Les graphiques (partie 1)

Thibault LAURENT

17 Novembre 2014

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.

Prérequis

Avant de commencer, vous devez effectuer les opérations suivantes afin de disposer de tous les éléments nécessaires à l'apprentissage de cet E-thème.

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("Z:/Thibault Pro/cours 14-15/R/cours4")
```

2. Dans un sous répertoire nommé "Ressource", placer le fichier donnees.txt. Il contient la définition des jeux de données utilisés au cours de cet E-thème. Le charger sous R avec la fonction *source()*.

```
source("Ressource/donnees.txt")
```

Pour vérifier que le **data.frame** nommé **df** a bien été chargé, utiliser la fonction *ls()* :

```
ls()
```

```
## [1] "df"  "vec"
```

1. Les fonctions graphiques

1.1 Principe

Il existe trois grandes familles de fonctions dédiées aux graphiques :

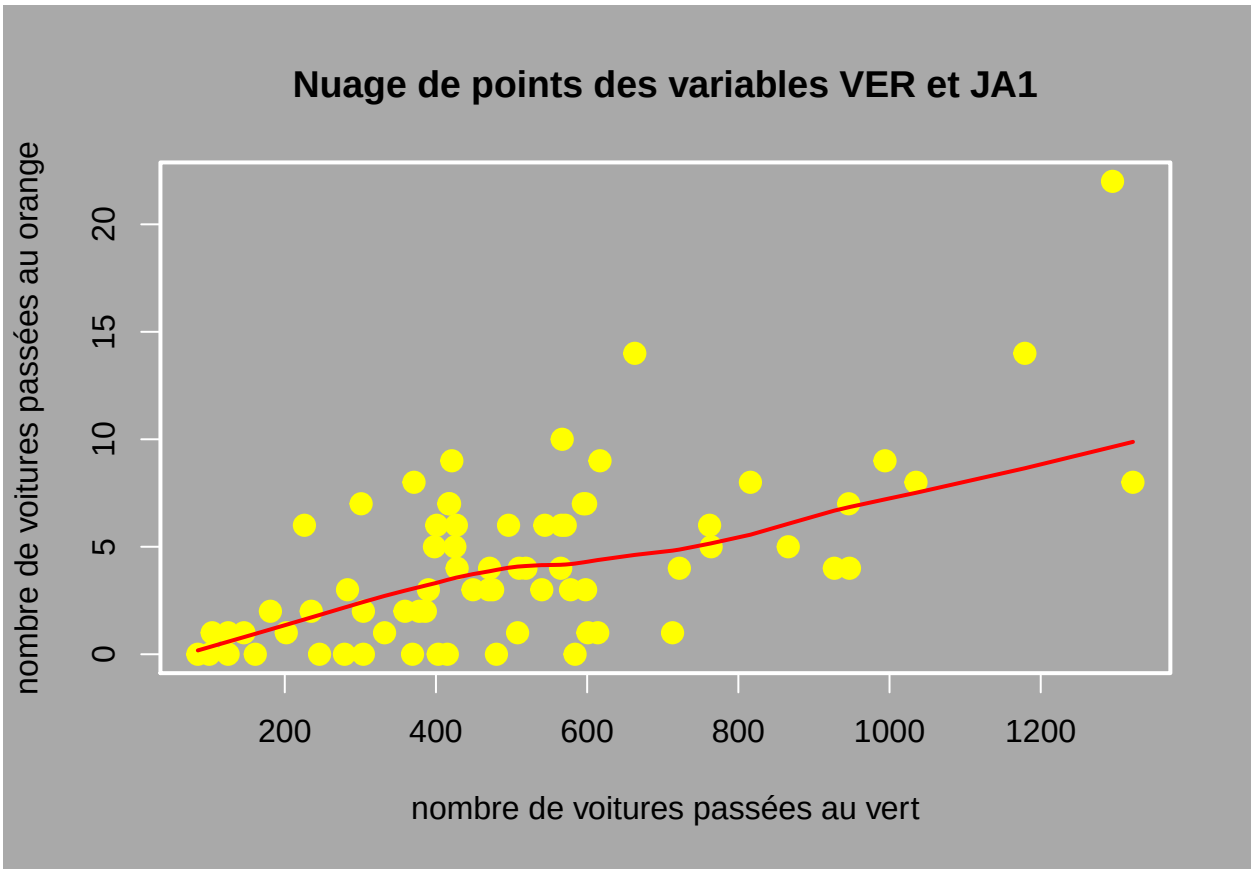
- fonctions de haut-niveau : elles permmettent d'ouvrir une fenêtre graphique et de réaliser un graphique de type : nuage de points, diagramme en barres, histogramme, etc. En principe, une fonction de haut-niveau appelée toute seule est suffisament bien programmée et paramétrée pour permettre à l'utilisateur 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.). Toutefois, si l'utilisateur n'est pas totalement satisfait du rendu de son graphique, il peut avoir recours aux deux autres types de fonctions graphiques suivantes.
- fonctions de bas-niveau : elles permettent d'ajouter des points, des lignes, des polygonnes, une légende, des étiquettes, etc. à un graphique déjà existant.
- paramétrages : ces fonctions permettent de définir/modifier le "style" des graphiques.

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 **VER** (nombre de voitures passées lorsque le feu était vert) et **JA1** (nombre de voitures passées lors de la première seconde du orange) du jeu de données **df**. Pour cela, on va utiliser un nuage de points et représenter les observations avec des cercles remplis en jaune de grosse taille. On représentera dans un second temps la courbe de régression non paramétrique de **JA1** en fonction de **VER**, 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 :
# un fond en gris (bg=), un cadre en blanc (fg=) avec trait épais (lwd=)
# (l'objet op stocke les paramètres précédents modifiés)
op<-par(bg = "darkgrey", fg = "white", lwd = 2)
with(df,{ # voir explication de with() ci-dessous
# Appel d'une fonction de haut-niveau avec en options
# - le type (pch=), la taille (cex=) et la couleur (col=) des points
# - la légende sur les axes x (xlab=) et y (ylab=)
plot(VER[TrH!="0h-24h"],JA1[TrH!="0h-24h"],
     pch = 20, cex = 2, col = "yellow",
xlab="nombre de voitures passées au vert",
ylab="nombre de voitures passées au orange")
# Appel d'une 1ère fonction de bas-niveau
lines(lowess(VER[TrH!="0h-24h"],JA1[TrH!="0h-24h"]),col=2,lwd=2)
}) # Fin de la fonction with()
# Appel d'une 2nde fonction de bas-niveau
title("Nuage de points des variables VER et JA1") # on met un titre
```




