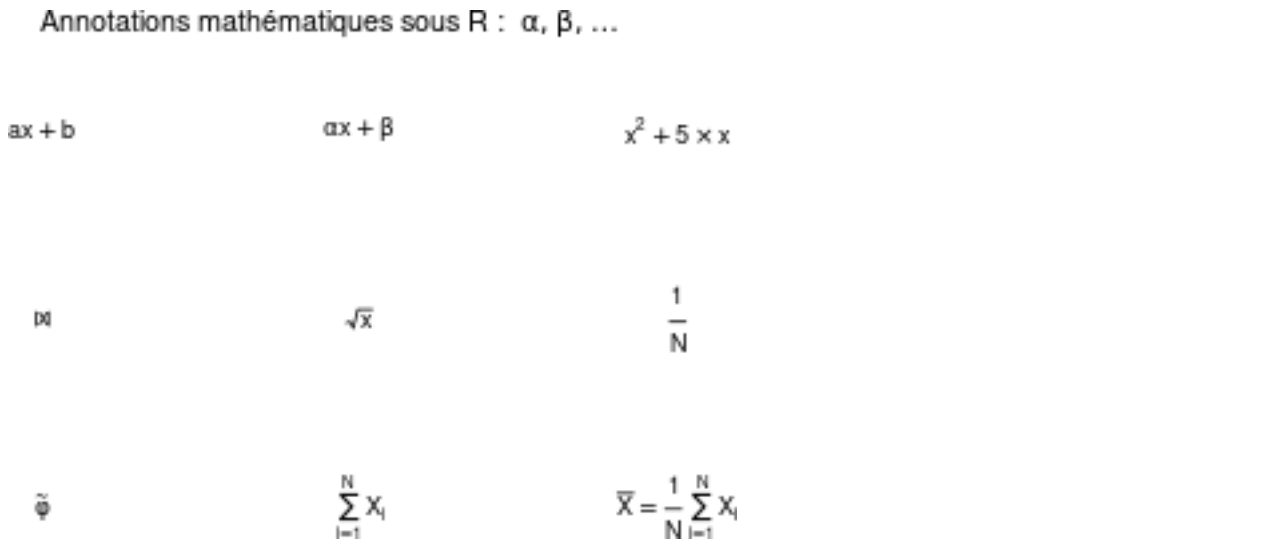

```r
par(op)
```

**Remarque :** on voit ici qu'il a fallu utiliser plusieurs étapes et fonctions pour arriver à construire un graphique à la **ggplot2**. Il faut ainsi garder en tête que le package **ggplot2** exécute plusieurs fonctions et programmes qui sont invisibles par l'utilisateur !

### 1.3.10 Intégrer une formule mathématique dans un graphique

Pour représenter des symboles mathématiques dans une fenêtre graphique, l'idée est de faire appel à un langage qui ressemble à LaTeX. Pour indiquer à **R** qu'il s'agit bien d'une expression particulière, on utilise la fonction `bquote()` ou `expression()`. Nous ne rentrerons pas dans les détails car ces fonctions sont plus rarement utilisées : nous présentons ici quelques exemples et nous renvoyons le lecteur à l'article An approach to Providing Mathematical Annotation in Plots de Paul Murrel et Ross Ihaka (2000) pour plus de précisions.

```
plot.new()
op <- par(mar = c(0, 0, 2, 0))
title(bquote("Annotations mathématiques sous R : "~list(alpha, beta, ...)))
text(rep(seq(0, 1, 0.5), 3), rev(rep(seq(0, 1, 0.5), each = 3)),
     as.expression(list(bquote(a*x+b), bquote(alpha*x+beta),
                        bquote(x^2+5%*%x), bquote(abs(x)),
                        bquote(sqrt(x)), bquote(over(1,N)),
                        bquote(tilde(phi)), bquote(sum(X[i],i==1,N)),
                        bquote(bar(X) == over(1,N) ~ sum(X[i],i==1,N))
                       )
                      )
     )
```

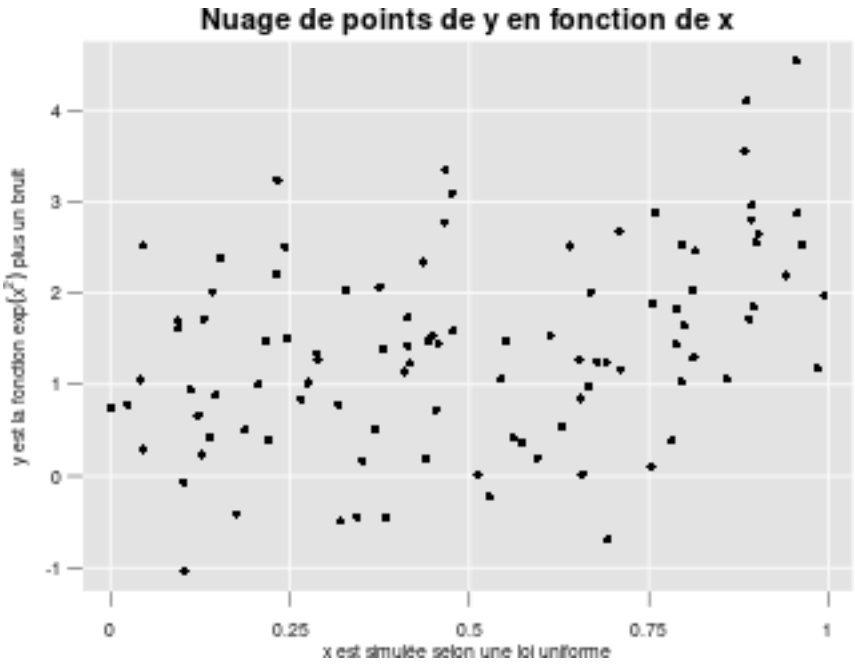



```
par(op)
```

**Exercice 4.2**

Trouver le code qui permet d'obtenir le graphique suivant où `x` et `y` sont définis ainsi:

```
set.seed(123)
x <- runif(100, 0, 1)
y <- exp(x ^ 2) + rnorm(100)
```




### 1.3.11 Sauvegarder son graphique

On peut utiliser directement le menu au-dessus des fenêtres graphiques qui permet facilement de sauvegarder un graphique au format *.pdf* ou parmi les autres formats *.jpeg*, *.png*, etc. A noter que **RStudio** permet également de modifier la taille de sauvegarde de la fenêtre.

Cependant, il peut parfois être utile d'utiliser directement les commandes spécifiques à **R** pour sauvegarder le graphique. Le fonctionnement est le suivant, On appelle d'abord une des fonctions `pdf()`, `bmp()`, `jpeg()`, `png()`, etc. selon le format désiré et on précise d'abord le nom du fichier à sauvegarder et éventuellement les dimensions du fichier. On insère ensuite les commandes graphiques (le graphique ne sera pas affiché dans **R**, car il est en cours d'enregistrement dans le fichier de sortie) et enfin, on finit obligatoirement par la commande `dev.off()` qui indique que l'opération est terminée.

**Exemple** : dans cet exemple, le graphique sera sauvegardé dans le répertoire de travail courant.

```
pdf(file = "figure1.pdf", width = 7, height = 6)
op <- par(mfrow = c(2, 2), oma = c(1, 1, 3, 0), mar = c(2, 3, 3, 1),
          mgp = c(2, 1, 0))
with(diam_ech, {
  hist(price)
  hist(carat)
  hist(x)
  hist(y)
})
mtext("Référence : données diamonds",
      side = 1, line = 0, cex = 0.5, col = "blue", outer = TRUE)
mtext("Quelques histogrammes",
      side = 3, line = 1, cex = 2, col = "blue", outer = TRUE)
par(op)
title("Quelques histogrammes")
dev.off()
```

# 2 Graphiques pour la statistique descriptive

On a vu dans la section précédente la "logique" de programmation utilisée pour représenter des graphiques sous **R**. On va s'intéresser dans cette section plus particulièrement aux fonctions graphiques disponibles sous **R** utiles pour l'analyse statistique descriptive.

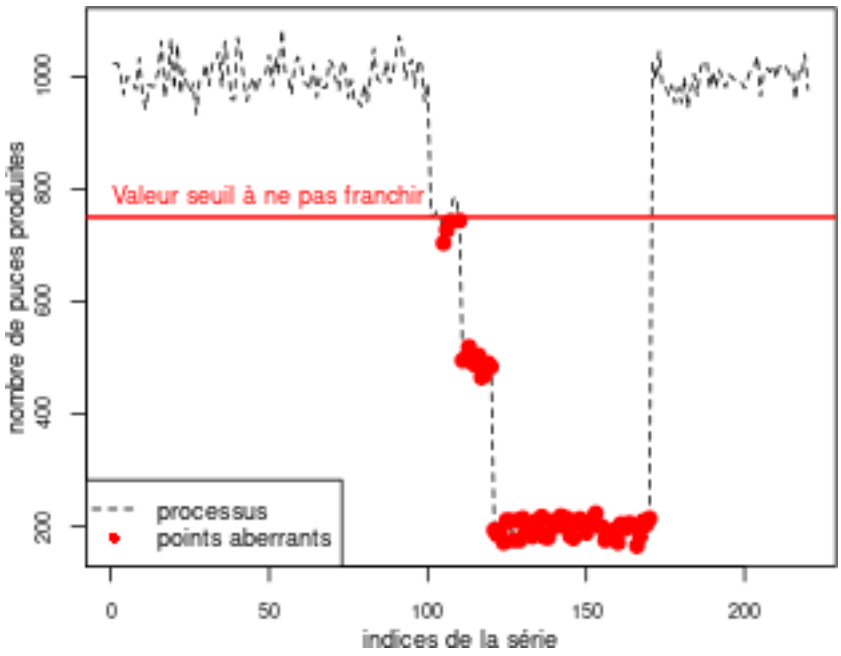
## 2.1 Analyse unidimensionnelle

### 2.1.1 Représentation d'un processus discret

La fonction `plot()` appliquée à une seule variable quantitative `x` de taille $n$ (un objet de type `numeric` ou `integer`) renvoie le graphique des valeurs de `x` par rapport à leurs indices dans le vecteur (de 1 à $n$). Ce graphique n'a pas trop d'intérêt si les observations sont indépendantes et identiquement distribuées (i.i.d). En revanche, lorsqu'il s'agit de données issues d'un processus discret ou bien d'une série temporelle, ce type de représentation peut être intéressant, car il permet de visualiser l'évolution de la série; par exemple, cela peut permettre d'observer un changement de comportement.

**Exemple** : dans une usine qui produit des puces électroniques, on suit par heure le nombre de puces produites. Lorsque tout se passe bien, les machines ont été calibrées pour produire environ 1000 puces par heure. Lorsqu'il y a une machine défectueuse, cela entraîne une baisse de la production. Pour repérer s'il y a un problème dans la production, l'usine s'est fixée comme valeur seuil à ne pas franchir, la valeur 750. Nous avons simulé une série statistique supposant reproduire un tel phénomène. Les 100 premières valeurs sont simulées selon une loi de Poisson $\mathcal{P}(1000)$, puis nous avons simulé un incident telle que la production diminue progressivement jusqu'à atteindre une loi de Poisson $\mathcal{P}(200)$. Une fois le problème réparé, la série repart sur une loi de Poisson $\mathcal{P}(1000)$.

```
# simulation de la série
x <- c(rpois(100, 1000), rpois(10, 750), rpois(10, 500),
       rpois(50, 200), rpois(50, 1000))
# représentation de la série (fonction haut-niveau)
op <- par(mar = c(3, 3, 1, 1),
          mgp = c(2, 1, 0),
          cex.axis = 0.8)
plot(x, xlab = "indices de la série",
     ylab = "nombre de puces produites",
     type = "l",  # option qui indique de relier les points entre eux par une ligne
     lty = 2)     # option qui indique quel type de traits utilisés
# Valeur seuil (fonction bas-niveau abline())
abline(h = 750, lwd = 2, col = "red")
# Commentaire (fonction bas-niveau text())
text(50, 750, "Valeur seuil à ne pas franchir", pos = 3,
     col = "red")
# Valeurs à problèmes (fonction bas-niveau points())
ind <- 1:length(x)
points(ind[x < 750], x[x < 750], pch = 20,   # la fonction points(x, y) permet d'ajouter
       cex = 2, col = "red")                 # des nouveaux points au graphique
# Ajout d'une légende (fonction legend())
legend("bottomleft", legend = c("processus", "points aberrants"),
       lty = c(2, NA), pch = c(NA, 16), col = c("black", "red"))
```




```
par(op)
```

Dans l'exemple ci-dessus, nous avons utilisé une nouvelle fonction de bas-niveau. Il s'agit de la fonction `legend()`. Pour définir l'emplacement de la boîte de légende, on utilise soit des coordonnées `x` et `y` ainsi `legend(x, y, ...)`, soit on utilise un de ces 4 `character` comme premier argument de la fonction : `"topleft"`, `"topright"`, `"bottomleft"`, `"bottomright"` qui placera la boîte dans le coin "haut à gauche", "haut à droite", "bas à gauche", "bas à droite". Ensuite, l'argument `legend=` est un vecteur de `character` contenant les légendes à écrire. Enfin, pour chacune de ces légendes, on décrit s'il s'agit d'une ligne (`lty=`), d'un point (`pch=`), d'une couleur (`col=`).

### 2.1.2 Représentation d'une série temporelle

Si les indices du processus discret sont remplacés par des dates, alors on parlera de série temporelle. Il existe plusieurs façons de représenter une telle série dont nous allons présenter ici seulement le chronogramme. Pour plus de détails sur l'étude des séries temporelles, on réfèrera le lecteur à l'ouvrage d'Yves Aragon paru en 2011 (2ème édition en 2016), "Séries temporelles avec **R**" (site web du livre).

