Initiation à l'environnement de programmation statistique R

Monique Simier, Laure Vélez

UMR MARBEC, Dispositif Ecologie Numérique

20-21 Mars 2023

Ce cours R a pour objectif :

1. de donner un aperçu l'environnement de programmation statistique R,
2. d'acquérir les bases du langage,
3. d'apprendre à générer, importer ou exporter des données sous R,
4. d'apprendre à procéder à la description statistique d'un jeu de données avec R,
5. de décrire les principales fonctionnalités graphiques de R,
6. d'acquérir des notions sur la programmation en R,
7. de fournir une bibliographie renvoyant à quelques documents et sites web utiles.

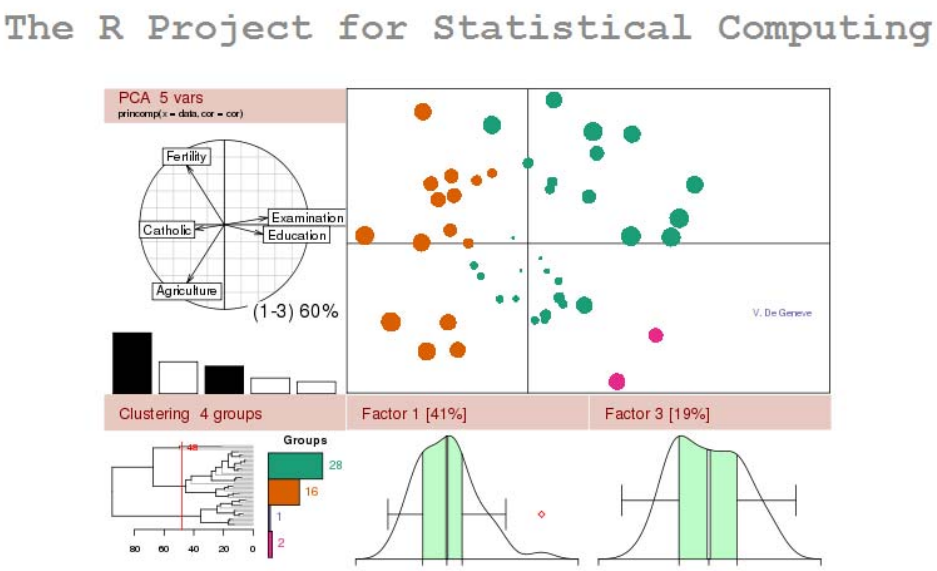
1. Un aperçu de R

1.1. Qu'est-ce que R ?

R est un **langage de programmation** et un **environnement d'analyse statistique et graphique**.

Il est basé sur le langage **S** développé dans les années 1975-1976 par John Chambers des laboratoires Bells et distribué commercialement sous la forme du logiciel **S-PLUS**. Il y a quelques différences, mais la plupart des programmes écrits en S tournent sous R.

R est un logiciel **libre**, soutenu par la *R Foundation for Statistical Computing* et distribué gratuitement sur internet : http://www.r-project.org/



R est disponible sur la plupart des plateformes : **UNIX (ou LINUX), Windows** et **MacOS**. **Ce support de cours utilise la version de R pour Windows.** Les mises à jours de R sont fréquentes, et on peut les télécharger gratuitement sur le site du CRAN (*Comprehensive R Archive Network*) : http://cran.r-project.org/ ou sur un de ses miroirs partout dans le monde.

R comporte de **nombreuses fonctions pour les analyses statistiques et les représentations graphiques** (exportables sous différents formats). C'est un logiciel très dynamique, dont la communauté d'utilisateurs ne cesse de s'élargir. Ses fonctionnalités sont en constante extension grâce à un grand nombre de « **packages** », bibliothèques de fonctions spécialisées écrites par les utilisateurs et qui peuvent être téléchargés gratuitement sur le site du CRAN.

Par rapport à un logiciel de statistique « classique » à menus, R nécessite un certain investissement pour apprendre les bases du langage. Une **documentation en ligne** est disponible et de nombreux auteurs ont écrit des ouvrages ou des documents en français ou en anglais pour apprendre R. Une liste non exhaustive est donnée en dernière page du présent document.

1.2. Installer R

Aller sur le site du CRAN (*Comprehensive R Archive Network*) : http://cran.r-project.org/ ou sur un de ses miroirs partout dans le monde. Pour Windows, cliquer sur le lien Download R for Windows ce qui a pour effet de télécharger un exécutable **R-x.y.z-win.exe** que vous stockez sur votre disque dur. **x.y.z** correspond au numéro de version : par exemple la version distribuée actuellement est la version 4.2.2.

Exécuter ensuite ce programme d'installation et acceptez toutes les options par défaut qui vous sont proposées. Un répertoire est créé : **C:\Programmes\R\R-4.2.2**. NB : vous pouvez faire cohabiter plusieurs versions de R sur votre ordinateur, elles sont installées dans des répertoires différents.

1.3. Installer l’interface RStudio

Plusieurs interfaces utilisateur ont été développées pour R. **RStudio** est la plus utilisée et fonctionne sur toutes les plateformes (Windows / Mac OS / Linux). Cette interface permet d’avoir simultanément à l’écran la console, le script R ouvert dans un éditeur beaucoup plus convivial et adapté à la syntaxe R, la liste des objets R, le résultat de l’aide, etc.