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

Remarque : la syntaxe de 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

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, on peut se retrouver encombré par un grand nombre de fenêtres ouvertes.

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) :

- Si on souhaite ouvrir une nouvelle fenêtre graphique, 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", soit de façon locale à l'intérieur de chaque fonction de haut et bas niveau.

a. 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 graphique
op<-par(bg = "darkgrey", fg = "white", lwd = 2)
# Appel d'une fonction de haut-niveau
plot(df$VER)
# Appel d'une autre fonction de haut-niveau :
# le nouveau paramétrage graphique a été conservé...
hist(df$VER)
par(op) # pour retourner au paramétrage graphique initial
hist(df$VER)
```

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 et bas-niveau (*plot()*, *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.

Autrement dit, en pratique, on a deux stratégies pour définir de nouveaux paramètres graphiques.

- On définit les nouveaux paramètres dans la fonction par() et on lance les fonctions de haut-niveau à la suite. Par exemple, on veut réaliser une série de graphiques dans lesquels on souhaite représenter tous les points en rouge avec le symbole "carré". De plus, on souhaite que la couleur des étiquette des abscisses et ordonnées soient représentée en orange. Dans ce cas, il peut être utile de définir dans la fonction *par()* ce nouveau paramétrage avant de lancer les graphiques à la suite :

```
# Appel de la fonction par() pour un nouveau paramétrage graphique
op<-par(col="red", pch=22, col.lab="orange")
# Appel d'une fonction de haut-niveau
plot(df$VER)
# Appel d'une fonction de haut-niveau
# on garde ici le nouveau paramétrage graphique
plot(df$JA1)
# Appel d'une fonction de haut-niveau
# on garde ici le nouveau paramétrage graphique
plot(df$RO1)
par(op) # retour au paramétrage graphique initial
```

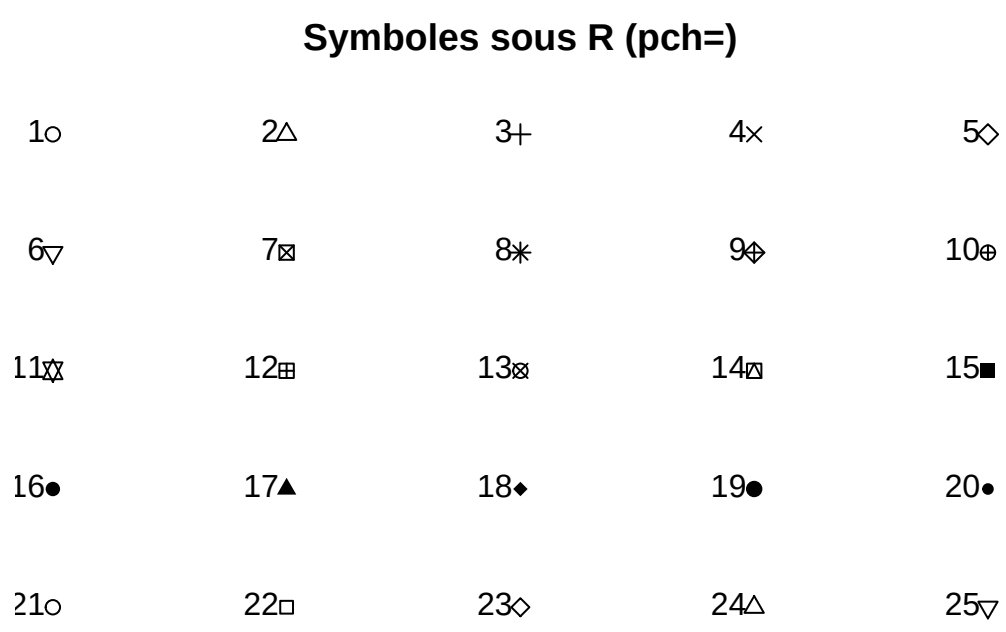
- On définit les nouveaux paramètres graphiques à l'intérieur des fonctions de haut-niveau. Ces paramètres ne seront donc appliqués que sur le graphique courant. Par exemple, on reprend la série de graphiques précédents, mais on souhaite utiliser des couleurs et symboles différents pour chaque variable. Ici, on ne fait pas appel à la fonction *par()* :