**Exemple** : on a repris ici le tout premier exemple d'Aragon (2011) qui représente l'évolution de la population française et aux Etats-Unis. On commence par charger le package associé au livre ainsi que les données :

```
# chargement du package associé au livre
require("caschrono")
# chargement des données de population
data("popfr")
```

L'objet `popfr` est de classe `ts` qui contient la série des observations ainsi que les dates correspondantes. Pour obtenir ces dates, on utilise la fonction `time()`.

```
class(popfr)
```

```
## [1] "ts"
```

```
time(popfr)
```

```
## Time Series:
## Start = 1846
## End = 1951
```

```
## Frequency = 0.2
##  [1] 1846 1851 1856 1861 1866 1871 1876 1881 1886 1891 1896 1901 1906 1911 1916
## [16] 1921 1926 1931 1936 1941 1946 1951
```

Finalement, pour représenter le graphique d'un tel objet, l'idée est bien entendu de représenter les valeurs de la série en ordonnées et les dates correspondantes en abscisses. Pour cela, on va utiliser une fonction générique de la fonction `plot()`. Une **fonction générique** est en quelque sorte une extension d'une fonction connue afin d'utiliser le même nom de fonction, mais qui s'applique sur de nouvelles classes d'objets (cette notions sera vue en détails dans le cours de **R** avancé). Aussi, la fonction `plot()` vue précédémment peut s'appliquer sur différents types d'objet (dont la classe `ts`). Pour connaître les fonctions génériques de la fonction `plot()`, on utilise la fonction `methods()` :

```
methods(plot)
```

```
##  [1] plot,ANY-method                plot,color-method
##  [3] plot.aareg*                    plot.acf*
##  [5] plot.agnes*                    plot.areg*
##  [7] plot.areg.boot*                plot.aregImpute*
##  [9] plot.biVar*                    plot.clusGap*
## [11] plot.cox.zph*                  plot.curveRep*
## [13] plot.data.frame*               plot.decomposed.ts*
## [15] plot.default                   plot.dendrogram*
## [17] plot.density*                  plot.describe*
## [19] plot.diana*                    plot.drawPlot*
## [21] plot.ecdf                      plot.factor*
## [23] plot.formula*                  plot.function
## [25] plot.gbayes*                   plot.ggplot*
## [27] plot.gtable*                   plot.hcl_palettes*
## [29] plot.hclust*                   plot.histogram*
## [31] plot.HoltWinters*              plot.isoreg*
## [33] plot.lm*                       plot.magick-image*
## [35] plot.medpolish*                plot.mlm*
## [37] plot.mona*                     plot.partition*
## [39] plot.ppr*                      plot.prcomp*
## [41] plot.princomp*                 plot.profile.nls*
## [43] plot.Quantile2*                plot.R6*
## [45] plot.raster*                   plot.rm.boot*
## [47] plot.rpart*                    plot.shingle*
## [49] plot.silhouette*               plot.spec*
## [51] plot.spline*                   plot.stepfun
## [53] plot.stl*                      plot.summary.formula.response*
## [55] plot.summary.formula.reverse*  plot.summaryM*
## [57] plot.summaryP*                 plot.summaryS*
## [59] plot.Surv*                     plot.survfit*
## [61] plot.table*                    plot.trans*
## [63] plot.transcan*                 plot.trellis*
## [65] plot.ts                        plot.tskernel*
## [67] plot.TukeyHSD*                 plot.varclus*
## [69] plot.xyVector*                 plot.zoo
## see '?methods' for accessing help and source code
```

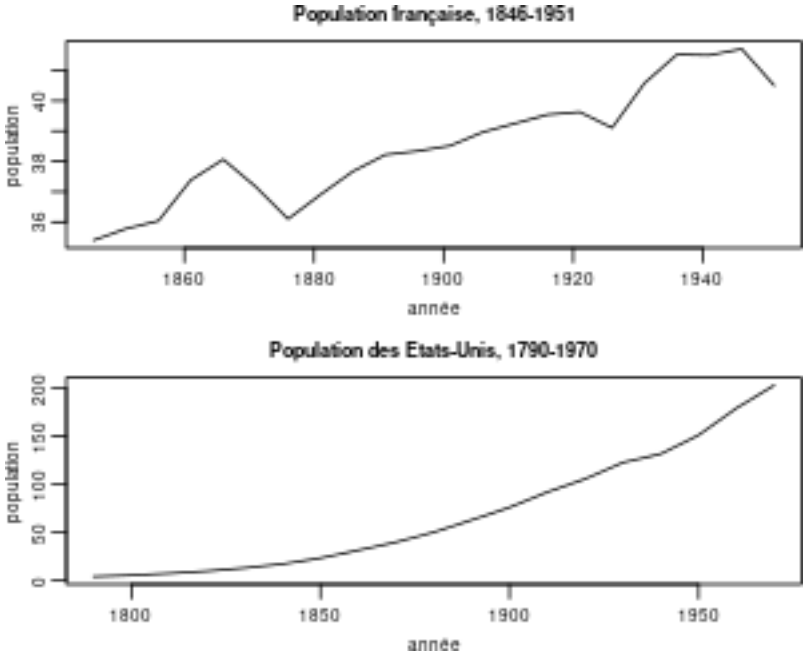
**Remarque 1** : les fonctions suivies d'un astérisque sont des fonctions non visibles, c'est-à-dire que leurs codes ne s'affichent pas directement lorsqu'on tape leur nom dans la console. Pour afficher le code de ces fonctions, il faut utiliser la fonction `getAnywhere()`. Par exemple :

```
getAnywhere(plot.acf)
```

**Remarque 2** : on constate que le nombre de fonctions génériques de la fonction `plot()` est très important. Pour les appeler, on peut soit appeler la fonction par son nom complet (`plot.ts()` par exemple) ou simplement par la commande `plot()`.

Ici, on représente les deux séries temporelles dans la même fenêtre graphique en utilisant les commandes vues dans la partie précédente.

```
# paramètres graphiques
op <- par(mfrow = c(2, 1), mar = c(2.5, 2.5, 2, 2),
          mgp = c(1.5, .5, 0), oma = c(0, 0, 0, 0),
          cex.main = .8, cex.lab = .7, cex.axis = .7)
# appel de la fonction générique plot.ts
plot.ts(popfr, xlab = "année", ylab = "population",
        main = "Population française, 1846-1951")
plot(uspop, xlab = "année", ylab = "population",
     main = "Population des Etats-Unis, 1790-1970")
```



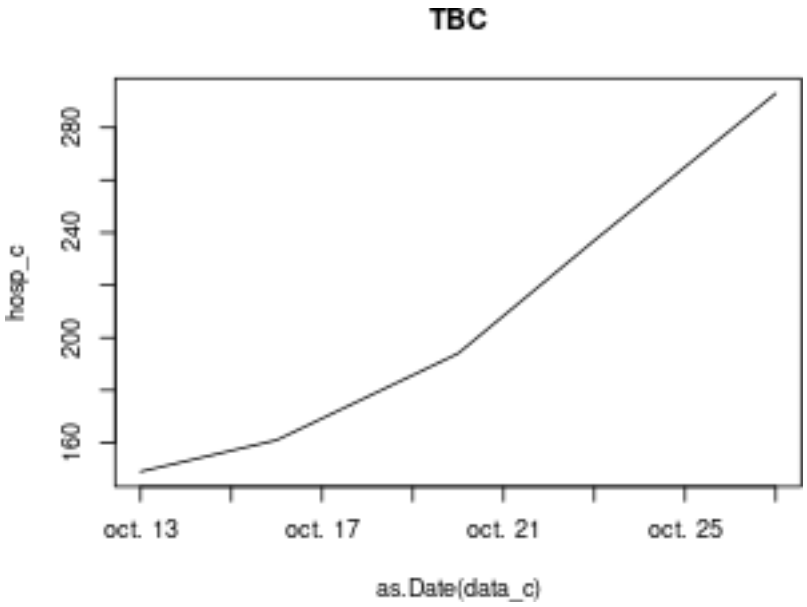

```
par(op)
```

**Remarque :** utiliser la fonction `plot.ts()` revient à utiliser la fonction `plot()` de base auquelle des arguments ont été remplis par défaut. Par exemple, ce n'est pas la peine d'écire l'option `type="l"` car ce paramètre a été défini à l'intérieur de la fonction `plot.ts()`.

**Exercice 4.3**

- Représenter une courbe de série temporelle liée à des données de Covid 19 que vous aurez importer. L'idéal sera d'importer le fichier de données depuis un site que vous aurez choisi (par exemple https://www.data.gouv.fr/fr/datasets/donnees-hospitalieres-relatives-a-lepidemie-de-covid-19/), mais vous pouvez aussi copier/coller vos données dans la console. Par exemple:

```
data_c <- c("2020-10-27", "2020-10-23", "2020-10-20", "2020-10-16", "2020-10-13")
hosp_c <- c(293, 237, 194, 161, 149)
plot(as.Date(data_c), hosp_c, type = "l")
title("TBC")
```




### 2.1.3 Représentation de lois de distribution "théoriques"

En général, on scinde les lois de distributions en deux familles selon la nature de la variable $X$ :

- lorsque $X$ est discrète, c'est-à-dire qu'elle prend ses valeurs parmi un nombre fini ou dénombrable de valeurs, les distributions les plus connues sont : la loi binomiale (de paramètres $n$ et $p$) et la loi de Poisson (de paramètre $\lambda$).
- lorsque $X$ est continue, c'est-à-dire que ses valeurs possibles sont dans $\mathbb{R}$, les distributions les plus connues sont : la loi de Laplace/Gauss dite aussi loi normale (de paramètres $\mu$ et $\sigma$), la loi de Fisher (de paramètres $\nu_1$ et $\nu_2$) et la loi de Student (de paramètre $k$).

Pour caractériser une loi de distribution, on utilise plus particulièrement deux outils qui peuvent être représentés graphiquement :

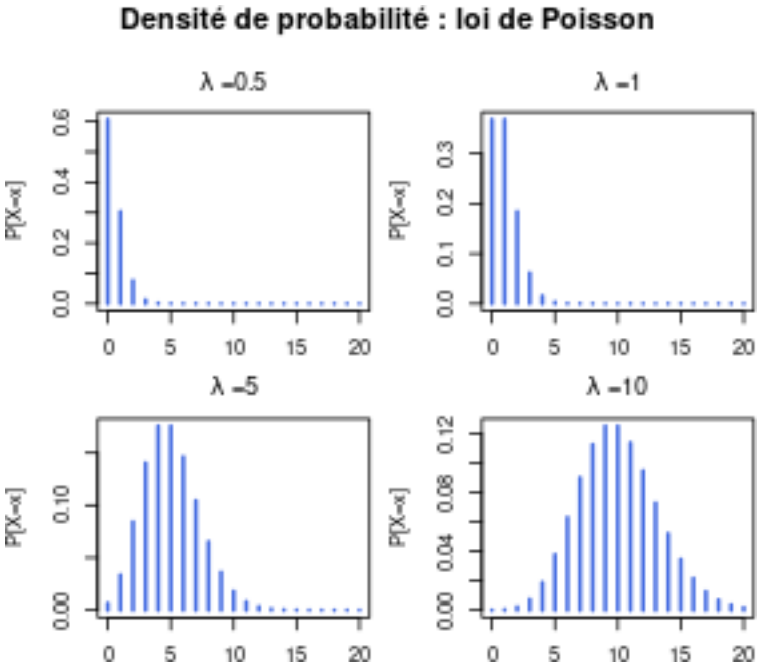
- la densité de probabilité (usuellement notée $f$). Dans **R**, les fonctions qui permettent de calculer ces fonctions commencent par la lettre `d` suivi de l'abbréviation de la loi. Par exemple, `dpois()`, `dbinom()`, `dnorm()`, etc.
- la fonction de répartition (usuellement notée $F$). Dans **R**, les fonctions qui permettent de calculer ces fonctions commencent par la lettre `p` suivi de l'abbréviation de la loi. Par exemple, `ppois()`, `pbinom()`, `pnorm()`, etc.

Selon la nature de la variable $X$, ces outils ne seront pas représentés de la même façon (voir exemples ci-dessous). Pous plus d'informations, on se reportera à l'ouvrage de Saporta (1990) "Probabilités, Analyse des données et Statistique".

**Exemple 1** : représentation de la densité de probabilités de la loi de Poisson pour différentes valeurs de $\lambda$. La fonction `dpois(x, lambda)` renvoie la probabilité d'obtenir la valeur entière `x` lorsque $X \sim \mathcal{P}(\lambda)$. Comme $X$ ne prend que des valeurs discrètes, on représente les probabilités d'obtenir une valeur $x$ par un trait vertical (option `type="h"` de la fonction `plot()`).

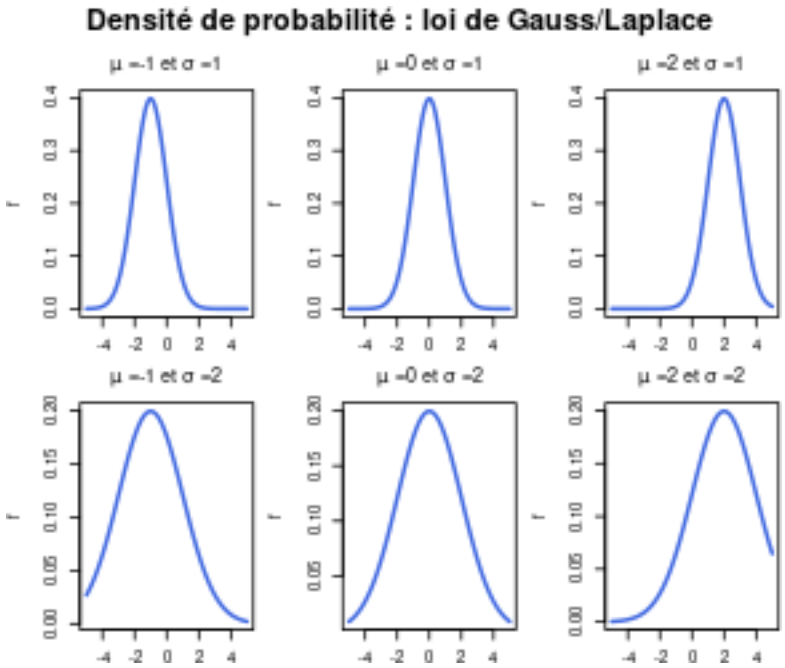
```
x <- seq(0, 20, 1)
op <- par(mfrow = c(2, 2), oma = c(0.5, 2, 2, 2),
          mar = c(1.8, 4, 2.7, 0.7))
for (lambda in c(0.5, 1, 5, 10)) {
  plot(x, dpois(x, lambda), type = "h", lwd = 2,
       ylab = "P[X=x]", col="royalblue",
```

```
       main = bquote(lambda ~ paste("=", .(lambda))))
}
par(op)
title("Densité de probabilité : loi de Poisson", line = -1,
      outer = TRUE)
```



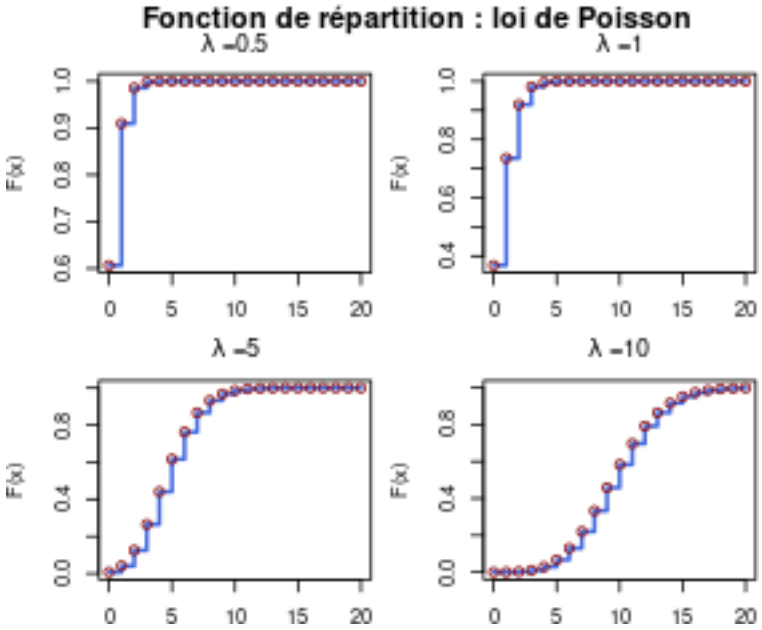

**Exemple 2** : représentation de la densité de probabilités de la loi de Gauss/Laplace pour différentes valeurs de $(\mu, \sigma)$. La fonction `dnorm(x, mean = , sd = )` renvoie la valeur de la fonction de densité $f_X$ au niveau de la valeur `x` lorsque $X \sim \mathcal{N}(\mu, \sigma^2)$. Comme $X$ est continue, $f_X$ existe pour tout $x \in \mathbb{R}$. On représente ici cette fonction pour différentes valeurs de $\mu$ et $\sigma$ avec un trait continu (option `type="l"` de la fonction `plot()`)

```
x <- seq(-5, 5, 0.1)
op <- par(mfrow = c(2, 3), oma = c(0.5, 2, 2, 2),
          mar = c(1.8, 4, 2.7, 0.7))
for (sigma in c(1, 2)) {
  for (mu in c(-1, 0, 2)) {
    plot(x, dnorm(x, mu, sigma), type = "l", lwd = 2,
         ylab = "f", col = "royalblue",
         main = bquote(mu~paste("=",.(mu)," et")~sigma~paste("=",.(sigma))))
  }
}
par(op)
title("Densité de probabilité : loi de Gauss/Laplace",
      line = -1, outer = TRUE)
```