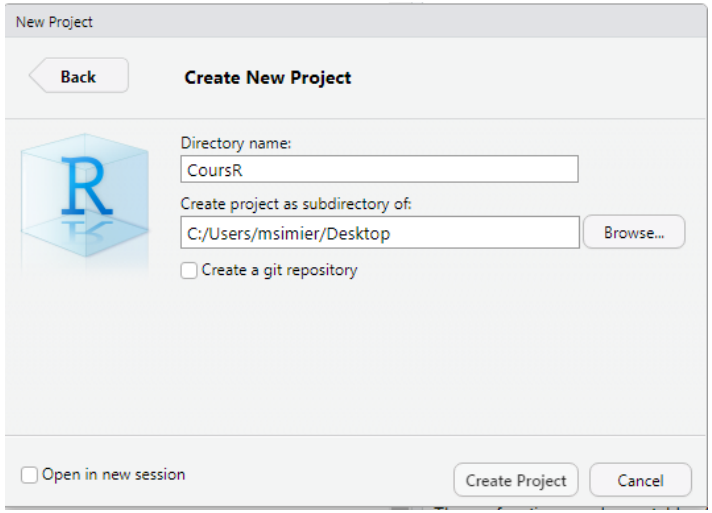
Pour l’installer, aller sur https://rstudio.com/products/rstudio/download/

Choisissez la version de **RStudio Desktop** adaptée à votre système d’exploitation et téléchargez-la, puis exécutez l'installation en acceptant tous les choix par défaut proposés.

1.3. Utiliser R avec RStudio

Lancez RStudio à partir du menu des programmes ou de la barre de tâches.

Pour commencer, créez un nouveau projet R en utilisant le menu ***File, New Project***, puis ***New directory*** (pour créer un répertoire de travail sur votre ordinateur) ou ***Open directory*** si vous avez déjà un répertoire de travail. Ici, nous créons un nouveau répertoire. A la fenêtre suivante, choisissez ***New Project*** et à la fenêtre ci-dessous, créez un répertoire ***CoursR*** sur le bureau (ou ailleurs) et validez par ***Create Project***.

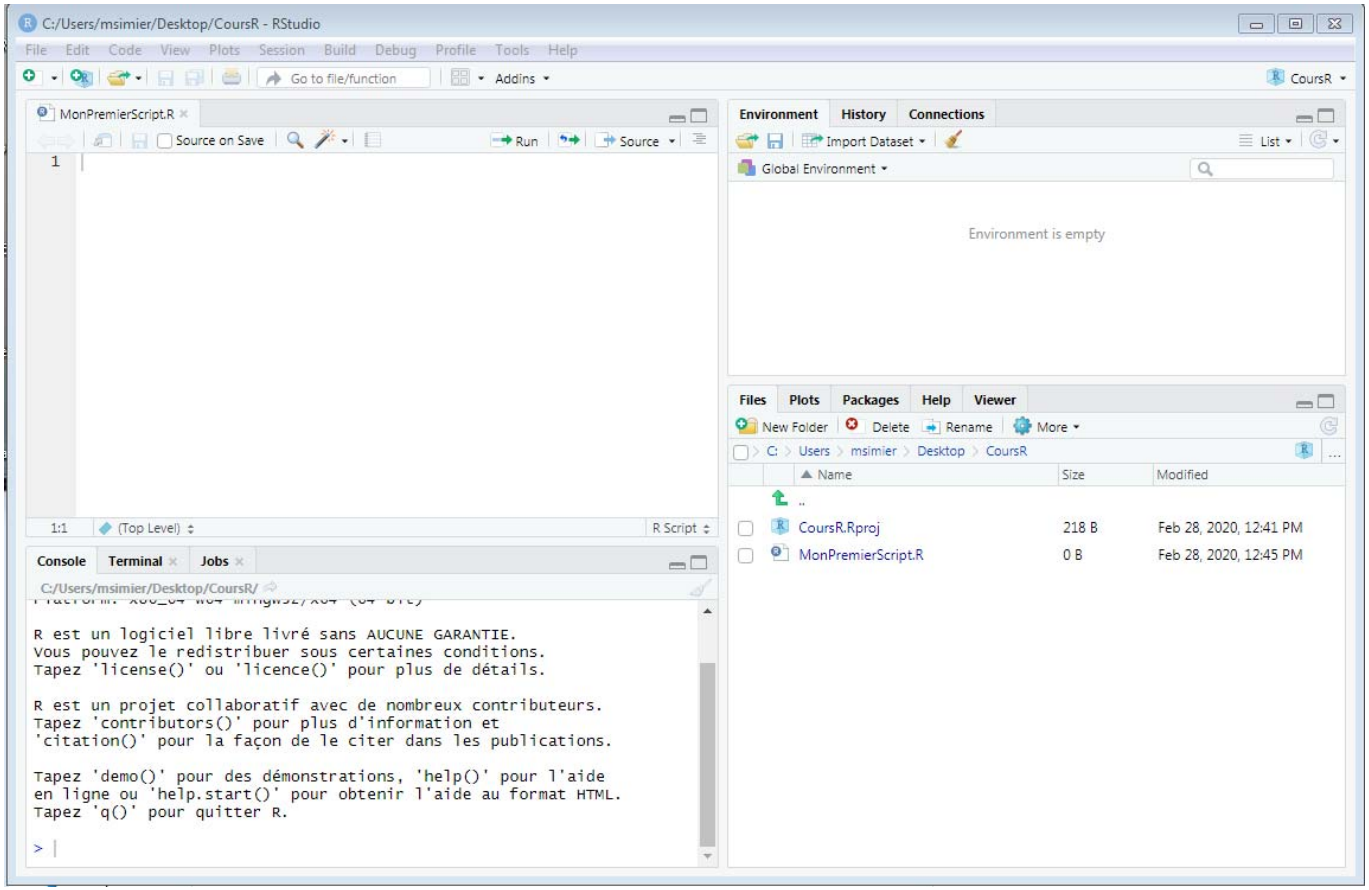


Créez un nouveau script R à l'intérieur du projet **CoursR** en utilisant ***File, New File*** et choisissez ***Rscript.*** Le nouveau script est ouvert dans une fenêtre ***Untitled1***. Pour l'enregistrer, utilisez ***File, Save as*** et donnez-lui un nom, par exemple **MonPremierScript.R**. Il est automatiquement enregistré dans le répertoire CoursR. Fermez RStudio avec ***File, Quit Session***. Vérifiez que dans le répertoire CoursR vous avez maintenant :

- CoursR.Rproj,
- le script R.