```
# Appel d'une fonction de haut-niveau
# on définit un nouveau paramétrage pour le graphique courant seulement
plot(df$VER, col="red", pch=10, col.lab="orange")
# Appel d'une fonction de haut-niveau
# on définit un nouveau paramétrage pour le graphique courant seulement
plot(df$JA1, col="darkblue", pch=5, col.lab="grey")
# Appel d'une fonction de haut-niveau
# on définit un nouveau paramétrage pour le graphique courant seulement
plot(df$RO1, col="cyan", pch=17, col.lab="magenta")
```

b. Les différents symboles pour les points

On peut représente 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 abscisses qui est vide
- **main=** donne un titre au graphique

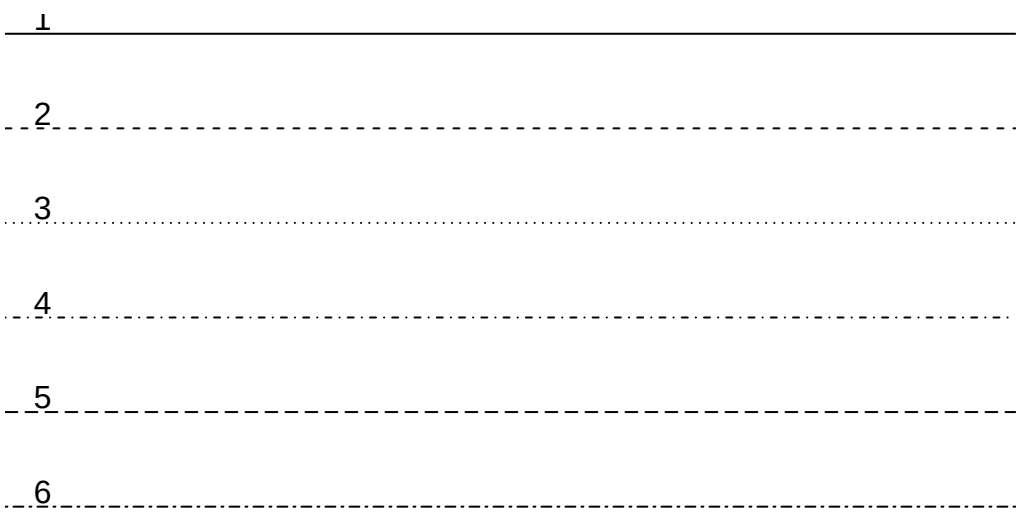
La fonction *text()* permet de représenter des étiquettes dans un graphique. Les deux premiers arguments correspondent aux coordonnées du point où sera représentée l'étiquette qu'on met en 3ème argument. Les options que nous avons utilisées sont :

- **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

c. 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)
```