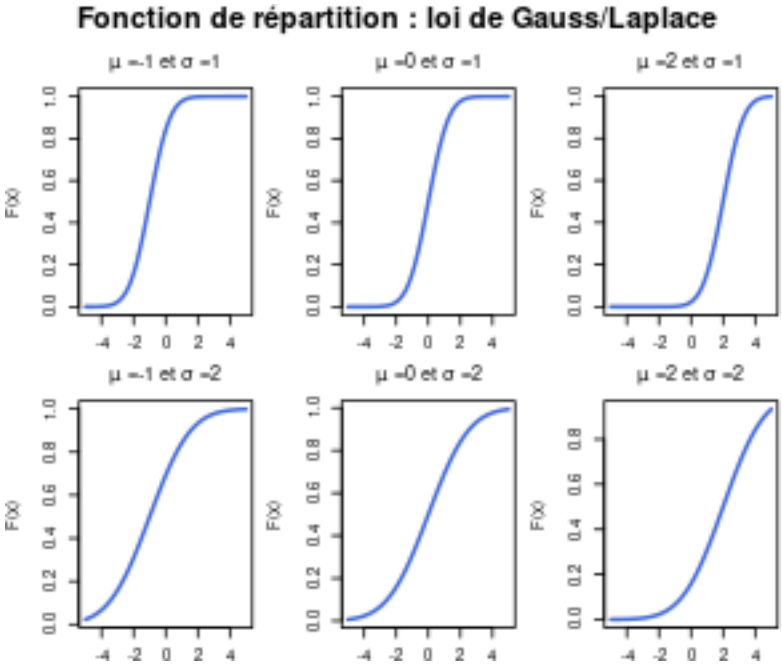

**Exemple 3** : représentation de la fonction de répartition de la loi de Poisson pour différentes valeurs de $\lambda$. La fonction `ppois(q, lambda)` renvoie la probabilité d'obtenir une valeur inférieure ou égale à la valeur entière `x` lorsque $X \sim \mathcal{P}(\lambda)$. Comme $X$ est discrète, la forme de la fonction de répartition est en escalier (option `type="s"` de la fonction `plot()`) :

```
x <- seq(0, 20, 1)
op <- par(mfrow = c(2, 2), oma = c(0.5, 2, 2, 2),
          mar = c(1.8, 4, 2.7, 0.7))
for (lambda in c(0.5, 1, 5, 10)) {
  plot(x, ppois(x, lambda), type = "s",
       main = bquote(lambda~paste("=",.(lambda))),
       pch = 16, lwd = 2, ylab = "F(x)", col = "royalblue")
  points(x, ppois(x, lambda), col = "dark red")
}
par(op)
title("Fonction de répartition : loi de Poisson", line = 1.75)
```




**Exemple 4** : représentation de la fonction de répartition de la loi de Laplace/Gauss pour différentes valeurs de $(\mu, \sigma)$. La fonction `pnorm(x, mean = , sd = )` renvoie la probabilité d'obtenir une valeur inférieure ou égale à la valeur `x` lorsque $X \sim \mathcal{N}(\mu, \sigma^2)$. Comme $X$ est continue, la courbe est également continue :

```
x <- seq(-5, 5, 0.1)
op <- par(mfrow = c(2, 3), oma = c(0.5, 2, 2, 2),
          mar = c(1.8, 4, 2.7, 0.7))
for (sigma in c(1, 2)) {
  for (mu in c(-1, 0, 2)) {
    plot(x, pnorm(x, mu, sigma), type = "l", lwd = 2,
         ylab = "F(x)", col = "royalblue",
         main = bquote(mu~paste("=",.(mu)," et")~sigma~paste("=",.(sigma))))
  }
}
par(op)
title("Fonction de répartition : loi de Gauss/Laplace",
       line = -1, outer = TRUE)
```




**Exercice 4.4**

Pour s'exercer sur les graphiques et réviser ses connaissances sur les lois de distribution, le lecteur pourra reprendre les exemples précédents et les appliquer sur d'autres lois de distribution parmi :

- loi beta (voir `?dbeta`), loi gamma (voir `?dgamma`) pour les lois continues
- loi hypergéométrique (voir `?dhyper`)

L'ensemble des lois connues sous **R** sont accessibles en faisant :

```
?Distributions
```

### 2.1.4 Représentation d'une variable quantitative discrète

Pour faire un résumé d'une variable quantitative discrète, on peut réaliser un tableau de fréquences. Reprenons le jeu de données `diam_ech`. La variable `table` correspond à la largeur relative du diamant. Il s'agit plutôt d'une variable continue, mais en ne prenant que la partie entière, on peut la considérer comme variable quantitative discrète. Pour cela, on ne va considérer que la partie entière de cette variable :

```
diam_ech$table <- round(diam_ech$table)
```

Une façon de savoir combien il y a de valeurs distintes pour cette variable est d'utiliser la fonction `unique()`, couplée avec la fonction `length()` :

```
length(unique(diam_ech$table))
```

```
## [1] 21
```

On constate donc que ce sont souvent les mêmes valeurs qui reviennent compte tenu que le jeu de données contient 53940 observations. Autrement dit, il existe un nombre fini de valeurs possibles et on peut donc résumer cette variable par un tableau de fréquences absolues et/ou relatives :

```
tab <- table(diam_ech$table)
```

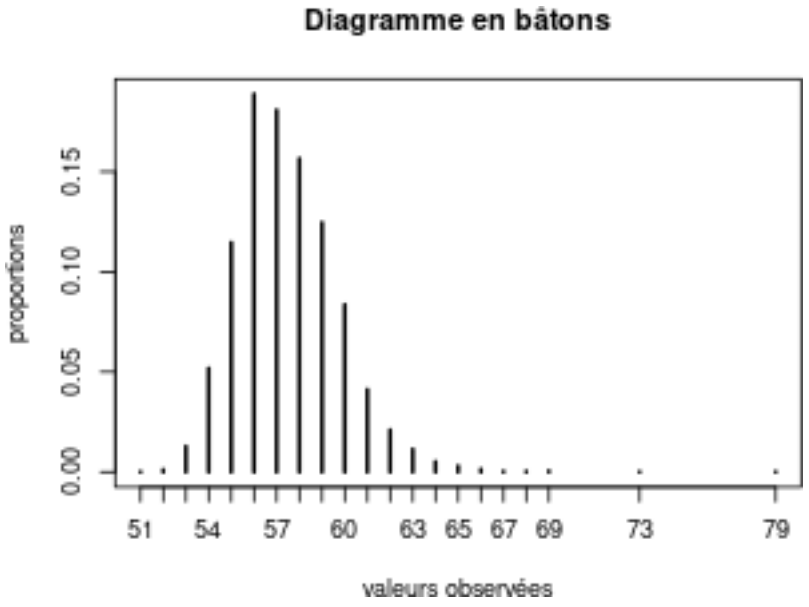
Pour intégrer ce tableau dans un document `R` Markdown, on eut utiliser la fonction `kable()` du package **knitr** et ajouter l'option `results = 'asis'` parmi les options du chunck:

```
knitr::kable(
    data.frame(valeurs = names(tab),
               effectifs = as.numeric(tab),
               proportion = as.numeric(prop.table(tab))))
```

| valeurs | effectifs | proportion |
|---|---|---|
| 51 | 1 | 0.0002 |
| 52 | 6 | 0.0012 |
| 53 | 64 | 0.0128 |
| 54 | 259 | 0.0518 |
| 55 | 573 | 0.1146 |
| 56 | 944 | 0.1888 |
| 57 | 904 | 0.1808 |
| 58 | 783 | 0.1566 |
| 59 | 623 | 0.1246 |
| 60 | 418 | 0.0836 |
| 61 | 206 | 0.0412 |
| 62 | 105 | 0.0210 |
| 63 | 57 | 0.0114 |
| 64 | 26 | 0.0052 |
| 65 | 15 | 0.0030 |
| 66 | 7 | 0.0014 |
| 67 | 2 | 0.0004 |
| 68 | 2 | 0.0004 |
| 69 | 3 | 0.0006 |
| 73 | 1 | 0.0002 |
| 79 | 1 | 0.0002 |

L'outil graphique qui permet de représenter un tel tableau est le diagramme en bâtons :

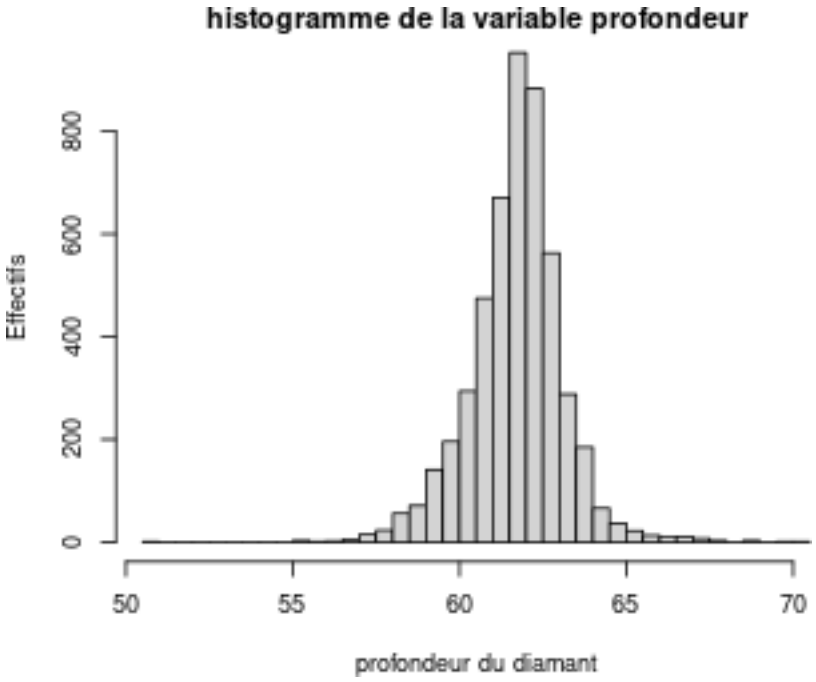
```
plot(prop.table(tab), xlab = "valeurs observées",
     ylab = "proportions")
title("Diagramme en bâtons")
```



### 2.1.5 Représentation d'une variable quantitative

**2.1.5.1 L'histogramme** En général, on utilise un histogramme pour représenter une variable quantitative $X$. Pour plus de détails sur l'histogramme, on renvoie le lecteur à ces quelques pages extraites du livre de Saporta [1990]. On reprendra les notations du livre de Saporta : on note $e_0, e_1, ..., e_k$ les bornes et l'on note pour chaque classe $[e_{i-1}, e_i[$ l'effectif $n_i$ (*frequency* ou *count* en anglais) et la fréquence $f_i$ (*relative frequency* ou *proportion* en anglais). Par défaut, la fonction `hist()` représente les $n_i$ en ordonnées car les classes sont d'amplitudes égales (c'est-à-dire que $(e_i - e_{i-1}) = constante$ quelque soit $i$). On notera que $e_0, e_1, ..., e_k$ sont calculées par défaut selon un algorithme particulier (voir `help(nclass)`). On peut aussi donner les valeurs de $e_0, e_1, ..., e_k$; pour cela, il faudrait utiliser l'option `breaks=` en précisant le vecteur des $e_0, e_1, ..., e_k$. On peut également utiliser l'option `nclass=` qui permet de donner le nombre de barres qu'l on souhaite (à noter qu'un algorithme est quand même appliqué pour définir un nombre de classes qui sera le plus proche possible de celui demandé). On applique ici la fonction `hist()` sur la variable "profondeur du diamant" en précisant qu'on souhaite un nombre de classes proche de 30.

```
op <- par(oma = c(0, 0, 0, 0), mar = c(4, 4, 1, 1))
res_hist <- hist(diam_ech$depth, nclass = 30,
                 xlab = "profondeur du diamant",
                 ylab = "Effectifs",
                 main = "histogramme de la variable profondeur")
```




```
par(op)
```

```
print(res_hist)
```

```
## $breaks
##  [1] 50.5 51.0 51.5 52.0 52.5 53.0 53.5 54.0 54.5 55.0 55.5 56.0 56.5 57.0 57.5
## [16] 58.0 58.5 59.0 59.5 60.0 60.5 61.0 61.5 62.0 62.5 63.0 63.5 64.0 64.5 65.0
## [31] 65.5 66.0 66.5 67.0 67.5 68.0 68.5 69.0 69.5 70.0 70.5
##
## $counts
##  [1]   1   0   0   0   0   0   0   0   0   3   1   2   5  15  23  56  71 141 196
## [20] 294 475 671 953 883 563 288 185  66  37  22  13  10  11   7   3   0   3   0
## [39]   1   1
```

```
##
## $density
##  [1] 0.0004 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0012
## [11] 0.0004 0.0008 0.0020 0.0060 0.0092 0.0224 0.0284 0.0564 0.0784 0.1176
## [21] 0.1900 0.2684 0.3812 0.3532 0.2252 0.1152 0.0740 0.0264 0.0148 0.0088
## [31] 0.0052 0.0040 0.0044 0.0028 0.0012 0.0000 0.0012 0.0000 0.0004 0.0004
##
## $mids
##  [1] 50.75 51.25 51.75 52.25 52.75 53.25 53.75 54.25 54.75 55.25 55.75 56.25
## [13] 56.75 57.25 57.75 58.25 58.75 59.25 59.75 60.25 60.75 61.25 61.75 62.25
## [25] 62.75 63.25 63.75 64.25 64.75 65.25 65.75 66.25 66.75 67.25 67.75 68.25
## [37] 68.75 69.25 69.75 70.25
##
## $xname
## [1] "diam_ech$depth"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
```

On constate que la fonction `hist()` retourne un objet de type `list` qui contient un certain nombre d'informations dont :

- La valeur des $e_0, e_1, ..., e_k$ :

```
res_hist$breaks
```

```
##  [1] 50.5 51.0 51.5 52.0 52.5 53.0 53.5 54.0 54.5 55.0 55.5 56.0 56.5 57.0 57.5
## [16] 58.0 58.5 59.0 59.5 60.0 60.5 61.0 61.5 62.0 62.5 63.0 63.5 64.0 64.5 65.0
## [31] 65.5 66.0 66.5 67.0 67.5 68.0 68.5 69.0 69.5 70.0 70.5
```

- La valeur des $n_i$ :

```
res_hist$counts
```

```
##  [1]   1   0   0   0   0   0   0   0   0   3   1   2   5  15  23  56  71 141 196
## [20] 294 475 671 953 883 563 288 185  66  37  22  13  10  11   7   3   0   3   0
## [39]   1   1
```

- La valeur des densités de probabiltés (c'est-à-dire $\frac{f_i}{e_i - e_{i-1}}$) :

```
res_hist$density
```

```
##  [1] 0.0004 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0012
## [11] 0.0004 0.0008 0.0020 0.0060 0.0092 0.0224 0.0284 0.0564 0.0784 0.1176
## [21] 0.1900 0.2684 0.3812 0.3532 0.2252 0.1152 0.0740 0.0264 0.0148 0.0088
## [31] 0.0052 0.0040 0.0044 0.0028 0.0012 0.0000 0.0012 0.0000 0.0004 0.0004
```

- La valeur des barycentres des classes $\frac{1}{2}(e_i + e_{i-1})$ :

```
res_hist$mids
```

```
##  [1] 50.75 51.25 51.75 52.25 52.75 53.25 53.75 54.25 54.75 55.25 55.75 56.25
## [13] 56.75 57.25 57.75 58.25 58.75 59.25 59.75 60.25 60.75 61.25 61.75 62.25
## [25] 62.75 63.25 63.75 64.25 64.75 65.25 65.75 66.25 66.75 67.25 67.75 68.25
## [37] 68.75 69.25 69.75 70.25
```

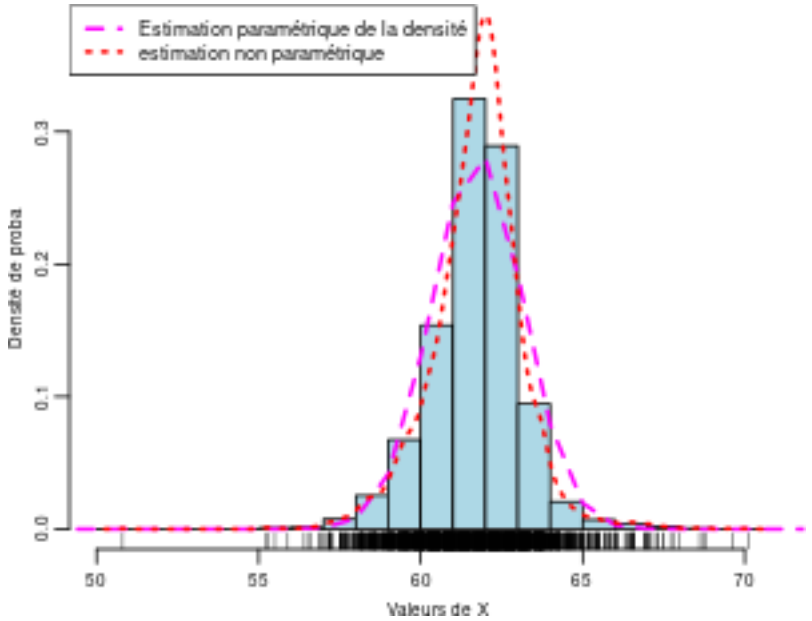
L'inconvénient de représenter les effectifs en ordonnées est qu'on ne peut pas représenter l'estimation d'une

densité par-dessus car les échelles ne correspondent pas. L'option `freq=FALSE` permet de représenter les densités de probabilités en ordonnées au lieu des effectifs (*frequency* en anglais).