Relancez RStudio en double-cliquant sur CoursR.Rproj dans l'explorateur.

La fenêtre RStudio va ressembler à ceci :



La fenêtre en bas à droite permet de voir l'ensemble des fichiers du répertoire de travail (onglet **Files**), les graphiques (onglet **Plots**), les librairies de fonctions (onglet **Packages**), l'aide en ligne (onglet **Help**) et de visualiser du contenu web en local (onglet **Viewer**).

La fenêtre en haut à droite permet d'afficher les objets R présents dans le répertoire de travail (onglet ***Environment*** par défaut), ou l'historique des commandes (onglet ***History***).

La fenêtre en bas à gauche correspond à la **console** R. Dans cette fenêtre console, les commandes peuvent être tapées en face du signe **> (le prompt)** et les résultats s'affichent en mode texte.

La fenêtre en haut à gauche est un **éditeur** permettant de gérer un ou plusieurs **scripts** R (par les onglets). Plutôt que de taper les commandes dans la console, on utilise généralement R à partir de scripts, qui sont des successions de commandes R placées dans un fichier, une seule commande par ligne, ou si on en met plusieurs, les séparer par des « ; ». Le script peut être modifié dans cette fenêtre et sauvegardé (icône petite disquette). L'intérêt d'un script est non seulement de conserver des instructions R pour une prochaine session, mais aussi de permettre d'ajouter des commentaires : toute ligne commencée par le symbole # ne sera pas exécutée.

Pour exécuter une ligne d'un script, positionnez le curseur dessus et cliquez sur le bouton ***Run*** situé au-dessus de l'éditeur de script (ou son raccourci clavier ***Ctrl Entrée***). La commande s'exécute dans la console et le curseur se positionne automatiquement sur la ligne suivante du script. Pour exécuter plusieurs lignes à la fois, surlignez-les avec la souris et cliquez sur ***Run***. Pour exécuter l'ensemble du script, utilisez le bouton ***Source***.

Pour bien structurer votre script, utilisez ***Code, Insert Section*** et donnez un titre à la nouvelle section du script. Cela permet d'obtenir une table des matières de votre script, dans laquelle vous pouvez vous déplacer avec ***Alt Shift J*** ou en affichant la table des matières avec l'icône tout à fait à droite de la barre directement au-dessus de l'éditeur.

1.5. Quelques commandes R

Toutes les commandes de R sont sous la forme d'un nom de **fonction** prédéfini, suivi d'une **parenthèse** (), qui peut être vide, mais qui sert en général à transmettre à la fonction des **arguments** (informations pour l'exécution de la commande).

Pour connaître le nom du répertoire de travail en cours, tapez dans la console **getwd()**:

```
getwd()
[1] "C:/Users/msimier/Documents/Stage R"
```

Pour fixer un répertoire de travail, utilisez **setwd()** en indiquant le chemin complet du répertoire de travail entre guillemets. Par exemple :

```
setwd("C:/Users/msimier/Documents/Stage R")
```

NB : Il ne faut pas utiliser l'antislash **** mais le **slash /** pour séparer les éléments de l'arborescence, ou alors le doubler : \\. En effet le \ est un caractère d'échappement pour R, c'est-à-dire qu'il indique que le caractère qui suit doit être interprété de manière différente : **\n** pour retour à la ligne, **\t** pour tabulation…

Vous pouvez aussi utiliser cette commande pour remonter d'un niveau dans l'arborescence :

```
setwd("..")
```

En règle générale, on évitera de changer de répertoire de travail pendant une session R, la meilleure façon de travailler étant de placer les données et les scripts relatifs à un projet dans le répertoire associé. On peut avoir plusieurs projets en parallèle.

Sous R, vous pouvez effectuer des calculs simples dont la réponse sera donnée directement dans la console :

```
2+3
[1] 5
```

Si vous souhaitez conserver le résultat de ce calcul dans un objet R utilisez l'**opérateur d'assignement <- ou =**. Les commandes suivantes placent le résultat de l'addition dans un objet n, puis affichent le contenu de n :

```
n <- 2+3
n
[1] 5
```

R fait la différence entre majuscules ou minuscules. L'objet n sera différent de l'objet N :

```
N <- 4+6
N
[1] 10
```

On vérifie que l'objet n créé précédemment existe toujours en tapant simplement son nom :

```
n
[1] 5
```

Pour voir la liste des objets R en mémoire, tapez la commande **ls()** :

```
ls()
[1] "n" "N"
```

Si vous quittez R en choisissant de sauver une image de la session, vous verrez apparaître un nouveau fichier **.RData** dans votre dossier de travail. Il contient l'environnement R, à savoir l'ensemble des objets R créés lors de la session. Ils seront automatiquement remis en mémoire quand vous relancerez une prochaine session R depuis ce répertoire de travail.

Pour supprimer un objet R de la mémoire, utilisez la commande **rm()** :

```
rm(N)
ls()
[1] "n"
```

Pour effacer tous les objets en mémoire d'un seul coup et sans demande de confirmation (à faire en début de session R quand on veut repartir à zéro), utilisez :

```
rm(list=ls())
```

Cette commande équivaut à cliquer sur le bouton avec un pinceau au dessus de la fenêtre ***Environment***.