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

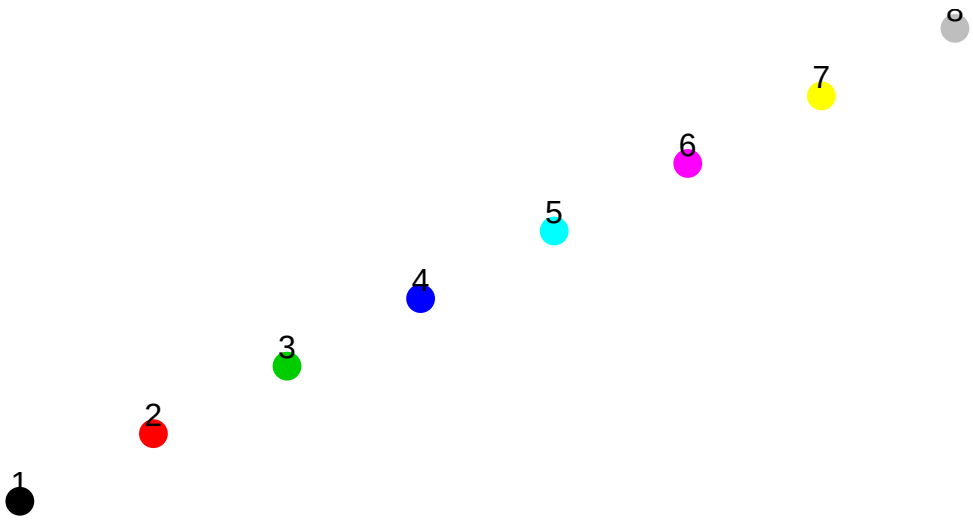
- la fonction *plot.new()* ouvre une nouvelle fenêtre graphique dans le cadre $[0,1] \times [0,1]$ sans représenter aucun axe, ni aucunes informations.
- la fonction *abline()* permet de représenter soit une droite horizontale (*abline(h=)*), soit une droite verticale (*abline(v=)*), soit une droite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

d. 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 :

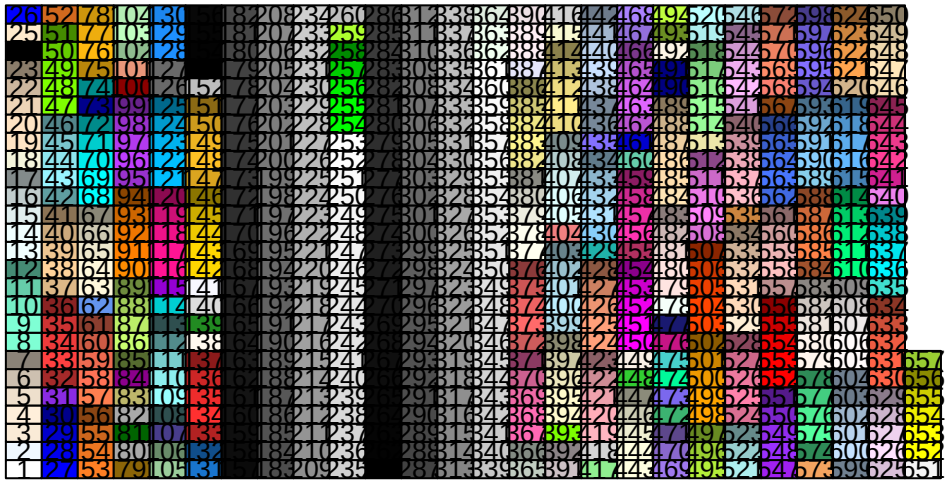
- un nombre compris entre 1 et 8 pour les couleurs de base

```
plot(1:8, col=1:8, pch=16, cex=2, axes=FALSE,
 xlab="",ylab="", main="Couleurs sous R (col=)")
text(1:8,as.character(1:8), pos=3, offset=0.2)
```



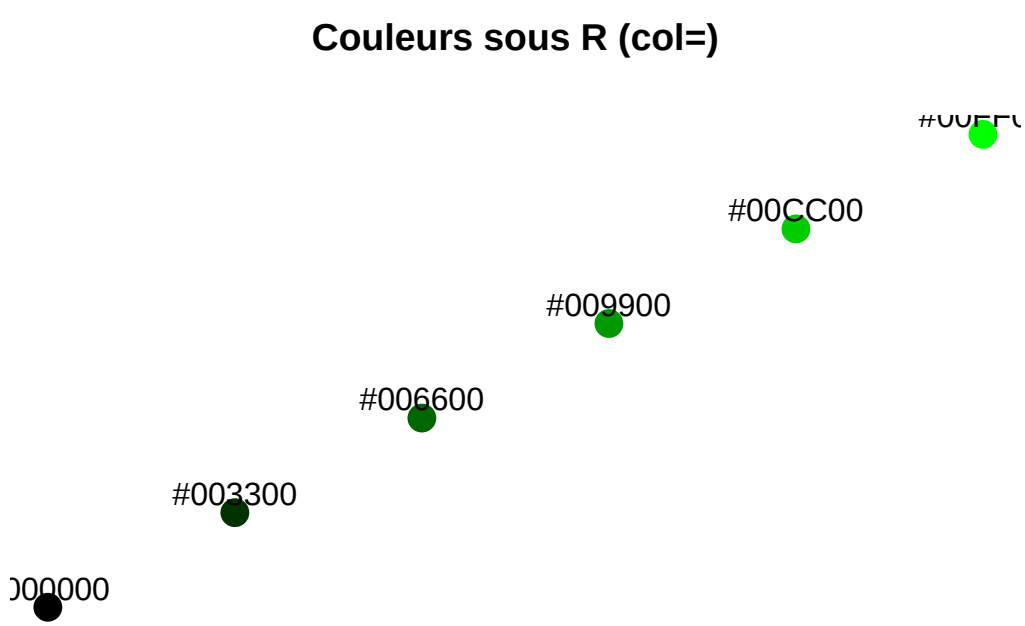
- un nom de couleurs parmi les 657 noms disponibles dans la fonction *colors()*. Pour représenter la palette de couleurs disponibles, exécuter les lignes de codes suivantes :

```
plot(1:27,type="n", axes=FALSE,
 xlab="",ylab="", main="Couleurs sous R (col=colors()[k])")
k=0 # k sera l'indice du vecteur des couleurs qu'on va 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)
 }
}
```