**2.1.5.2 La fonction de densité** Dans certains cas, on a une connaissance *a priori* sur la distribution d'une variable quantitative. En d'autres termes, on sait que la variable $X$ est issue d'une distribution connue et on peut ainsi utiliser des outils de la "Statistique mathématique" pour estimer de façon adéquate le(s) paramètre(s) associé(s). Par exemple, si on suppose que la variable `VER` est issue d'une loi normale, alors les estimateurs les plus vrais semblants (maximum-likelihood estimation) pour $\mu$ et $\sigma^2$ sont respectivement $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ et $\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$.

**Exemple** : on va représenter par-dessus l'histogramme de la variable `depth`, la fonction de densité estimée en utilisant la fonction `dnorm()` vue précédemment.

```
op <- par(mar = c(2.5, 2.5, 2, 2),
          mgp = c(1.5, .5, 0), oma = c(0, 0, 0, 0),
          cex.main = .8, cex.lab = .7, cex.axis = .7)
hist(diam_ech$depth, freq = FALSE, col = "lightblue",
     xlab = "Valeurs de X", ylab = "Densité de proba",
     main = "", ylim = c(0, 0.38), nclass = 20)
x <- seq(0, 1500, 1)
lines(x, dnorm(x, mean(diam_ech$depth), sd(diam_ech$depth)),
     lty = 2, lwd = 2, col = "magenta")
lines(density(diam_ech$depth), col = "red", lty = 3, lwd = 2)
legend("topleft", lty = 2:3, lwd = 2,
       col = c("magenta", "red"), cex = 0.8,
       legend = c("Estimation paramétrique de la densité",
                  "estimation non paramétrique"))
rug(diam_ech$depth)
```

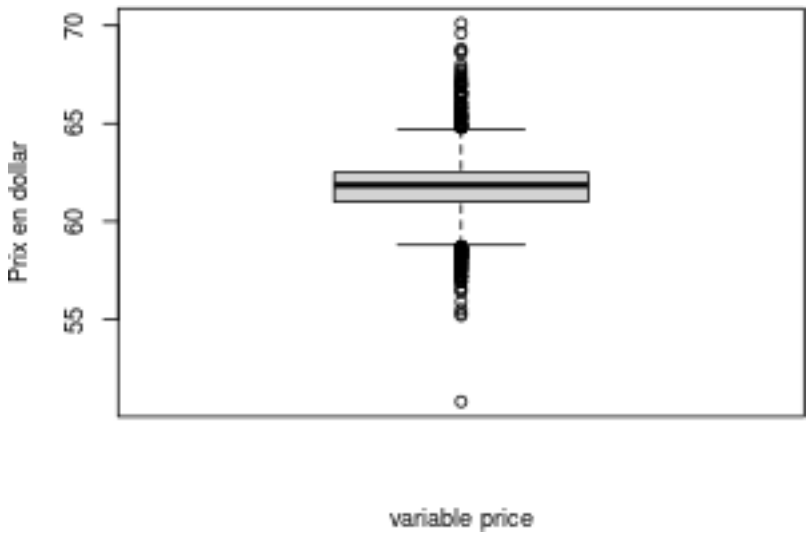



```
par(op)
```

**Remarque** : lorsqu'on ne sait rien de la loi de distribution théorique, on fait une estimation non paramétrique de la densité avec la fonction `density()`. Par ailleurs, dans le code-ci-dessus, la fonction `rug()` permet de représenter les valeurs des observations sur l'axe des abscisses par un trait vertical.

**2.1.5.3 La boîte à moustaches** L'autre outil généralement utilisé pour représenter une variable quantitative est la boîte à moustache (fonction `boxplot()`) qui résume quelques caractéristiques de position (médiane, quartiles, minimum, maximum). Cet outil permet notamment de repérer plus facilement les valeurs extrêmes. Les options `xlab=` et `ylab=` permettent de donner une légende respectivement aux axes des abscisses et des ordonnées. Voici un premier exemple simple :

```
b <- boxplot(diam_ech$depth, xlab = "variable price",
             ylab = "Prix en dollar")
```



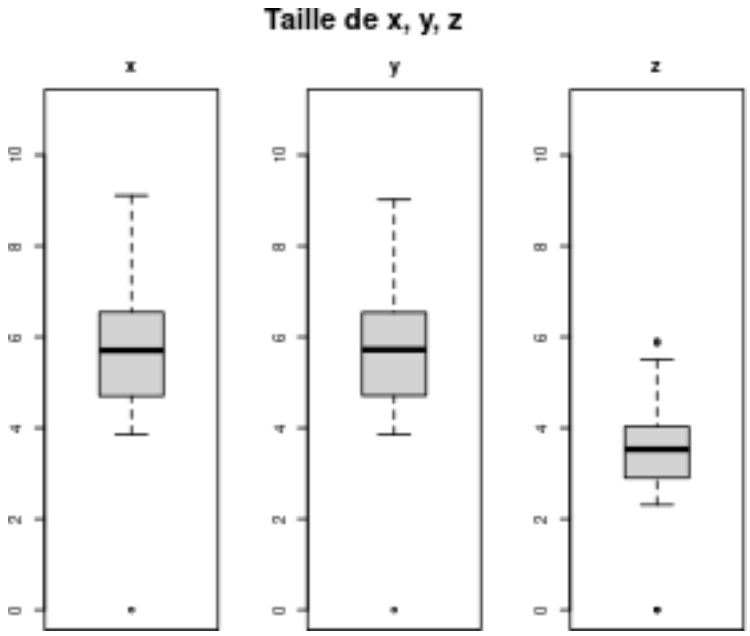

```
print(b)
```

L'objet `b` créé ci-dessous contient en autres les informations suivantes :

- `stats`=c($moustache_{inf}$, $Q_1$, $Q_2$, $Q_3$, $moustache_{sup}$) où $Q_1$, $Q_2$ et $Q_3$ sont les 1er, 2nd et 3ème quartiles, $moustache_{inf} \approx \max(Q_1 - 1.5(Q_3 - Q_1), \min_i x_i)$ et $moustache_{sup} \approx \min(Q_3 + 1.5(Q_3 - Q_1), \max_i x_i)$
- `out` contient les valeurs extrêmes, c'est-à-dire les valeurs qui sont au-dessus (respectivement en-dessous) de $moustache_{sup}$ (resp. $moustache_{inf}$).

**Exemple** : lorsque l'on a un nombre important de variables quantitatives dont l'étendue est plus ou moins la même (c'est le cas notamment des données biologiques issues des puces ADN), on peut représenter les boîtes à moustaches côte à côte pour comparer les distributions entre elles. Ici, on représente les variables correspondant à la taille, la largeur et la profondeur qui sont toutes exprimées en milimmètres et donc comparables.

```
op <- par(mfrow = c(1, 3), oma = c(0.5, 2, 2, 2),
       mar = c(1.8, 4, 2.7, 0.7))
with(diam_ech,{
       boxplot(x, ylim = c(0, 11), main = "x")
       boxplot(y, ylim = c(0, 11), main = "y")
       boxplot(z, ylim = c(0, 11), main = "z")
})
par(op)
title("Taille de x, y, z", line = -1, outer = TRUE)
```



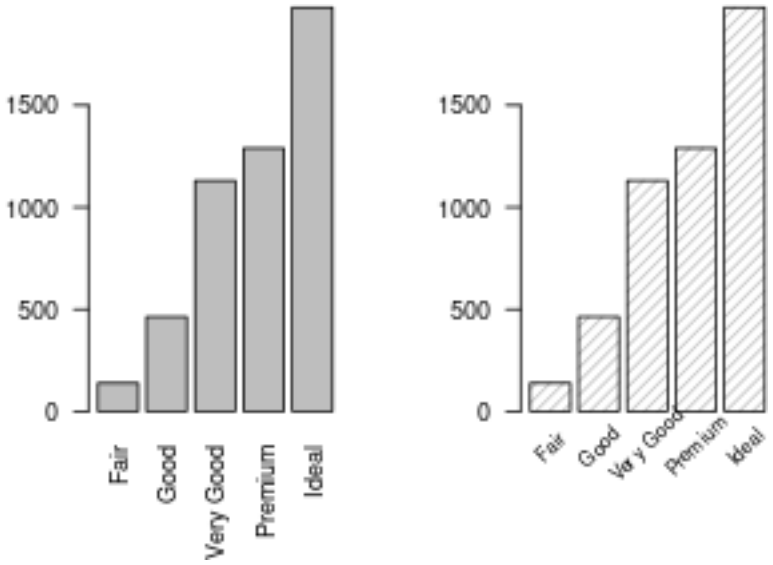

**Remarque** : cette représentation n'est pas à confondre avec les boîtes à moustaches parallèles que nous allons voir dans la section suivante et dont le principe est de croiser une variable quantitative avec une variable qualitative.

### 2.1.6 Représentation d'une variable qualitative

Les deux outils graphiques utilisés sont le diagramme en tuyaux d'orgues (*barplot* en anglais) et le camembert (*pie* en anglais).

**2.1.6.1 Le diagramme en tuyaux d'orgues** La fonction `plot()` appliquée à une variable qualitative (un objet de type `character` ou `factor`) renvoie automatiquement un diagramme en tuyaux d'orgues. En réalité, la fonction `plot()` appliquée à un objet `factor` fait appel à la fonction générique `plot.factor()` qui elle-même fait appel à la fonction `barplot()`. C'est donc dans l'aide de cette dernière qu'on trouvera les paramètres spécifiques à ce graphique (et aussi dans la fonction `par()` qui on le rappelle contient tous les paramètres graphiques communs à toutes les fonctions graphiques). La fonction `barplot()` s'utilise différemment de `plot.factor()` dans la mesure où `barplot` prend comme argument d'entrée un objet de type `table`, autrement dit, un tableau de fréquences obtenu avec la fonction `table()`. Par exemple :

```
op <- par(mfrow = c(1, 2))
plot(diam_ech$cut, cex.lab = 0.6, las = 2)
xtick <- barplot(table(diam_ech$cut),
        cex.lab = 0.6,
        las = 2, # permet de représenter les étiquettes /_ aux axes x et y
        density = 20, # hachure les barres
        xaxt = "n")   # ne représente pas les étiquettes sur l'axe des x
text(x = xtick,       # xtick contient l'abscisse des étiquettes
     par("usr")[3] - 150,
     labels = c("Fair", "Good", "Very Good", "Premium", "Ideal"),
     cex = 0.8,
     srt = 45,        # rotation des étiquettes
     xpd = T)         # permet d'écrire en-dehors de la fenêtre graphique
```



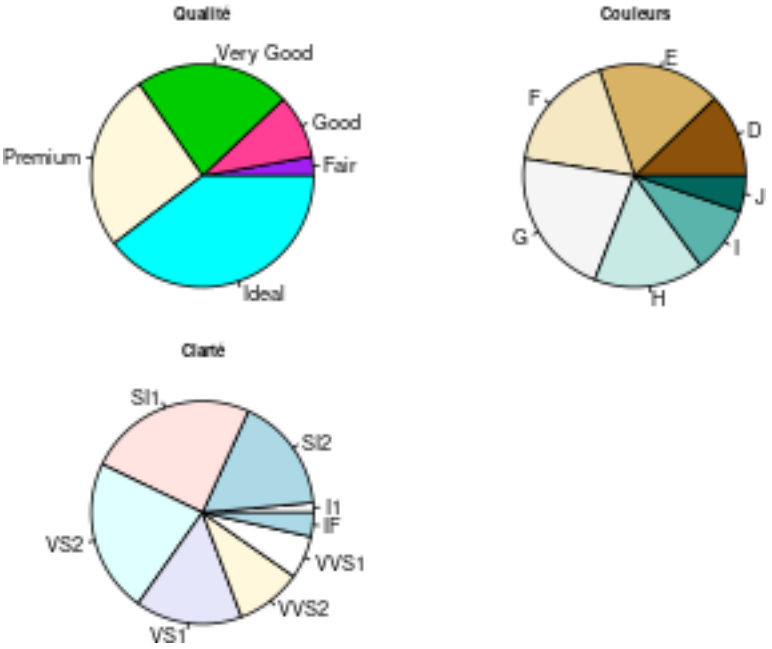

```
par(op)
```

**Remarque 1**: l'option `las=2` permet de représenter les légendes (abscisses et ordonnées) perpendiculairement à leurs axes. L'option `density=` permet d'hachurer les barres au lieu de les remplir.

**Remarque 2**: dans le cas où la variable est qualitative ordinale (c'est-à-dire qu'il y a une notion d'ordre entre les modalités), il peut être intéressant de trier les modalités pour représenter les modalités dans le bon ordre. Dans l'exemple ci-dessus, les modalités sont déjà triées.