Pour obtenir l'aide en ligne sur une fonction R, utilisez la commande **help()** qui peut s'abréger par un simple point d'interrogation suivi du nom de la fonction :

```
help(rm)
?rm
```

Ces deux façons d'appeler l'aide de R ouvriront une nouvelle fenêtre donnant les informations relatives à la fonction rm() dans l'onglet **Help** en bas à droite de RStudio.

2. Le langage R

2.1. Elément de base du langage : le vecteur

Comme la plupart des logiciels, R peut gérer des données de type numérique, alphanumérique (ou caractère) et logique (Vrai/Faux). Sa particularité est que la structure de données élémentaire est le **vecteur, une série de valeurs de même type**. On parlera de vecteur numérique (*numerical vector*), alphanumérique (*character vector*) ou logique (*logical vector*).

Pour constituer un vecteur, on peut utiliser la fonction de concaténation **c()** :

```
v1 <- c(0,14,23)
v1
[1] 0 14 23
```

La fonction **length()** permet de connaître le nombre d'éléments constitutifs d'un vecteur.

```
length(v1)
[1] 3
```

La fonction **mode()** permet de connaître le type d'un vecteur.

```
mode(v1)
[1] "numeric"
```

Les nombres décimaux doivent toujours être encodés avec un **point décimal**, les chaînes de caractères entourées de **guillemets doubles " "**, et les valeurs logiques codées par les valeurs **TRUE** et **FALSE** ou leurs abréviations **T** et **F** respectivement. Les données manquantes sont codées par défaut par la chaîne **NA**.

```
v2 <- c(0.1,14,23.75)
v2
[1] 0.10 14.00 23.75

v3 <- c("rouge", "vert", "bleu foncé", "vert")
v3
[1] "rouge"  "vert"  "bleu foncé" "vert"
mode(v3)
[1] "character"

v4 <- c(TRUE, FALSE, F, T)
v4
[1]  TRUE FALSE FALSE  TRUE
mode(v4)
[1] "logical"
```

Un **facteur** (*factor*) est un type particulier de vecteur alphanumérique ou numérique traité comme une variable qualitative à plusieurs niveaux (*levels*). Notez ci-dessous la différence entre v3, vecteur de type alphanumérique créé plus haut et f3, facteur construit à partir de v3 par la fonction **as.factor().**

```
f3 <- as.factor(v3)
f3
[1] rouge      vert       bleu foncé  vert
Levels: bleu foncé rouge vert
```

Au facteur f3 est associé un attribut *Levels*, que l'on peut également afficher par la fonction **levels**() pour voir la liste des différentes modalités prises par le facteur f3.

Pour changer le type d'un vecteur (dans la mesure où les valeurs le permettent), utiliser les fonctions **as.numeric**, **as.character** ou **as.logical** :

```
v5 <- as.character(v1)
v5
[1] "0" "14" "23"
```

Notez les guillemets qui indiquent que le contenu de v5 est de type alphanumérique

```
v6 <- as.logical(v1)
v6
[1] FALSE TRUE TRUE
```

Lors de la conversion d'un vecteur numérique en logique, les valeurs ≠ 0 donnent TRUE et les valeurs = 0 donnent FALSE.

```
v7 <- as.numeric(v3)
Message d'avis :
NAs introduits lors de la conversion automatique
v7
[1] NA NA NA
```

Il n'est pas possible de transformer directement des valeurs alphanumériques en numériques. Cependant R crée tout de même le vecteur et le remplit de *NA* qui est le symbole de la valeur manquante (*Non Available*). Si vos données à importer sous R contiennent des valeurs manquantes, elles devront être codées par *NA*.

2.2. Opérations sur les vecteurs

Il existe sous R trois types d'opérateurs : arithmétiques, comparaison et logique.

Les opérateurs **arithmétiques** sont + (addition) - (soustraction) * (multiplication) / (division) ^ (puissance).

Les opérateurs de **comparaison** sont == (égal) != (différent) < (inférieur) > (supérieur) <= (inférieur ou égal) >= (supérieur ou égal).

Les opérateurs **logiques** sont ! x (NON logique) x & y ou x && y (ET logique) x|y ou x||y (OU logique xor(x,y) OU exclusif.

Les vecteurs peuvent être utilisés dans des **expressions logiques**, renvoyant alors un vecteur logique correspondant au résultat de l'expression appliquée sur chaque élément du vecteur de départ.

```
x <- c(10, 20, 30, 40, 50)
x>30
[1] FALSE FALSE FALSE TRUE TRUE
```

Les vecteurs peuvent être utilisés dans des **expressions arithmétiques** classiques. Les opérations sont alors effectuées élément par élément.

```
x <- c(1, 2, 3, 4, 5)
y <- c(0.5, -0.5, 0.5 , -0.5, 0.5)
z <- x*y ; z
[1] 0.5 -1.0 1.5 -2.0 2.5
```

Les différents vecteurs intervenant dans l'expression ne doivent pas nécessairement présenter la même longueur. Si ce n'est pas le cas, le résultat est un vecteur possédant une longueur identique à celle du plus long vecteur de l'expression. Les valeurs des vecteurs plus courts sont alors **recyclées**, éventuellement partiellement, de manière à atteindre cette longueur.

```
x <- c(1, 2, 3, 4, 5)
y <- c(0.5, -0.5)
z <- x*y ; z
[1] 0.5 -1.0 1.5 -2.0 2.5
```

Si cela se produit, R vous avertit que les objets en jeu ne sont pas de la même longueur :

```
Warning message:
In x * y :
  la taille d'un objet plus long n'est pas multiple de la taille d'un objet plus
court
```

L'extraction d'éléments d'un vecteur est rendue possible par l'utilisation d'un **vecteur d'index**, placé entre crochets. On peut extraire des éléments d'un vecteur de plusieurs manières :