- 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 :

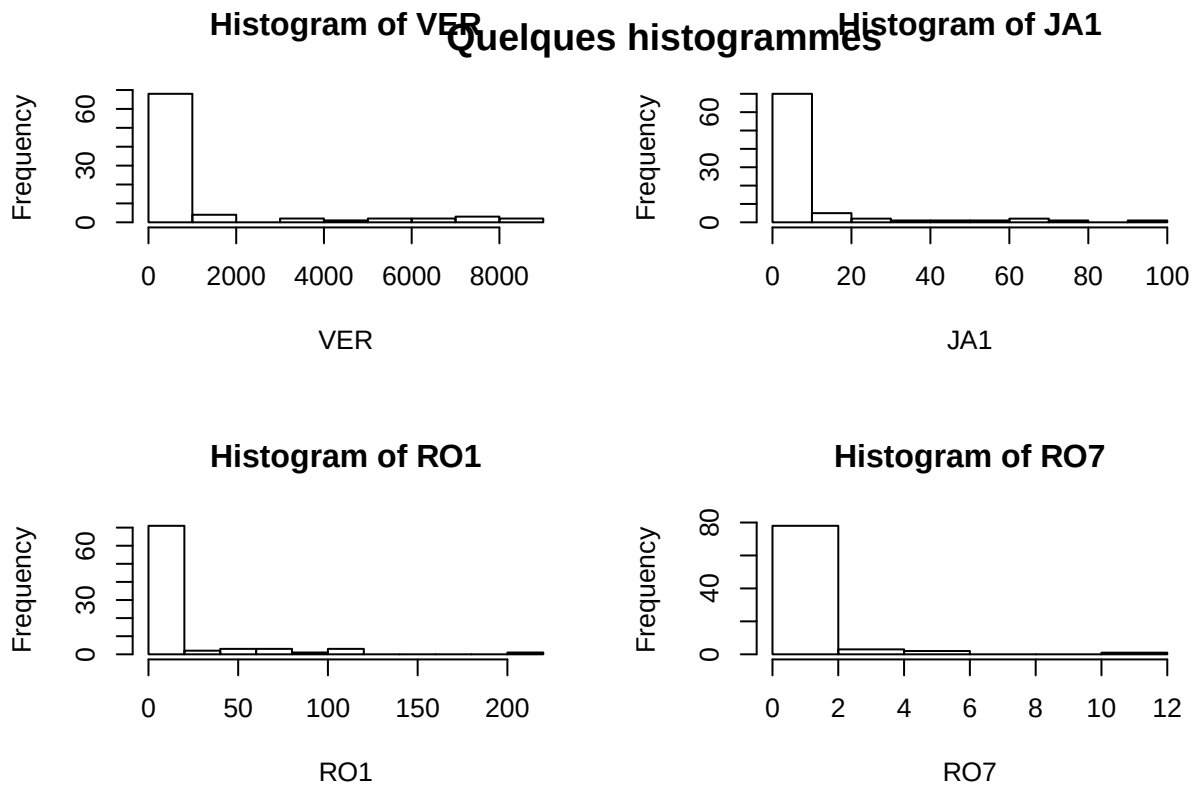
```
code.coul<-rgb(0,seq(0,1,0.2),0)
plot(1:6, col=code.coul, pch=16, cex=2, axes=FALSE,
 xlab="",ylab="", main="Couleurs sous R (col=)")
text(1:6,as.character(code.coul), pos=3, offset=0.2)
```



e. Partitionner le fenêtre graphique 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 :