**2.1.6.2 Le diagramme circulaire** La fonction `pie(x)` permet de représenter un diagramme circulaire où `x` contient un objet de type `table`, *i.e.* le tableau de fréquences (effectifs ou proportions) des modalités associées à la variable $X$.

```
op <- par(mfrow = c(2, 2), mar = c(0.5, 0.5, 1, 1),
          mgp = c(1.5, .5, 0),
          cex.main = .8, cex.lab = .7, cex.axis = .7)
with(diam_ech, {
       pie(table(cut), main = "Qualité",
           col = c("purple", "violetred1", "green3", "cornsilk", "cyan"))
       pie(table(color), main = "Couleurs", col = brewer.pal(7, "BrBG"))
       pie(table(clarity), main = "Clarté")
})
par(op)
```




## 2.2 Analyse bidimensionnelle

### 2.2.1 Croisement de deux variables qualitatives

La table de contingence permet de faire le résumé du lien entre deux variables qualitatives. Voici un exemple de table de contingence (variable `cut` croisée avec `color`). Nous verrons dans le chapitre suivant que c'est à partir de cette table qu'on sera en mesure ensuite de construire le test d'indépendence du $\chi^2$. Pour intégrer ce tableau dans notre document **R** Markdown, on fait de nouveau appel à la fonction `kable()` du package **knitr**:

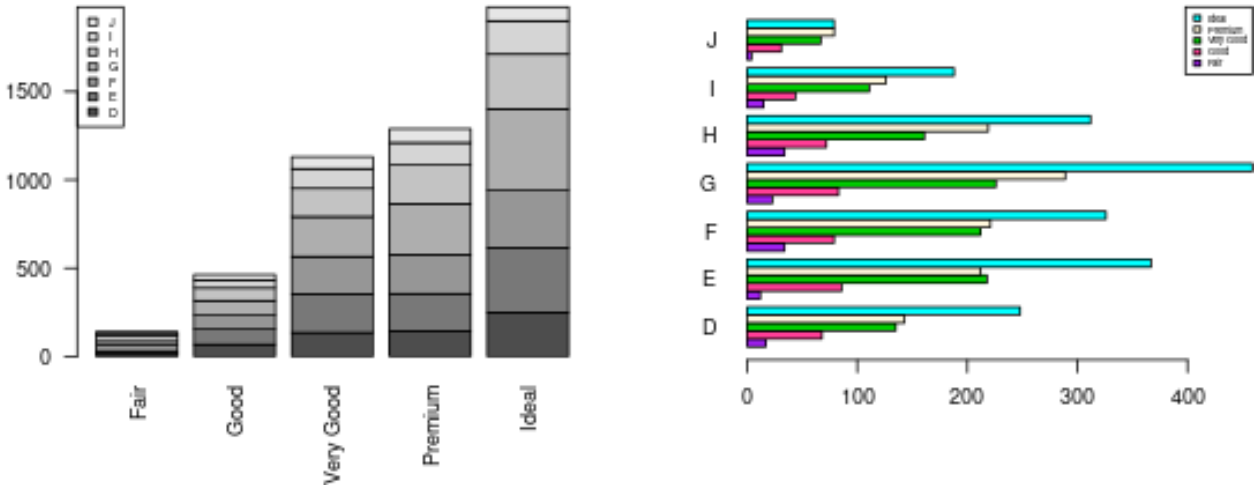
```
tab <- table(diam_ech$cut, diam_ech$color)
knitr::kable(tab)
```

| | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|
| Fair | 17 | 12 | 34 | 23 | 34 | 15 | 4 |
| Good | 68 | 86 | 79 | 83 | 72 | 44 | 31 |
| Very Good | 135 | 218 | 212 | 226 | 161 | 111 | 67 |
| Premium | 143 | 212 | 221 | 289 | 219 | 126 | 79 |
| Ideal | 248 | 367 | 326 | 459 | 312 | 188 | 79 |

Pour représenter cette table, on fera de nouveau appel à la fonction `barplot()` qui propose deux types de représentation selon l'option `beside=`.

```
op <- par(mfrow = c(1, 2))
barplot(t(tab), cex.lab = 0.6, las = 2,
        legend.text = TRUE, # indique qu'une légende doit être affichée
        args.legend = list(x = "topleft", cex = 0.6)) # list avec paramètres de légende
barplot(tab,
        beside = T, # plusieurs barres par modalités
        cex.lab = 0.6, las = 1,
        horiz = TRUE, # orientation horizontale
        legend.text = TRUE,
        col = c("purple", "violetred1", "green3", "cornsilk", "cyan"),
```

```
        args.legend = list(x = "topright", cex = 0.45))
```

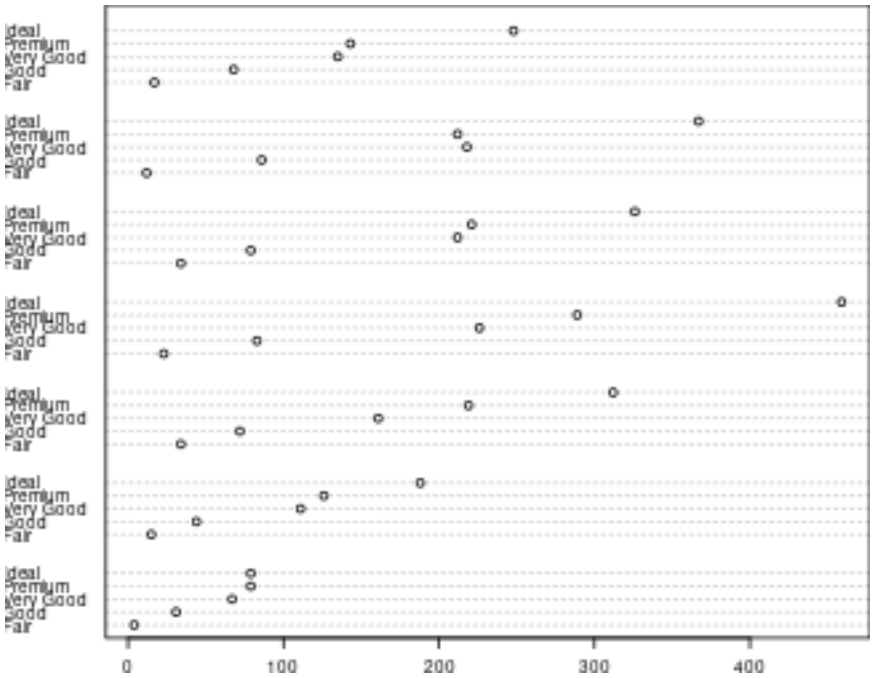



```
par(op)
```

**Remarque** : selon qu'on utilise `tab` ou `t(tab)` comme argument d'entrée, on permute l'emplacement des variables sur le graphique.

Il exite une alternative à la fonction `barplot()`. Il s'agit de la fonction `dotchart()` :

```
op <- par(mar = c(2, 5, 0, 0))
dotchart(tab, cex = 0.7)
```




```
par(op)
```

### 2.2.2 Croisement de deux variables quantitatives

La fonction `plot(x, y, ...)` appliquée à deux vecteurs `x` et `y` de type `numeric` retourne le nuage de points de la variable $Y$ en fonction de $X$. On a vu déjà vu dans la 1ère partie du cours certaines options disponibles :

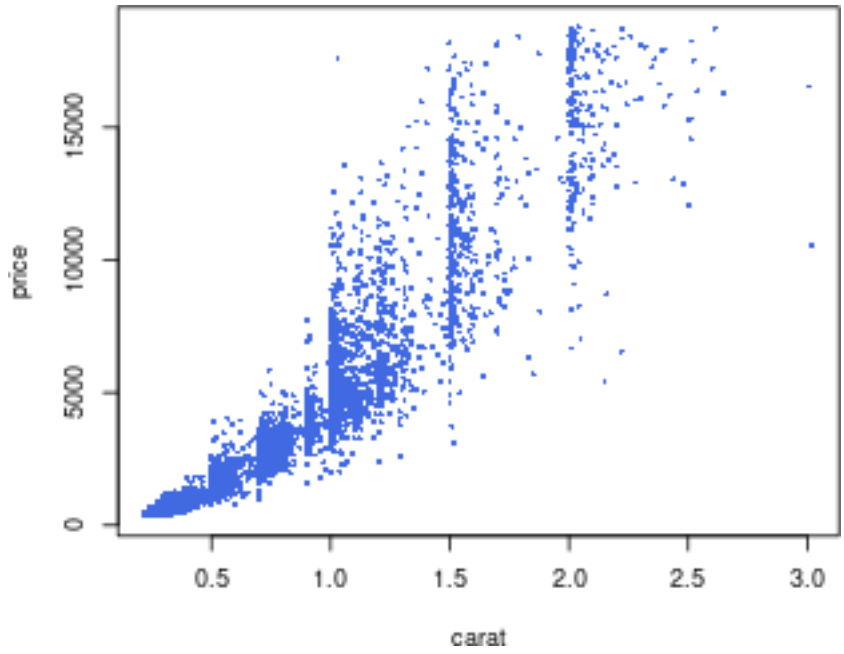
- `col` permet de modifier la couleur des points.
- `pch` (entier compris entre 1 et 25) permet de modifier le symbole des points.

Lorsque les variables que l'on souhaite représenter sont incluses dans un `data.frame`, il peut être intéressant d'utiliser la syntaxe suivante `plot(y ~ x, data = nom_data.frame, ...)`. En effet `y ~ x` est une syntaxe reconnue et utilisée dans **R** pour définir un modèle du type $Y$ est expliquée par $X$. On utilise cette forme (objet de type `formula`) pour tout type de régression.

**Exemple 1**

On va représenter le nuage de points de la variable `price` en fonction de `carat` :

```
op <- par(oma = c(0, 0, 0, 0), mar = c(4, 4, 1, 1))
plot(price ~ carat, data = diam_ech,
     col = "royalblue", pch = 16, cex = 0.5)
```




```
par(op)
```

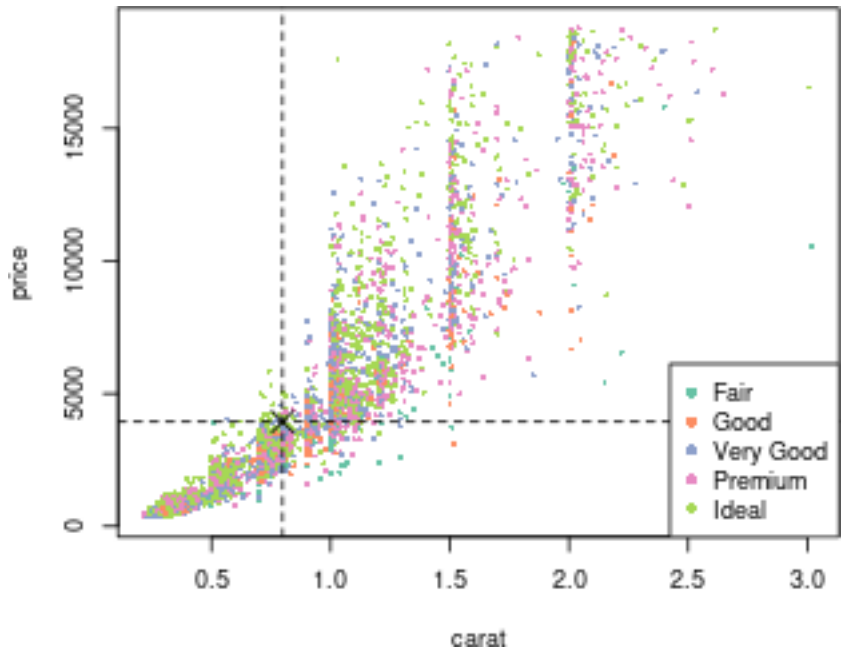
**Exemple 2**

Au graphique précédent, on va ajouter deux informations supplémentaires :

- on va représenter les points avec des couleurs différentes selon la qualité du diamant (on représentera ainsi l'information sur 3 variables en même temps : 2 variables quantitatives et une variable qualitative).
- on va ajouter 4 quadrants centrés autour du barycentre du nuage de point.

```
vec.col <- brewer.pal(5, "Set2")
op <- par(oma = c(0, 0, 0, 0), mar = c(4, 4, 1, 1))
plot(price ~ carat, data = diam_ech,
     col = vec.col[as.numeric(cut)],
     pch = 16,
     cex = 0.5)
with(diam_ech, {
       points(mean(carat), mean(price), pch = 4, cex = 2)
       abline(h = mean(price), lty = 2)
       abline(v = mean(carat), lty = 2)
       legend("bottomright", legend = levels(cut),
```

```
            col = vec.col, pch = 16)
})
```

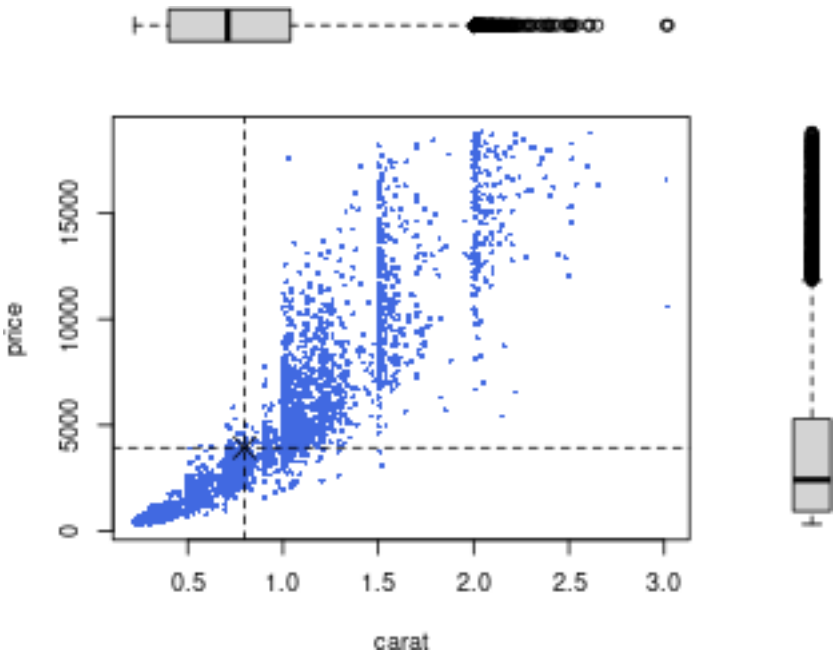



```
par(op)
```

**Remarque** : lorsqu'on fait `as.numeric()` sur une variable `factor`, on transforme les modalités de `factor` en une suite d'entier allant de 1 jusqu'à $K$ où $K$ est le nombre de modalités. En associant ce vecteur d'indices au vecteur des couleurs, ceci nous a permis d'attribuer à chaque modalité de `cut` une couleur.

**Exemple 3** : on va voir comment on peut ajouter au-dessus (respectivement à droite) du graphique les boîtes à moustaches de $X$ (resp. $Y$). Pour cela, on utilise une autre technique que celle vue dans la première partie du cours pour séparer en 3 parties distinctes la fenêtre graphique.

```
op <- par(fig = c(0, 0.8, 0, 0.8), mar = c(4, 4, 0, 0))
plot(price ~ carat, data = diam_ech, col = "royalblue", pch = 16, cex = 0.5)
with(diam_ech, {
     points(mean(carat), mean(price), pch = 4, cex = 2)
     abline(h = mean(price), lty = 2)
     abline(v = mean(carat), lty = 2)
     par(fig = c(0, 0.8, 0.7, 1), new = TRUE)
     boxplot(carat, horizontal = TRUE, axes = FALSE)
     par(fig = c(0.75, 1, 0, 0.8), new = TRUE)
     boxplot(price, axes = FALSE)
})
```




```
par(op)
```

**Remarque** : cette méthode consiste donc à appeler plusieurs fois la fonction `par()` en modifiant les paramètres `fig=c(x1, x2, y1, y2)` (donne les dimensions de la zone de la fenêtre sur laquelle on va travailler, par défaut : `fig=c(0, 1, 0, 1)` donne la fenêtre graphique entière) et `new=TRUE` (permet d'indiquer qu'on continue à travailler dans la fenêtre graphique en cours d'utilisation).