- en indiquant par des **entiers positifs** la position des éléments à **sélectionner** :

```
x[2] # Renvoie le second élément de x
[1] 20
x[c(2,5)] # Renvoie le second et le 5eme éléments de x
[1] 20 50
```

- en indiquant par des **entiers négatifs** la position des éléments à **supprimer** :

```
x[-2] # Supprime le second élément de x
[1] 10 30 40 50
x[c(-1,-2,-3)] # Supprime les éléments 1, 2 et 3
[1] 40 50
x[-c(1,2,3)] # Supprime les éléments 1, 2 et 3
[1] 40 50
```

- par un **vecteur logique** de même longueur que le vecteur que le vecteur indexé :

```
x[x>30] # Sélectionne les éléments strictement supérieurs à 30
[1] 40 50
```

La suppression d'éléments d'un vecteur de type facteur, n’entraîne pas automatiquement la mise à jour de l’attribut levels. La fonction **droplevels**() permet de supprimer les niveaux inutilisés :

```
f3
[1] rouge      vert       bleu foncé  vert
Levels: bleu foncé rouge vert
f3 <- f3[-1]
f3
[1] vert       bleu foncé  vert
Levels: bleu foncé rouge vert
f3 <- droplevels(f3)
f3
[1] vert       bleu foncé  vert
Levels: bleu foncé vert
```

Exercice 1 :

- Créer les 3 vecteurs suivants et vérifier qu'ils apparaissent dans la fenêtre « Global environment » en haut à droite :
  - **a**, un vecteur numérique de longueur 10 contenant les dizaines de 10 à 100 : 10, 20, 30…100;
  - **b** un vecteur alphanumérique (caractère) contenant les noms de 4 planètes du système solaire;
  - **c** un vecteur logique de longueur 5, composé de vrai, faux, vrai, faux, faux
- Utiliser les différentes fonctions que nous avons vues pour vérifier le contenu, la longueur, le type de ces trois vecteurs.
- Sélectionner de 3 manières différentes les éléments n°1,3,6,8 du vecteur a. Placer le résultat dans un vecteur **a1**.
- Générer un facteur **bf** à partir du vecteur b et vérifier le type de ces deux vecteurs.
- Supprimer de **bf** le premier élément. Vérifier et mettre à jour l'attribut levels.

2.3. Autres objets manipulés par R

2.3.1. La matrice : un objet à deux dimensions

Le type matrice (*matrix*) permet de définir un objet à deux dimensions, lignes et colonnes. Comme pour les vecteurs, tous les éléments d'une matrice doivent être de même type, consultable par la fonction **mode**(). Un élément d'une matrice est accessible par deux index, respectivement ligne, colonne.

```
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE,
               dimnames = list(c("row1", "row2"),
                               c("col1", "col2", "col3")))
mdat
     col1 col2 col3
row1    1    2    3
row2   11   12   13
mode(mdat)
[1] "numeric"
mdat[2,3]
[1] 13
mdat[,3]
[1] 3  13
mdat[1,]
[1] 1  2  3
```

2.3.2. La liste

La **liste** (*list*) est une collection ordonnée d'objets **qui ne sont pas nécessairement de même type**. Les listes sont très utilisées par R pour renvoyer les résultats des fonctions.

L'exemple ci-dessous crée une liste constituée de 3 éléments : un vecteur numérique de longueur 2, un vecteur numérique de longueur 5 (y et z créés plus haut) et un vecteur alphanumérique nb de longueur 2 :

```
liste1 <- list(coeff=y, prod=z, nb=c("un", "deux"))
liste1
$coeff
[1]  0.5 -0.5
$prod
[1]  0.5 -1.0  1.5 -2.0  2.5
$nb
[1] "un"   "deux"
```

Chaque élément de la liste a un nom (ce n'est pas obligatoire, mais plus simple pour la manipulation). Ainsi le premier élément de la liste peut être extrait de deux façons :

```
# extraction du premier élément de la liste par index
liste1[[1]]
[1] -0.5 0.5

# extraction du premier élément de la liste par nom
liste1$coeff
[1] -0.5 0.5
```

On peut aussi extraire une partie d'un élément de la liste :

```
# extraction d'un sous-élément de coeff par index
liste1[[1]][1]
[1] -0.5
```

2.3.3. Le tableau de données

Les tableaux de données (*data.frame*) constituent une classe particulière de listes, consacrée spécifiquement au stockage des données destinées à l'analyse. Chaque composant (le plus souvent un vecteur) de la liste forme alors l'équivalent d'une colonne ou variable, tandis que les différents éléments de ces composants correspondent aux lignes ou individus. En général les colonnes (variables) ont des noms et on y fait référence en utilisant l'opérateur $. A cette classe data.frame sont associées une série de fonctions destinées à faciliter la visualisation et l'analyse de leur contenu (plot(), summary() …).
Pour créer un data.frame sous R, utiliser la fonction data.frame() :

```
mydf <- data.frame(taille=c(167, 170, 185, 190), poids=c(56, 68, 70, 85))
mydf
  taille poids
1    167    56
2    170    68
3    185    70
4    190    85
mydf$taille
[1] 167 170 185 190
mydf[,1]
[1] 167 170 185 190
```

Nous verrons plus loin comment importer ses données sous R en générant un data.frame.