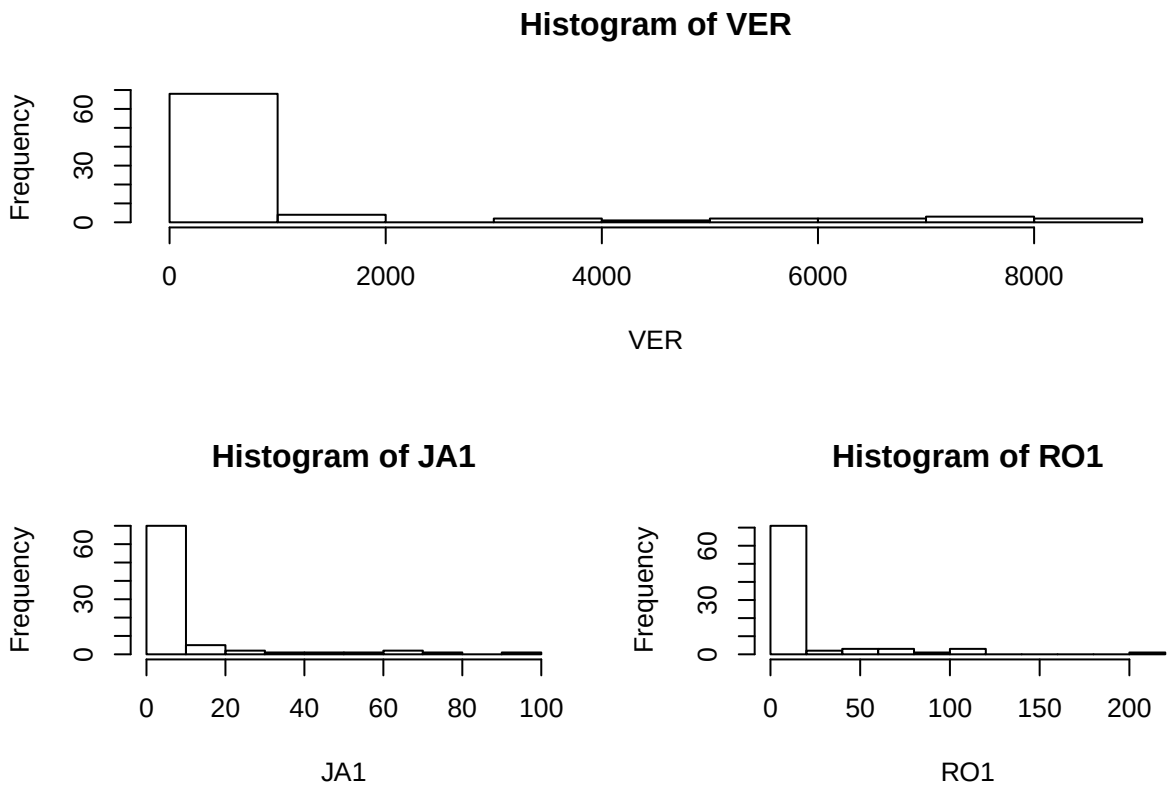
```
op<-par(mfrow=c(2,2))
with(df,{
hist(VER)
hist(JA1)
hist(RO1)
hist(RO7)
})
par(op)
title("Quelques histogrammes")
```