**Exemple 4** : on va voir comment on peut découper le jeu de données en 5 parties dépendant de la variable `cut` et représenter ensuite pour chaque modalité le nuage de points de `price` en fonction de `carat`. Pour découper l'échantillon en 5 partie, on va utiliser la fonction `split(x, f)` où `x` est un `data.frame` et `f` un factor qui va définir les groupes à découper :

```
diam_ech_split <- split(diam_ech, diam_ech$cut)
```

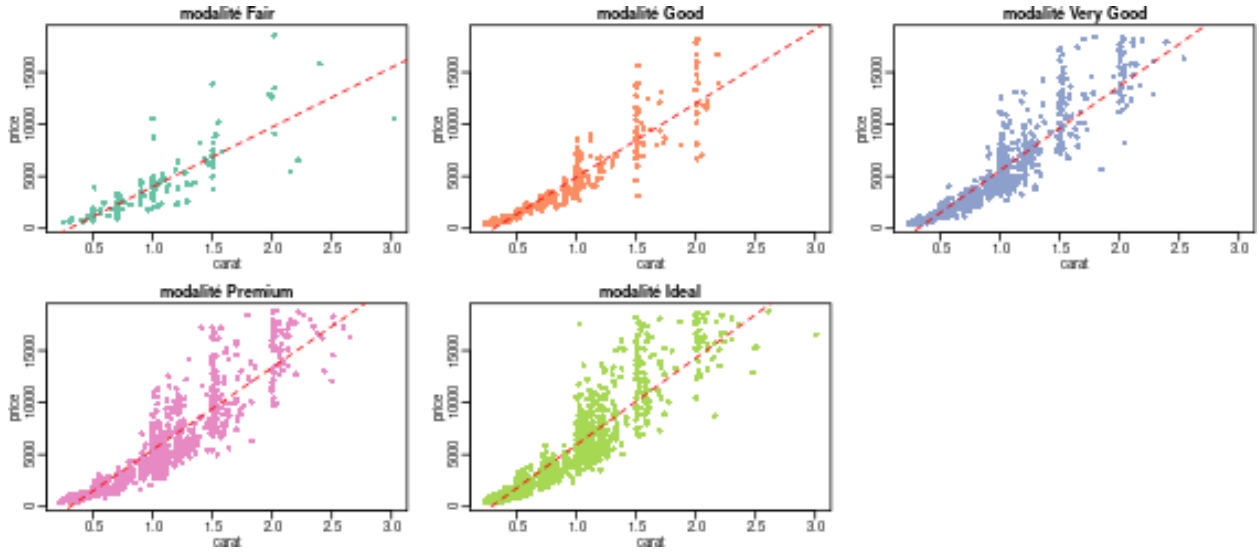
Pour représenter ensuite les 5 nuages de points, il va falloir utiliser une boucle et répéter pour chaque sous-échantillon la même sytaxe afin d'obtenir 5 graphiques ayant des propriétes similaires (échelle sur les axes, taille des points). On va changer uniquement le titre et la couleur des points selon la modalité de la varaible `cut`.

```
vec.col <- brewer.pal(5, "Set2")

op <- par(mfrow = c(2, 3),
          mar = c(3, 3, 1.5, 1),
          mgp = c(1.5, 0.5, 0)
          )

for(k in 1:length(diam_ech_split)) {
  plot(price ~ carat, data = diam_ech_split[[k]],
       xlim = range(diam_ech$carat),
       ylim = range(diam_ech$price),
       col = vec.col[k],
       pch = 16, cex = 1)
  title(paste("modalité", names(diam_ech_split)[k]))
  abline(lm(price ~ carat, data = diam_ech_split[[k]]),
         col = "red", lty = 2)
```

```
}
par(op)
```

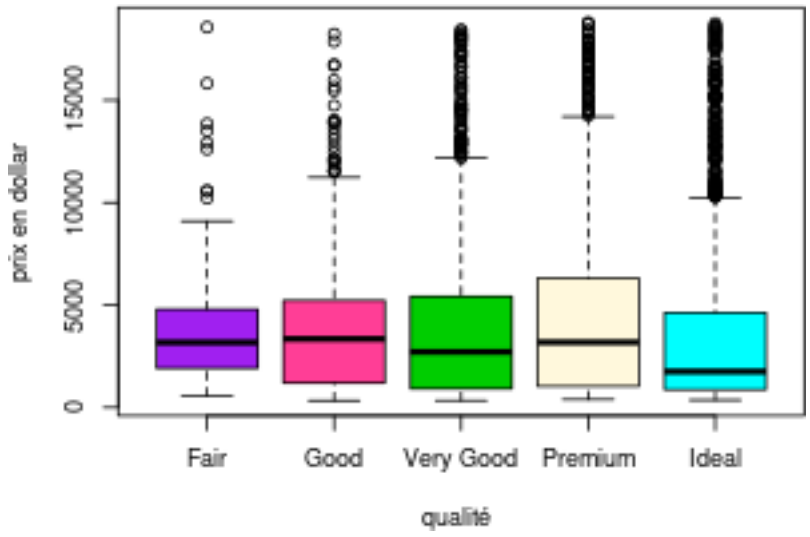



**Remarque:** ce type de représentation sera beaucoup plus facile à réaliser en utilisant la syntaxe **ggplot2**.

### 2.2.3 Croisement entre une variable quantitative et une variable qualitative

L'outil intéressant à utiliser est la boîte à moustache parallèle. Pour chaque modalité de la variable qualitative $X$, on représente la boîte à moustache de la variable quantitative $Y$. Pour réaliser ce graphique, on utilise la fonction `boxplot()` en précisant comme argument d'entrée une `formula`. Ici, on explique le prix du diamant en fonction de sa qualité :

```
vec.col <- c("purple", "violetred1", "green3", "cornsilk", "cyan")
boxplot(price ~ cut, data = diam_ech, col = vec.col,
        ylab = "prix en dollar", xlab = "qualité ")
```




**Exercice 4.5**

Choisir trois graphiques présentés ci-dessus et les appliquer sur la base de données `diam_ech` en utilisant les fonctions de base.

# 3 Résumé des principales fonctions et paramètres pour graphiques de base

On résume ici les principales fonctions graphiques vues jusqu'à présent ainsi que leurs paramètres principaux.

## 3.1 `par()`

| Arguments | Objectifs |
|---|---|
| `mar` | un vecteur `numeric` avec 4 valeurs; définit les marges d'un graphique (en nombre de lignes) |
| `mai` | un vecteur `numeric` avec 4 valeurs; définit les marges d'un graphique (en inches) |
| `oma` | un vecteur `numeric` avec 4 valeurs; définit les marges de la figure (en nombre de lignes) |
| `omi` | un vecteur `numeric` avec 4 valeurs; définit les marges de la figure (en inches) |
| `mgp` | un vecteur `numeric` avec 3 valeurs; définit le numéro de ligne pour titre/annotation/graduation |
| `bg` | une couleur pour définir la couleur de fond de la figure |
| `fg` | une couleur pour définir la couleur du cadre + graduations + points/lignes dans la figure |
| `pch` | un entier qui définit le style des points |
| `lty` | un entier qui définit le style des traits |
| `lwd` | un `numeric` qui définit l'épaisseur des traits |
| `col` | une couleur qui définit la couleur du cadre et des points/lignes dans la figure |
| `col.lab` | une couleur qui définit la couleur des légendes des axes x et y |
| `col.axis` | une couleur qui définit la couleur des annotations des axes x et y |
| `col.main` | une couleur qui définit la couleur du titre |
| `cex` | un `numeric` qui définit la taille des points |
| `cex.lab` | un `numeric` qui définit la taille des légendes |
| `cex.axis` | un `numeric` qui définit la taille des annotations |
| `cex.main` | un `numeric` qui définit la taille du titre |
| `mfrow` | un vecteur de deux entiers (ex : `c(2, 3)`) : division de la figure en 2 lignes et 3 colonnes |
| `las` | un entier (1 pour annotations // aux axes, 2 pour annotations horizontales, 3 pour perpendiculaires aux axes) |
| `bty` | forme du cadre; `"n"` pour enlever le cadre |

**NB :** la plupart des arguments ci-dessus pourront s'appliquer directement sur les fonctions de haut-niveau.

## 3.2 Fonctions de haut-niveau

| Fonctions | Arguments | Objectifs |
|---|---|---|
| `plot()` | `x` | un vecteur qui indique les coordonnées des abscisses |
| | `y` | (optionnel) un vecteur qui indique les coordonnées des ordonnées |
| | `xlab` | un `character` qui indique le titre de l'axe x |
| | `ylab` | un `character` qui indique le titre de l'axe y |
| | `ann` | un booléen pour représenter oui/non les légendes des x/y |
| | `xlim` | un vecteur de deux `numeric` qui indique les bornes de l'axe x |
| | `ylim` | un vecteur de deux `numeric` qui indique les bornes de l'axe y |
| | `main` | un `character` qui indique le titre |
| | `axes` | un booléen, qui indique si oui/non les axes/box/annotations sont représentés |
| | `type` | le type de graphique : `"p"` pour points, `"l"` pour lines, `"h"` pour traits verticals, `"n"` pour rien |
| | `xaxt` | `"n"` pour ne pas représenter l'axe et les annotations des abscisses |
| | `yaxt` | `"n"` pour ne pas représenter l'axe et les annotations des odonnées |
| `hist()` | `x` | un vecteur de valeurs `numeric` |
| | `freq` | un booléen qui indique oui/non si on représente les effectifs |

| Fonctions | Arguments | Objectifs |
|---|---|---|
| | `nclass` | nombre approximatif de classes |
| `boxplot()` | `x` | un vecteur de valeurs `numeric` |
| | `formula` | pour des boîtes à moustaches // |
| | `freq` | un booléen qui indique oui/non si on représente les effectifs |
| | `nclass` | nombre approximatif de classes |
| `barplot()` | `height` | un vecteur qui donne la taille des barres à représenter (ex : un objet de type table) |
| | `beside` | si deux variables qualitatives, est-ce qu'on représente sur plusieurs barres ou une seule |
| | `horiz` | un booléen, barre horizontale ou verticale |
| | `legend.text` | un booléen, oui/non ajout d'une légende |
| | `args.legend` | une `list` qui indique un certain nombre d'éléments de la légende |
| `pie()` | `x` | un vecteur avec les effectifs par groupes |

### 3.3 Fonctions de bas-niveau

| Fonctions | Arguments | Objectifs |
|---|---|---|
| `points()` | voir `plot()` | Ajoute un nuage de points |
| `lines()` | voir `plot()` | Ajoute des lignes continues |
| `abline()` | `h` | Ajoute un trait horizontal |
| | `v` | Ajoute un trait vertical |
| | `a`, `b` | Ajoute une droite de régression a + bx |
| `text()` | `x`, `y` | Ajoute une étiquette n'importe où dans la figure |
| `mtext()` | `side` | marge où ajouter du texte (1 en bas, 2 à gauche, 3 en haut, 4 à droite) |
| | `line` | numéro de ligne où ajouter la ligne |
| | `outer` | un booléen qui indique si oui/non il s'agit de la marge de la figure ou du graphique |
| `axis()` | `side` | où ajouter un axe (1 en bas, 2 à gauche, 3 en haut, 4 à droite) |
| | `at` | un vecteur de numeric qui indique les coordonnées des graduations |
| | `labels` | un vecteur de character qui indique les annotations |
| | `lwd` | l'épaisseur de l'axe |
| | `lwd.ticks` | l'épaisseur des graduations |

## 4 Introduction au package ggplot2

Le package **ggplot2** a été développé il y a quelques années déjà par Hadley Wickham (**RStudio**) qui a produit depuis de nombreux autres packages dont le but est de simplifier la syntaxe de base de **R**. La syntaxe que nous allons voir ici pour représenter des graphiques est complètement différente de celle que nous avons vue précédemment. Il s'agit ici d'une petite introduction.

Cette section est inspirée de ce tutoriel : Graphiques avec **ggplot2**.

Un pense-bête **ggplot2** : https://thinkr.fr/pdf/ggplot2-french-cheatsheet.pdf

```
require("ggplot2")
```

### 4.1 Le principe ggplot2 en 5 étapes

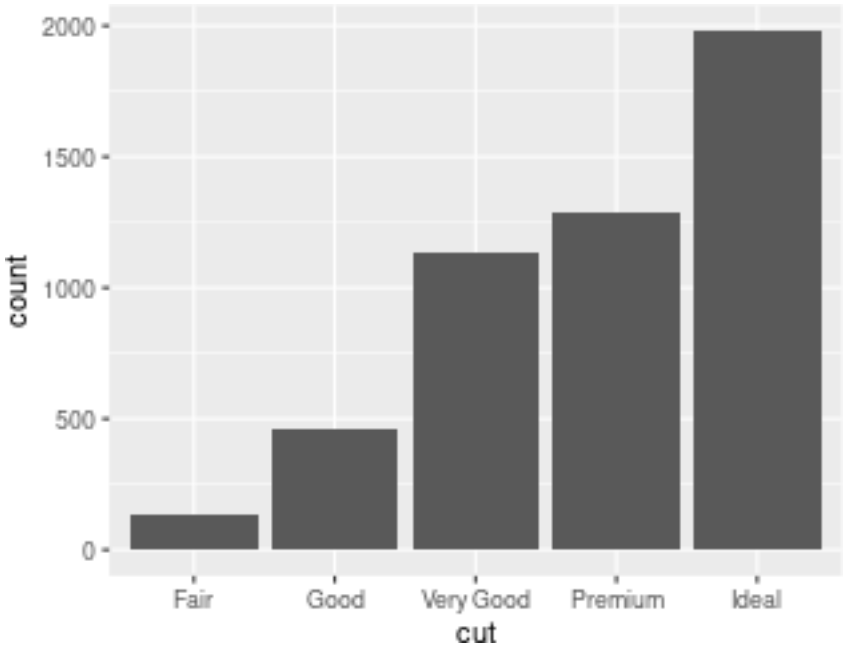
- introduire le jeu de données qui contient les variables que l'on veut représenter, celui-ci doit être sous forme de `data.frame`. Cela se fera avec la fonction `ggplot()`.

- spécification de la ou des variables à représenter, on inclut également les couleurs, les tailles des objets à représenter. Cela se fait avec la fonction `aes()`.
- spécification du type de représentation graphique souhaitée (nuage de points, diagramme en barres, histogramme, etc.). Cela se fait avec les fonctions de type `geom_XXX()`
- spécification d'éventuelles transformations des données pour la représentation souhaitée, en général reliée à une méthode statistique (densité, lissage, etc.).
- contrôler le lien entre les données et l'esthétique (modification de couleurs, gestion des axes, etc.). Cela se fait avec les fonctions de type `scale()`.

Un graphique **ggplot2** sera introduit par la fonction `ggplot()` et le symbole `+` va permettre d'indiquer quelles sont les différentes couches qui seront ajoutées au fur et à mesure.

**Exemple 1 :**

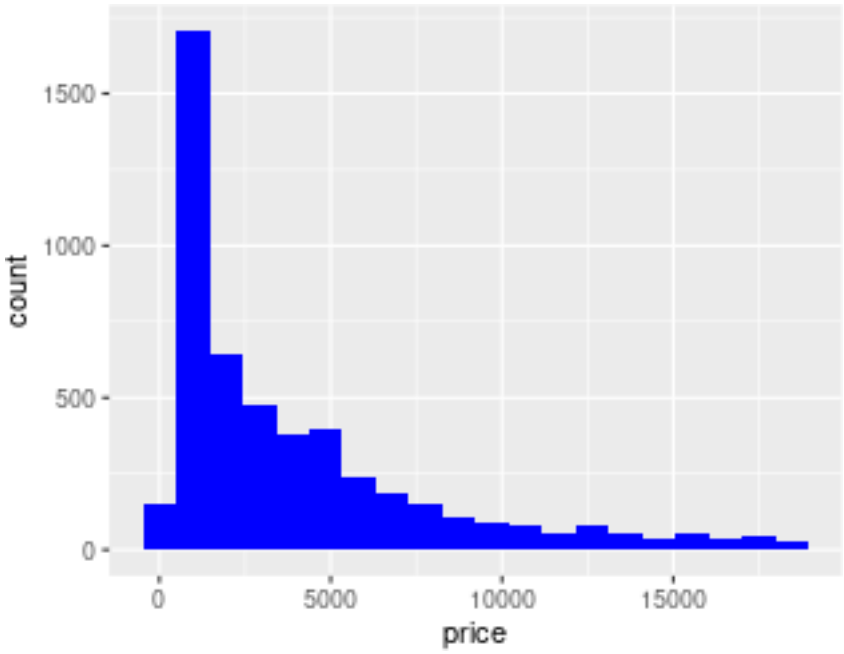
```
ggplot(diam_ech) +    # on va chercher des variables dans diam_ech, puis (symbole +)
  aes(x = cut) +      # on s'intéresse à la variable qui s'appelle cut, puis (symbole +)
  geom_bar()  # on représente un diagramme en barres de la variable appelée
```




**Remarque :** on constate dans le graphique ci-dessus qu'on n'a pas eu besoin de définir la couleur des barres, la couleur du fond d'écran, etc. car toutes ces options sont définies par défaut. Bien entendu, il est possible de modifier ces paramètres, mais cela ne se fera pas avec la fonction `par()`, comme nous l'avons vu pour les graphiques de base.