2.4. Concaténer des objets R en lignes ou en colonnes

Les fonctions **rbind()** et **cbind()** permettent de coller des objets R (vecteurs, matrices ou tableau de données) soit par lignes (rbind pour row) soit par colonnes (cbind pour column).

```
v8 <- rbind(v1,v3)
v8
      [,1]    [,2]          [,3]
v1     "0"    "14"          "23"
v3 "rouge"  "vert"  "bleu foncé"
is.matrix(v8)
[1] TRUE
v9 <- cbind(v1,v3)
v9
     v1   v3
[1,] "0" "rouge"
[2,] "14" "vert"
[3,] "23" "bleu foncé"
```

Dans le premier cas les deux vecteurs sont collés l'un en dessous de l'autre pour constituer une matrice de dimension (2,3). Dans le second cas, ils sont collés côte à côte et constituent une matrice (3,2). Dans les deux cas, on notera la conversion automatique de type du vecteur v1 de numérique en alphanumérique car une matrice R ne peut pas contenir différents types de données.

Exercice 2 :

- Constituez une matrice **m** de dimension [10,2] à partir du vecteur a et du vecteur b. Qu'observez-vous ? Quel est le mode de m ? Affichez la première colonne de m.
- De même créez une matrice **mf** à partir de a et de bf. Quel est le mode de mf ?
- Créez une liste contenant 3 éléments : le vecteur a, le vecteur b et le vecteur c. Donnez un nom personnalisé à chaque élément de la liste. Affichez le 2ème élément de deux manières différentes.

2.5. Les fonctions et les librairies (packages) R

2.5.1. Les fonctions R

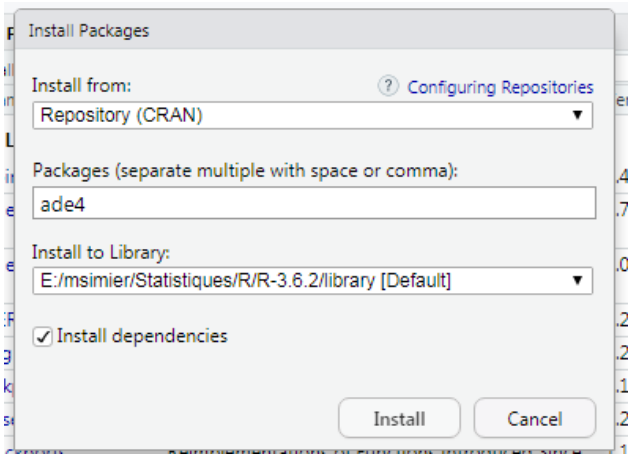
Les **fonctions** forment l'élément de base de la programmation sous R. Elles sont stockées en mémoire sous forme d'objets de type *function* et peuvent contenir une suite d'instructions R, y compris des appels à d'autres fonctions.

La syntaxe d'appel d'une fonction est simple : **nom_fonction(argument1 = valeur1, argument2 = valeur2, ...)**. Les arguments peuvent être spécifiés sous forme d'une liste nommée, la reconnaissance s'effectuant sur la base du nom de l'argument, d'une liste ordonnée, basée alors sur la position de l'argument dans la liste (l'emplacement des arguments manquants étant réservé au moyen des virgules de séparation), ou d'un mélange des deux.

```
vecteur1 <- c(1,2,3,4,5,6,7,8,9,10)
# on met dans vecteur2 le résultat de la fonction sin()
# qui calcule le sinus d'un vecteur
vecteur2 <- sin(vecteur1)
vecteur2
 [1]  0.8414710  0.9092974  0.1411200 -0.7568025 -0.9589243 -0.2794155  0.6569866
0.9893582  0.4121185 -0.5440211
```

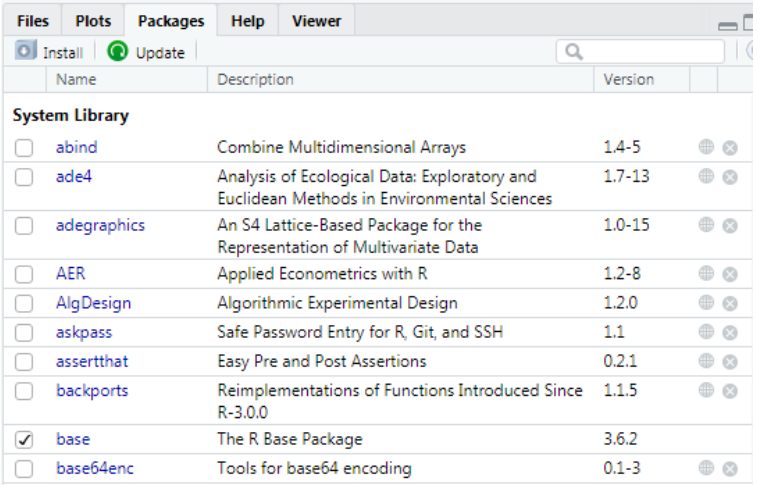
2.5.2. Les librairies de fonctions R

Le langage R est structuré en **libraires de fonctions** (packages ou libraries). Les fonctions élémentaires sont regroupées dans la librairie **base**, automatiquement chargée en mémoire au démarrage. Beaucoup d'autres sont disponibles sur le site du CRAN, et doivent être installées avant d'être utilisées. Pour cela, on utilise sous RStudio l'onglet ***Packages*** de la fenêtre en bas à droite, et on choisit ***Install*** :



Ce menu permet d'installer une librairie à partir du site du CRAN (par défaut), ou à partir d'une archive (tar ou zip). On tape le nom du /des package(s) à installer et, par défaut, il s'installe dans le répertoire correspondant à cette version de R.

La fenêtre ***Packages*** de RStudio affiche la liste des librairies installées avec un titre descriptif et le numéro de version. En cliquant sur le nom de la librairie (en bleu), on obtient une aide sur son contenu et la liste des fonctions qu'elle regroupe. Les librairies doivent être chargées à chaque session R par la commande **library(nom_librairie)**, ou en cochant la case devant le package dans la fenêtre ***Packages***.