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(df,{
hist(VER)
hist(JA1)
hist(RO1)
})
```



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...

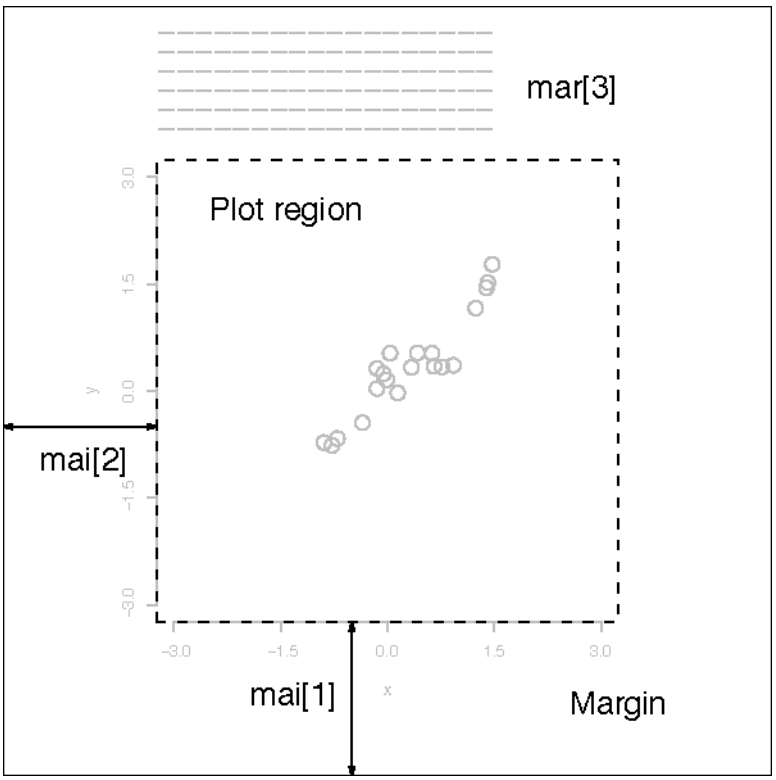
Remarque : pour plus d'informations sur le partitionnement de la fenêtre graphique, le lecteur pourra consulter cette page web.

f. Modifier les marges dans la fenêtre graphique

Pour modifier les marges dans la fenêtre graphique, il faut nécessairement modifier un des paramètres graphiques suivants dans la fonction *par()* :

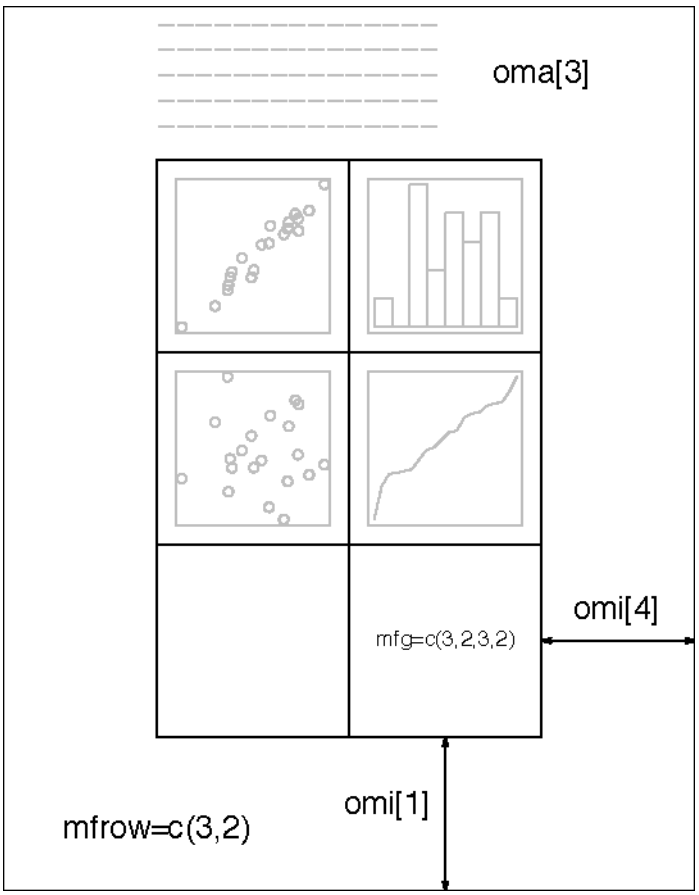
Pour modifier les paramètres sur le graphique, on modifie un de ces deux paramètres.

- **mai=c(bottom, left, top, right)** : contrôle toute la zone autour du graphique (paramètre égal à **c(1.02, 0.82, 0.82 , 0.42)** par défaut)
- **mar=c(bottom, left, top, right)** : contrôle la zone pris par les légendes (paramètre égal à **c(5.1, 4.1, 4.1, 2.1)** par défaut)



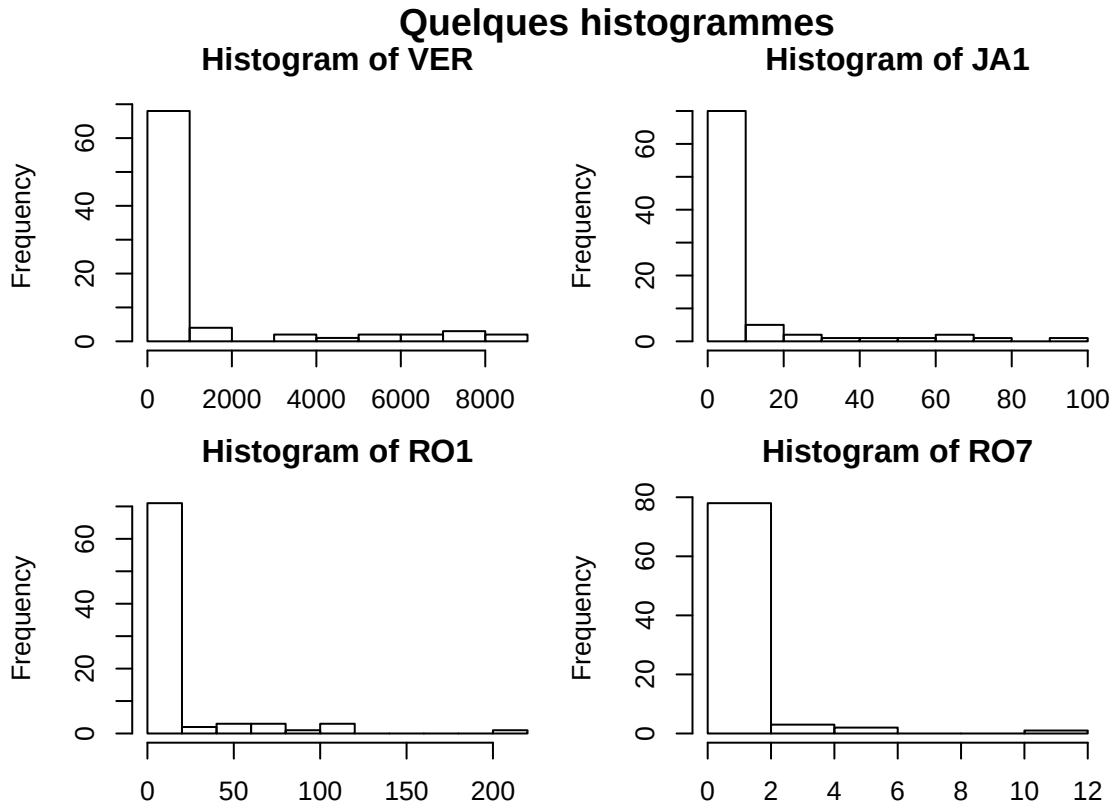
Lorsqu'on a partitionné la figure en plusieurs zones, on utilise un de ces deux paramètres pour "grignoter" de la marge par rapport aux cadres :

- **omi=c(bottom, left, top, right)** : contrôle toute la zone autour de la fenêtre graphique (paramètre égal à **c(0,0,0,0)** par défaut)
- **oma=c(bottom, left, top, right)** : contrôle la zone pris par les légendes (paramètre égal à **c(0,0,0,0)** par défaut).



Exemple : on reprend l'exemple dans lequel on avait représenté les 4 histogrammes. On veut laisser de la marge au titre global du graphique et on va rétrécir les marges à l'intérieur de chaque graphique.