**Exemple 2 :**

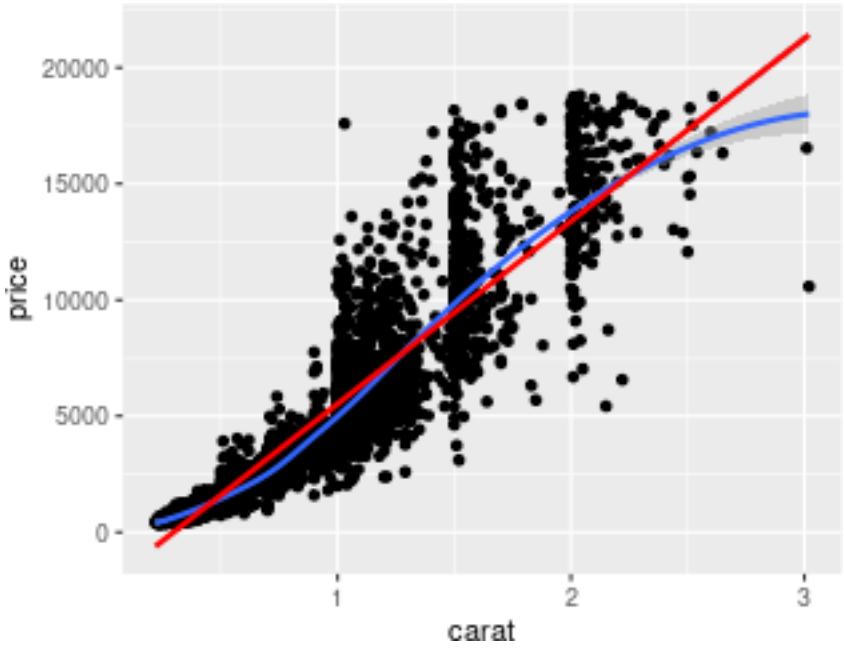
```
ggplot(diam_ech) +      # on va chercher des variables dans diam_ech
  aes(x = price) +      # on s'intéresse à la variable qui s'appelle price
  geom_histogram(bins = 20, fill = "blue")  # on représente un histogramme de cette variable
```




**Remarque :** on a utilisé l'option `bins=` qui permet de définir le nombre de barres et l'option `fill=` qui permet de changer la couleur de remplissage des barres. Les options utilisées ne portent donc pas le même nom que ceux de la fonction `hist()` vue précédemment car l'univers **ggplot2** est différent de celui des graphiques de base.

**Exemple 3 :**

```
ggplot(diam_ech) +                  # on va chercher des variables dans diam_sample
  aes(x = carat, y = price) +       # on s'intéresse aux 2 variables carat et price
  geom_point() +                    # on représente un nuage de points de ces 2 variables
  geom_smooth(method = "loess") +   # on ajoute une droite de rég. non paramétrique
  geom_smooth(method = "lm",        # on ajoute une droite de régression linéaire
              col = "red")
```

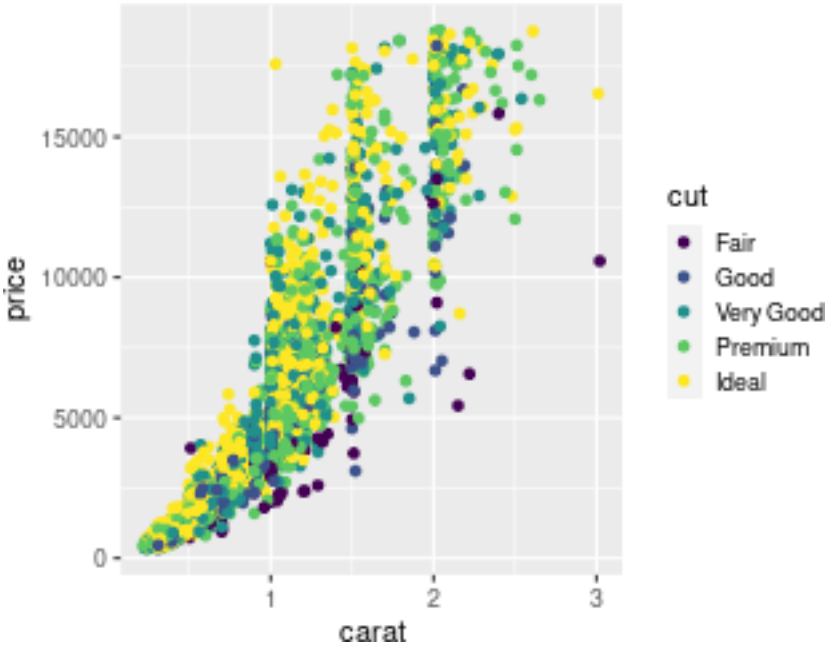


## 4.2 Les paramètres à régler

### 4.2.1 La fonction `aes()`

On a vu que la fonction `aes()` contenait les arguments `x` et `y` pour pouvoir donner le nom d'une ou deux variables à représenter. Elle contient également les options `color`, `size` et `fill` qui permettent d'ajouter le nom de variables qui serviront à représenter des objets de couleurs et tailles différentes en fonction des niveaux de ces variables. Par exemple :

```
ggplot(diam_ech) +
  aes(x = carat, y = price, color = cut) + # on ajoute le nom d'une variable à color
  geom_point()
```




### 4.2.2 Les fonctions de type `geom_XXX()`

| Geom | Description | Aesthetics |
|---|---|---|
| `geom_point()` | Nuage de points | `x`, `y`, `shape`, `fill` |
| `geom_line()` | Ligne (ordonnée selon x) | `x`, `y`, `linetype` |
| `geom_abline()` | Droite | `slope`, `intercept` |
| `geom_path()` | Ligne (ordre original) | `x`, `y`, `linetype` |
| `geom_text()` | Texte | `x`, `y`, `label`, `hjust`, `vjust` |
| `geom_rect()` | Rectangle | `xmin`, `xmax`, `ymin`, `ymax`, `fill`, `linetype` |
| `geom_polygon()` | Polygone | `x`, `y`, `fill`, `linetype` |
| `geom_segment()` | Segment | `x`, `y`, `fill`, `linetype` |
| `geom_bar()` | Diagramme en barres | `x`, `fill`, `linetype`, `weight` |
| `geom_histogram()` | Histogramme | `x`, `fill`, `linetype`, `weight` |
| `geom_boxplot()` | Boxplots | `x`, `y`, `fill`, `weight` |
| `geom_density()` | Densité | `x`, `y`, `fill`, `linetype` |
| `geom_contour()` | Lignes de contour | `x`, `y`, `fill`, `linetype` |
| `geom_smooth()` | Lissage | `x`, `y`, `fill`, `linetype` |
| Tous | | `color`, `size`, `group` |

Nous allons présenter ici seulement quelques-unes de ce ces fonctions.

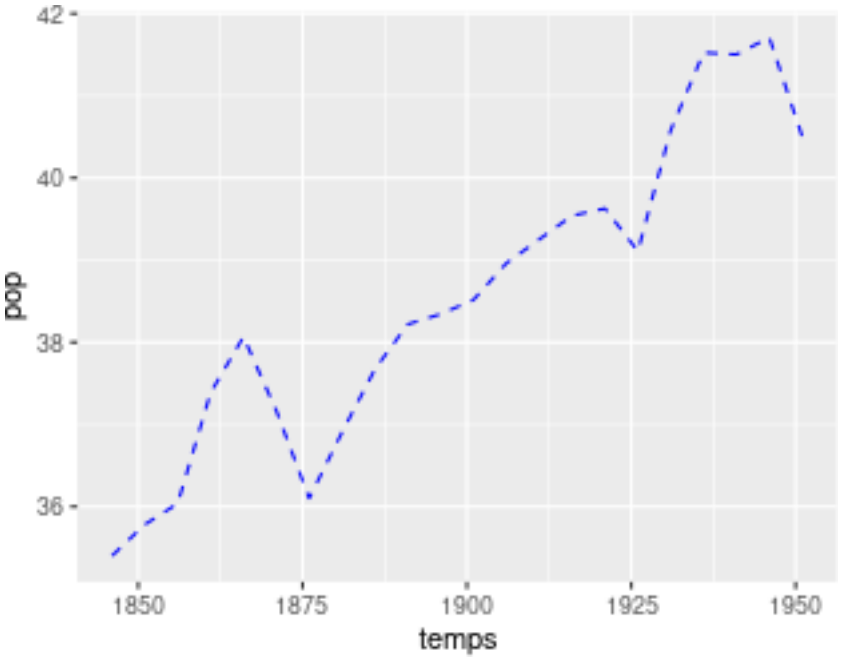
#### 4.2.2.1 Exemple de `geom_line()`

Il s'agit d'une fonction intéressante pour tracer les lignes brisées de type série temporelle. On reprend l'exemple de la population française. Il faut créer un `data.frame` pour pouvoir utiliser la syntaxe **ggplot2** :

```
popfr_df <- data.frame(temps = as.numeric(time(popfr)),
                       pop = as.numeric(popfr))
```

Ensuite, on utilise la syntaxe **ggplot2** :

```
ggplot(popfr_df) +
  aes(x = temps, y = pop) +
  geom_line(linetype = 2, colour = "blue")
```



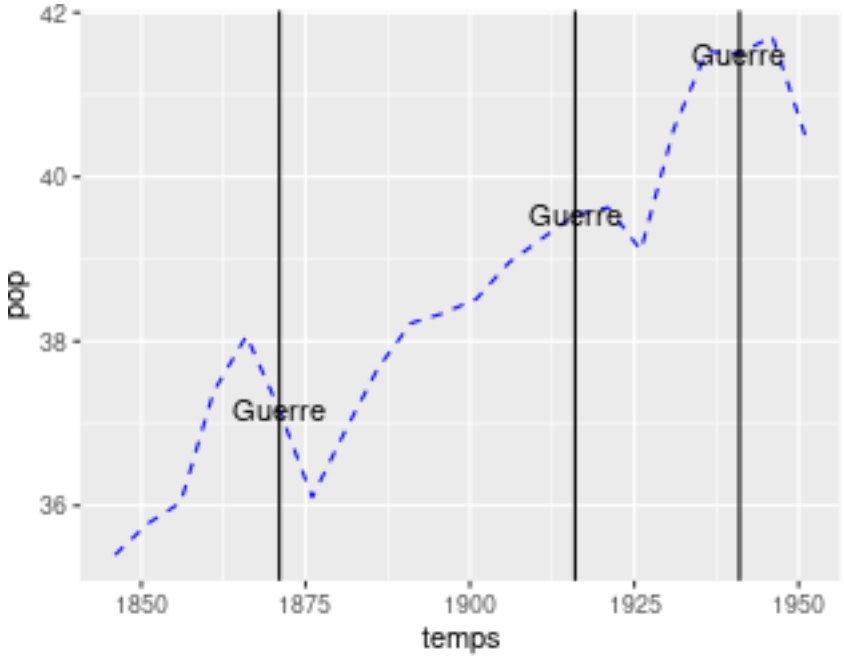

#### 4.2.2.2 Exemple de `geom_text()` et `geom_vline()`

On ajoute une colonne contenant au `data.frame` contenant les guerres :

```
popfr_df$guerre <- NULL
popfr_df[c(6, 15, 20), "guerre"] <- "Guerre"
```

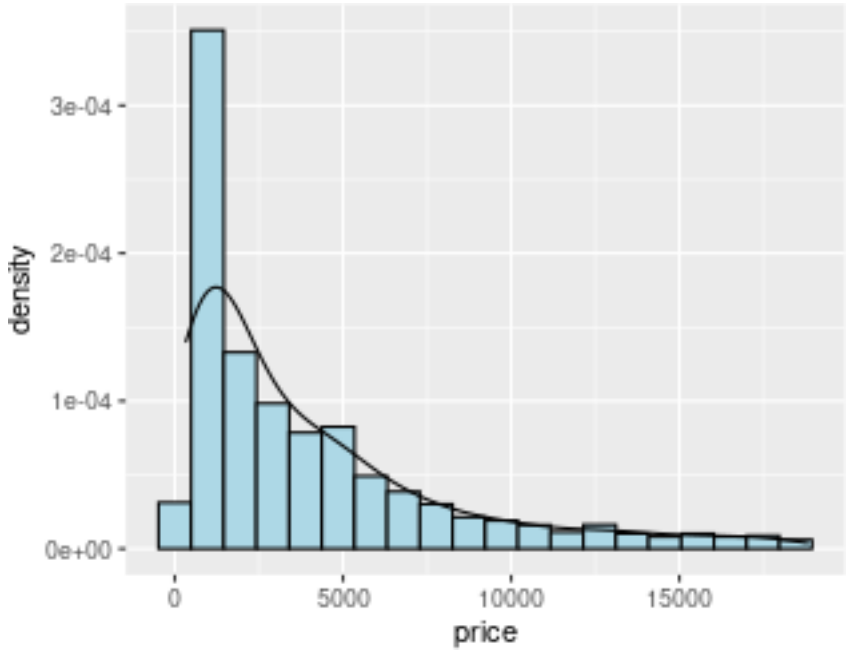
On représente les étiquettes en plus :

```
ggplot(popfr_df) +
  aes(x = temps, y = pop, label = guerre) +
  geom_line(linetype = 2, colour = "blue") +
  geom_text() +
  geom_vline(xintercept = c(1871, 1916, 1941))
```



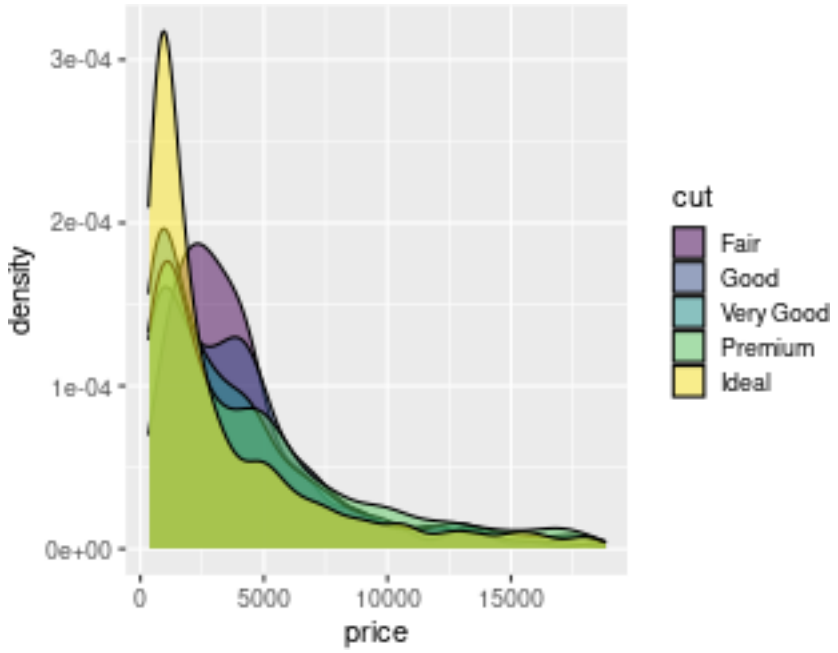

**4.2.2.3 Exemple 1 de `geom_density()`** On veut représenter l'histogramme et la densité non paramétrique de la variable `price` dans `diamants` :

```
ggplot(diam_ech) +
  aes(x = price) +
    geom_histogram(aes(y = ..density..), fill = "lightblue", colour = "black",
                   bins = 20) + # l'option bins = permet de donner le nombre de barres
    geom_density(colour = "black",
                 adjust = 2) # l'option adjust = permet de modifier le degré de lissage
```




**4.2.2.4 Exemple 2 de `geom_density()`** On va représenter simultanément sur le même graphique les densités non paramétriques de la variable `price` en fonction de `cut` :