Un changement de version de R nécessite le rechargement de toutes les librairies annexes dont la version change également. La fonction **update.packages()** permet de mettre à jour automatiquement les versions de tous les packages installés. Sous RStudio, on utilisera l'option Update de l'onglet ***Packages*** qui permet de sélectionner les packages à mettre à jour.

3. Manipulation des données sous R

3.1. Générer des données sous R

Des fonctions R permettent de générer automatiquement des vecteurs de données. Nous avons vu plus haut la fonction de concaténation **c()** qui permet de créer un vecteur. En voici quelques autres qui peuvent être utiles :

Pour générer une séquence régulière de nombres entiers, ici de 1 à 30 :

```
x <- 1:30 ; x
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30
```

Pour générer une séquence de nombres réels en spécifiant la première (ici 1) et la dernière valeur (ici 5) et l'incrément à utiliser (ici 0.5), utilisez la fonction **seq(from, to, by)** :

```
x <- seq(1, 5, 0.5) ; x
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Pour générer un vecteur dont tous les éléments sont identiques (ici 5 fois 1), utilisez **rep(x,times)** :

```
x <- rep(1, 5) ; x
[1] 1 1 1 1 1
```

Pour générer un facteur en spécifiant le nombre de niveaux (levels), ici 3 et le nombre k de répétitions pour chaque niveau (ici 5), utilisez **gl(levels, k)** :

```
x <- gl(3, 5) ; x
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3
```

On peut associer un label à chaque niveau du facteur :

```
x <- gl(2, 3, label=c("Male", "Female")) ; x
[1] Male   Male   Male   Female Female Female
Levels: Male Female
```

On peut aussi générer des séquences de données aléatoires suivant différentes lois de probabilité. Ainsi avec rnorm(n, mean=0, sd=1) on peut générer une série de n=100 valeurs aléatoires qui suivent une loi normale de moyenne 0 et d'écart-type 1 :

```
x <- rnorm(100, mean=0, sd=1) ; x
[1]  0.27970719 -0.92079279 -1.25536418 -0.17103455 …
[97]  0.64455943 -0.44523893  0.53135258 -0.75196965
```

Pour découper une variable numérique en classes et la transformer en facteur, on utilisera la fonction **cut()**, à laquelle on passe en paramètre le vecteur numérique à transformer, *breaks* qui peut être soit un vecteur des points de coupure, soit une valeur correspondant au nombre de classes souhaitées, éventuellement des labels pour les classes et *include.lowest*, une valeur logique indiquant si la plus petite valeur du premier break doit être incluse ou non. Voir **?cut** pour plus d'informations. La commande suivante découpe x en deux classes : du minimum (inclus) à 0 et de 0 (non inclus) au maximum :

```
xf <- cut(x, breaks= c(min(x), 0, max(x)), include.lowest=T,
labels=c("négatif","positif")) ; xf
[1] positif négatif négatif négatif …
[97] positif négatif positif négatif
Levels: négatif positif
```

3.2. Importer un jeu de données sous R

Toutes les lectures et écritures de données sous R se font dans le répertoire de travail. Plusieurs fonctions existent pour lire différents formats de données, issus d'autres logiciels (Excel, SAS, SPSS…), de bases de données relationnelles, de fichiers binaires ou texte. Nous nous limiterons ici à la lecture d'un fichier de données tabulaire au format texte dont les fonctions nécessaires sont dans le package base. On utilisera la fonction **read.table()** qui a pour effet de créer un objet R de type tableau de données (*data.frame*).

```
mydata <- read.table("data_in.txt")
```

Le premier paramètre (et le seul obligatoire) est le nom du fichier externe en format texte. Il doit être fourni entre guillemets et peut inclure le chemin d'accès au fichier si celui-ci ne se trouve pas dans le répertoire de travail. Voici ci-dessous quelques autres paramètres utiles, mais il y en a bien d'autres (*?read.table*).

`header`	Une valeur logique (T/F) qui indique si la première ligne contient les noms des colonnes, sinon elles s'appelleront V1, V2… header=F par défaut
`sep`	Le caractère séparant les colonnes du tableau. Par défaut c'est l'espace. Si votre fichier est délimité par des tabulations, utilisez sep= `"\t"`
`dec`	Le caractère utilisé comme séparateur decimal, par défaut c'est le `"."` mais vous pouvez définir la `","` si votre tableau est au format français.
`row.names`	Les noms des lignes. Cela peut être un vecteur de même longueur que le nombre de lignes du tableau, mais on utilise souvent le numéro d'une colonne (généralement la première) qui sert alors de label aux lignes et se trouve exclue du tableau.

Il existe aussi :

read.csv() : lit des fichiers csv au format anglais (`header=T, sep=",", dec="."`)

read.csv2() : lit des fichiers csv au format français (`header=T, sep=";", dec=","`)

read.delim() : lit des fichiers délimités par des tabulations au format anglais (`header=T, sep="\t", dec="."`)

read.delim2() : lit des fichiers délimités par des tabulations au format français (`header=T, sep="\t", dec=","`)

Un certain nombre de fonctions sont utiles pour vérifier un tableau de données sous R :

`dim(mydata)` donne les dimensions (lignes et colonnes) du tableau de données

`names(mydata)` donne les noms des colonnes du tableau de données

`head(mydata, n)` affiche les n premières lignes du tableau de données (n=6 par défaut)

`tail(mydata, n)` affiche les n dernières lignes du tableau de données (n=6 par défaut)

`str(mydata)` donne les dimensions et pour chaque colonne nom, type et un aperçu

`summary(mydata)` donne les statistiques élémentaires du tableau de données

Des jeux de données sont inclus dans R et dans les différents packages. On peut les charger en mémoire par **data(Nom_jeu_données)** et même consulter une page d'aide associée avec help() ou ? comme pour une fonction R. On utilise ici le jeu de données PlantGrowth qui donne les résultats d'une expérimentation visant à comparer les rendements (en poids sec) obtenus selon un contrôle et deux traitements différents (3 groupes) :