```
op<-par(mfrow=c(2,2), oma=c(0.5,2,2,2), mai=c(0.4,0.6,0.6,0.15),mar=c(1.8,4,2.7,0.7))
with(df,{
hist(VER)
hist(JA1)
hist(RO1)
hist(RO7)
})
par(op)
title("Quelques histogrammes")
```

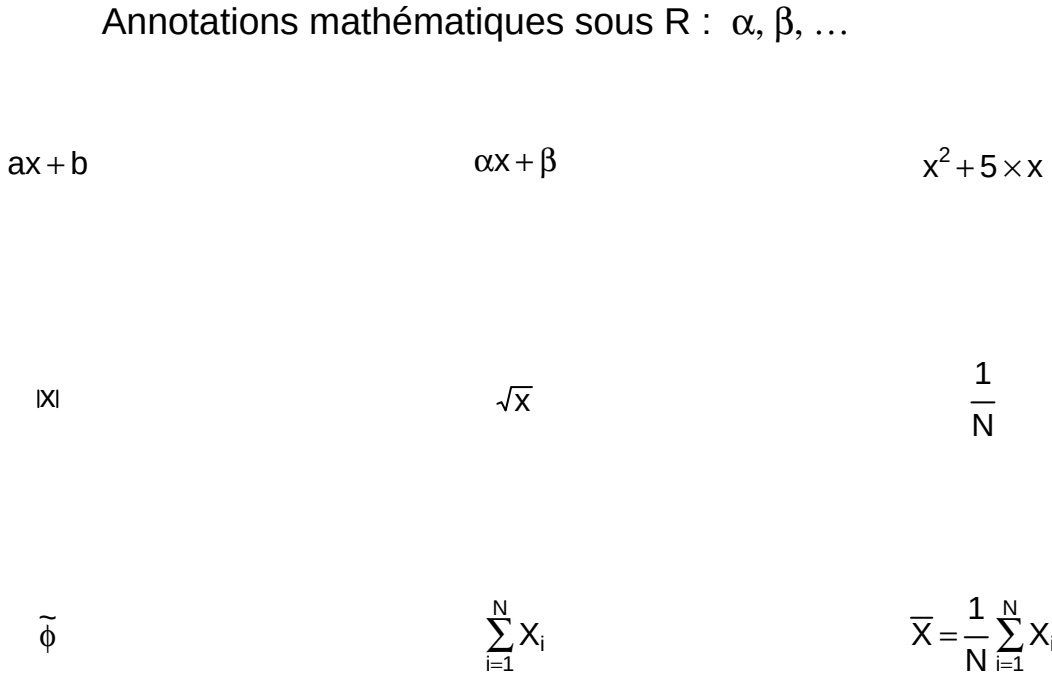


Remarque : l'ajustement des paramètres se fait à tâtons... Il faut noter que lorsqu'on modifie uniquement l'argument **mai**, l'argument **mar** est automatiquement réajusté. Cela peut donc mettre du temps avant d'arriver à un résultat totalement satisfaisant !

g. 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(c(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)
```

h. 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. Ce genre de commandes est très intéressant lorsqu'on génére un document de type Sweave car il automatise la sauvegarde d'un graphique sans avoir à faire queoi que ce soit manuellement. . .

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(0.5,2,2,2), mai=c(0.4,0.6,0.6,0.15),mar=c(1.8,4,2.7,0.7))
with(df,{
hist(VER)
hist(JA1)
hist(RO1)
hist(RO7)
```

```
})
par(op)
title("Quelques histogrammes")
dev.off()
```