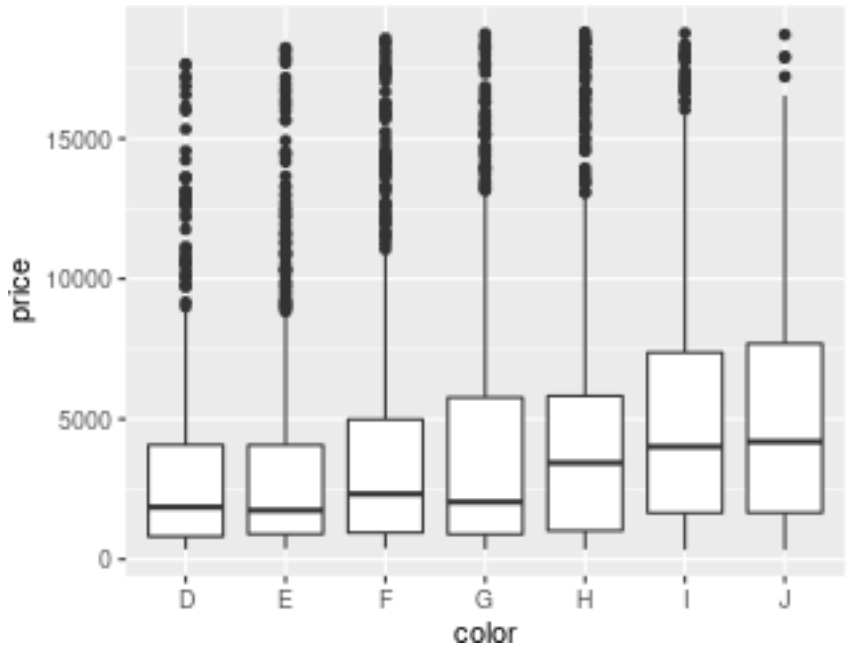
```
ggplot(diam_ech) +
  aes(x = price, fill = cut) +
  geom_density(alpha = 0.5)
```




#### 4.2.2.5 Exemple 1 de `geom_boxplot()`

On va représenter les boîtes à moustaches parallèles de `price` en fonction de `color` :

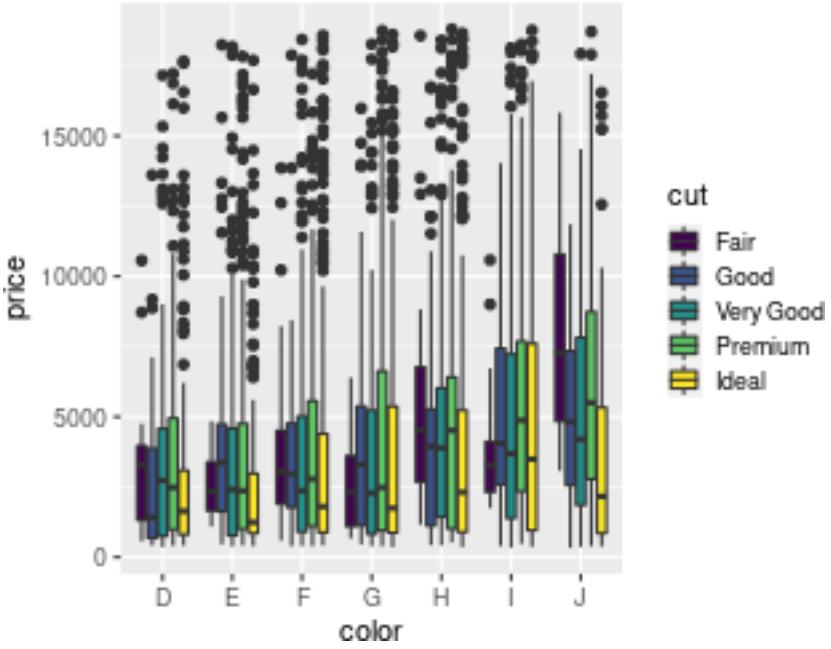
```
ggplot(diam_ech) +
  aes(x = color, y = price) +
  geom_boxplot()
```



#### 4.2.2.6 Exemple 2 de `geom_boxplot()`

On souhaiterait ajouter un second découpage de la variable `price` pour chaque modalité de la variable qualitative `color`. Il s'agit de la variable `cut`. Ceci se fait implement en ajoutant l'option `fill=cut` dans la fonction `aes()` :

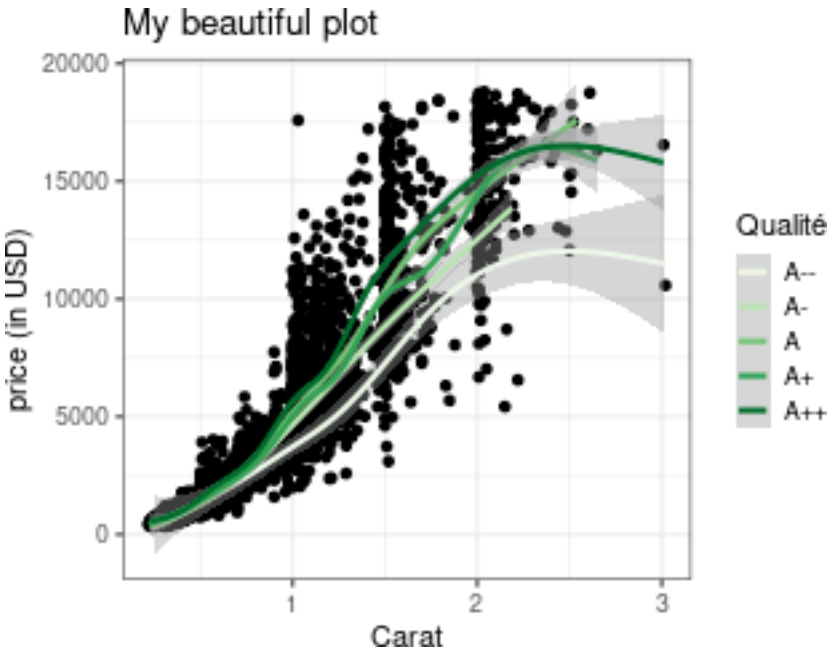
```
ggplot(diam_ech) +
  aes(x = color, y = price, fill = cut) +
  geom_boxplot()
```




### 4.2.3 Ajout de légendes

On présente ici un exemple utilisant les fonctions `theme_bw()`, `ylab()` et `ggtitle()` :

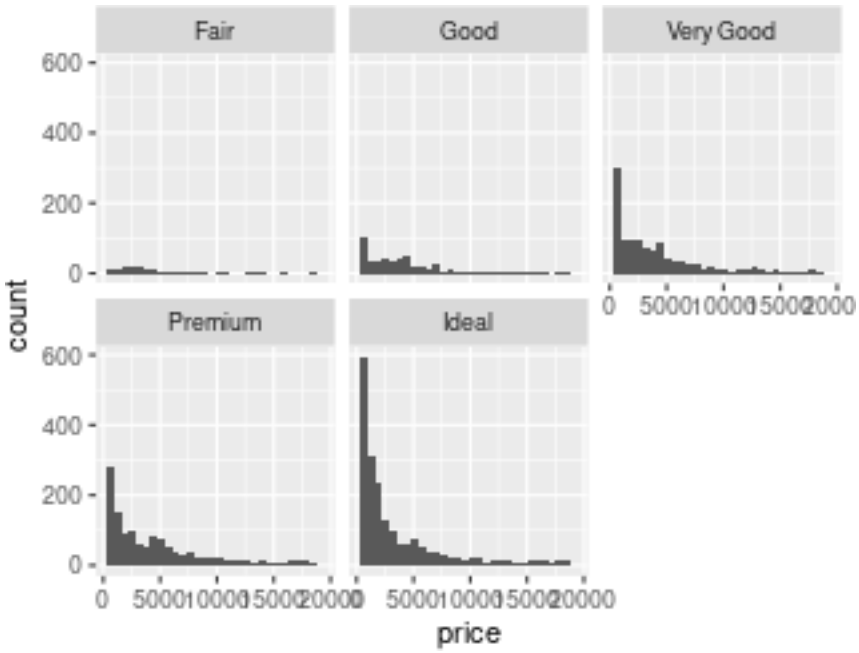
```
ggplot(diam_ech) +
  aes(x = carat, y = price) +
  geom_point() +
  geom_smooth(aes(colour = cut)) + # ajoute 5 courbes selon la variable cut
  theme_bw() +                     # modifie la couleur de fond
  xlab("Carat") +                  # modifie la légende de l'axe des x
  ylab("price (in USD)") +         # modifie la légende de l'axe des y
  ggtitle("My beautiful plot") +   # ajoute un titre
  scale_colour_brewer(name = "Qualité",   # modifie la légende de cut
                      labels = c("A--", "A-", "A", "A+", "A++") , # Etiquette
                      palette = "Greens")   # modifie la palette de couleurs
```




### 4.2.4 Graphiques conditionnels

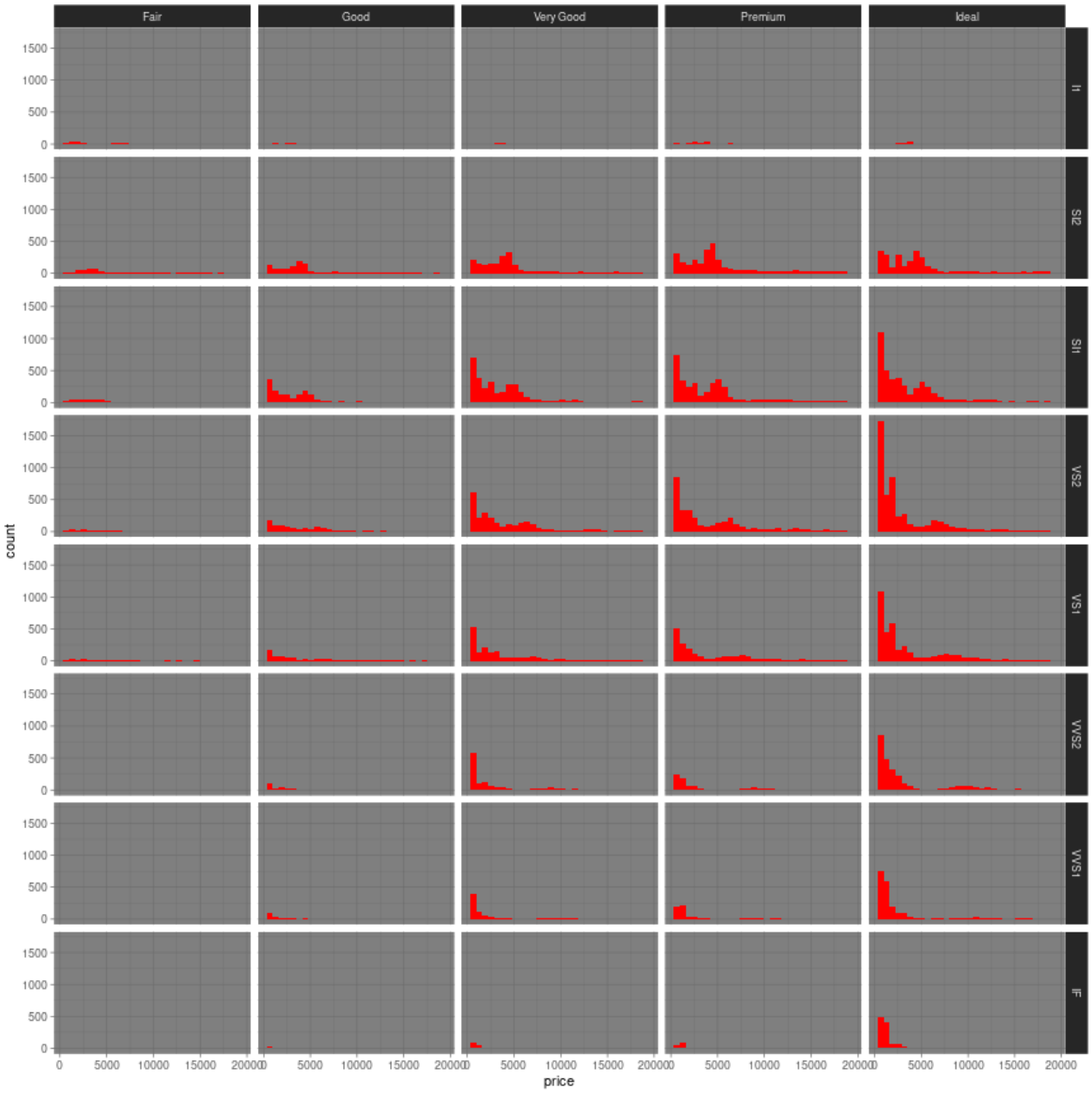
Un point fort de ce package est de permettre de faire des graphiques conditionels à une variable qualitative. Cela se fera avec les fonctions de type `facet_XXX()`. Par exemple, on souhaite représenter la distribution de la variable `price` en fonction de la variable `color`

```
ggplot(diam_ech) +
  aes(x = price) +
  geom_histogram() +
  facet_wrap(~ cut)
```




On peut également ajouter une dimension supplémentaire :

```
ggplot(diamonds) +
  aes(x = price) +
  geom_histogram(fill = "red") +
  facet_grid(clarity ~ cut) +
  theme_dark()
```

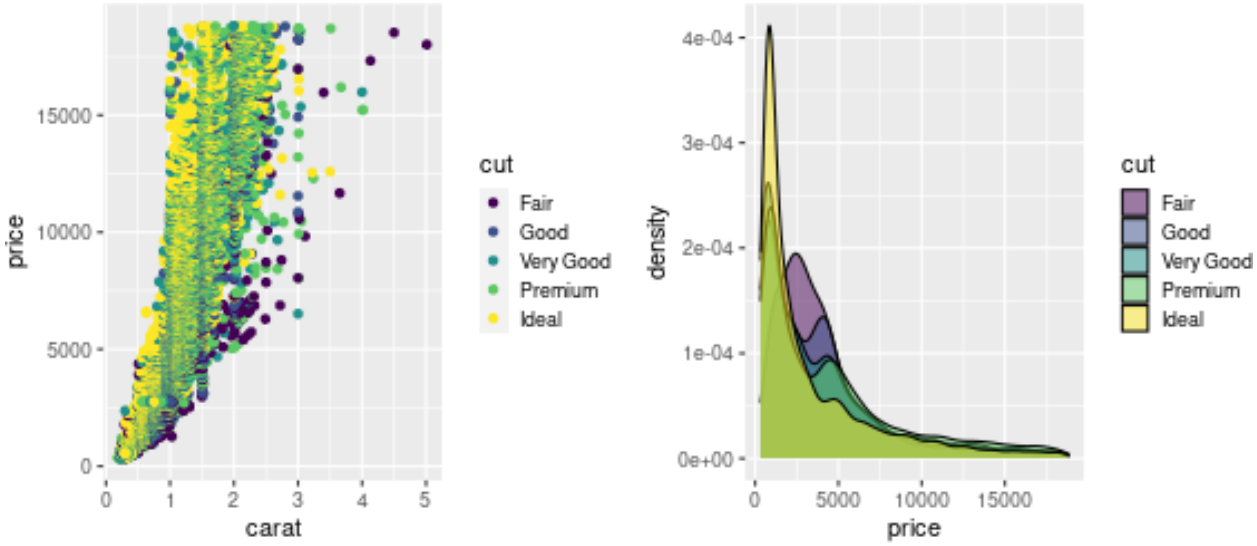



### 4.2.5 Mettre plusieurs graphiques dans la même fenêtre

Cela peut se faire avec la librairie **gridextra**. On présente ici un exemple :

```
library("gridExtra")
p1 <- ggplot(diamonds) +
  aes(x = carat, y = price, color = cut) +
  geom_point()
```

```
p2 <- ggplot(diamonds) +
  aes(x = price, fill = cut) +
  geom_density(alpha = 0.5)
grid.arrange(p1, p2, ncol = 2)
```




### 4.2.6 Sauvegarder un graphique gplot

Ceci se fait avec la fonction `ggsave()`

```
p <- ggplot(diamonds) +
  aes(x = price) +
  geom_histogram(fill = "red") +
  facet_grid(clarity ~ cut) +
  theme_dark()
ggsave("my_fig.pdf", plot = p, width = 4, height = 4)
```

**Exercice 4.6**

Choisir trois graphiques présentés ci-dessus et les appliquer sur la base de données `diam_ech` en utilisant syntaxe **ggplot2**.

**Bibliographie sur ggplot2 :**

- **R** cookbook : http://www.cookbook-r.com/Graphs/index.html
- Livre d'Hadley Wickham : **ggplot2**: Elegant graphics for data analysis