```
data(PlantGrowth)
help(PlantGrowth)

dim(PlantGrowth)
[1] 30  2
names(PlantGrowth)
[1] "weight" "group"
head(PlantGrowth, 3)
  weight group
1   4.17  ctrl
2   5.58  ctrl
3   5.18  ctrl
tail(PlantGrowth, 3)
   weight group
28   6.15  trt2
29   5.80  trt2
30   5.26  trt2
```

```
str(PlantGrowth)
'data.frame':    30 obs. of 2 variables:
 $ weight: num  4.17 5.58 5.18 6.11 4.5 4.61 5.17 4.53 5.33 5.14 ...
 $ group : Factor w/ 3 levels "ctrl","trt1",..: 1 1 1 1 1 1 1 1 1 1 ...
summary(PlantGrowth)
     weight       group   
 Min.   :3.590   ctrl:10  
 1st Qu.:4.550   trt1:10  
 Median :5.155   trt2:10  
 Mean   :5.073            
 3rd Qu.:5.530            
 Max.   :6.310
```

3.3. Exporter un jeu de données à partir de R

La fonction symétrique de read.table() est **write.table()** qui écrit dans un fichier texte un objet R (tableau de données ou autre) :

```
write.table(mydata, "data_out.txt")
```

Par exemple, pour exporter le jeu de données PlantGrowth au format texte, délimité par des tabulations, avec des noms de colonnes mais pas de noms de lignes et la virgule comme séparateur décimal :

```
write.table(PlantGrowth, "PlantGrowth.txt", sep="\t", col.names=T, row.names=F,
dec=",")
```

De la même manière, la fonction symétrique de read.csv() est **write.csv().**

3.4. Sauvegarder/recharger un/des objet(s) R

Il existe un format **.RData** spécifique à R, qui est compact et rapide, permettant d'enregistrer plusieurs objets R de différents types.

On utilise la commande **save** pour sauvegarder un ou des objets R de différents formats

```
save(object1, object2, object3, file="nameobject.RData")
```

Un fichier de nom 'nameobject.RData' apparaitra dans notre dossier de travail (onglet Files)

Pour le charger sous R on utilise la commande **load** :

```
load("nameobject.RData")
```

Tous les objets R contenus dans le .RData sous restaurés dans la session R avec leurs noms d'origine.

NB : La fonction **save.image()** est utilisée par RStudio pour sauvegarder l'ensemble de l'environnement de travail ('Workspace image (.RData)') quand on quitte la session

Un nouveau format de plus en plus utilisé est **.rds**
Les commandes **readRDS** et **saveRDS** permettent, comme la commande précédente, d'enregistrer des objets, mais un seul à la fois.

```
saveRDS(object, file="nameobject.rds")
```

Au chargement sous R on doit attribuer à l'objet le nom que l'on souhaite :

```
myObject <- readRDS("nameobject.rds")
```

NB : RStudio permet de charger un fichier .RData ou .rds apparaissant dans le dossier de travail (Files) en cliquant dessus. S'il est au format .RData et que le/les objet(s) qu'il contient existe(nt) déjà dans l'environnement de travail, une confirmation est demandée. S'il est au format .rds, il est restauré par défaut avec son nom 'nameobject'.

Exercice 3:

- Sous R, charger le jeu de données prédéfini **iris**, l'explorer en utilisant les fonctions vues ci-dessus. Vous pouvez utiliser l'aide R associée à iris pour en savoir plus.
- Exporter le *data.frame* iris en format texte dans le répertoire de travail sous le nom de **iris.txt**. Ensuite, vérifier la présence du fichier iris.txt dans le répertoire de travail et l'ouvrir avec un tableur.
- Dans ce tableur, ajouter une sixième colonne intitulée **codesp** contenant les trois premières lettres de la colonne Species. Enregistrer ce fichier au format texte délimité par des tabulations sous le nom **iris2.txt**.
- Puis retourner sous R et importer ce fichier de données dans un nouveau *data.frame* appelé **irisnew**. Vérifiez son contenu sous R avec dim, head, etc.
- Vous pourrez également enregistrer iris2 au format csv et le lire sous R avec la fonction correspondante.
- Sauver le jeu de données iris au format .rds. Puis le relire sous R en l'appelant **iris3**

4. Statistiques descriptives avec R

Soit une série statistique constituée par les valeurs prises par une variable (x) sur un certain nombre (n) d'observations. Reprenons l'exemple des iris de Fisher, et concentrons-nous sur la variable Sepal.Width, objet R de type vecteur numérique.

Les fonctions **min()** et **max()** donnent les valeurs minimales et maximales de ce vecteur, tandis que la fonction **range()** donne les deux à la fois, c'est-à-dire l'étendue de la distribution.

```
# affichage du contenu de la variable
iris$Sepal.Width
 [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 3.4 3.0 3.0 4.0 4.4 3.9
[18] 3.5 3.8 3.8 3.4 3.7 3.6 3.3 3.4 3.0 3.4 3.5 3.4 3.2 3.1 3.4 4.1 4.2
...

# Nombre d'observations = longueur du vecteur
length(iris$Sepal.Width)
[1] 150
# affichage des valeurs minimum et maximum et de l'étendue (range)
min(iris$Sepal.Width)
[1] 2
max(iris$Sepal.Width)
[1] 4.4
range(iris$Sepal.Width)
[1] 2 4.4
```

La variable iris$Sepal.Width est donc une série de 150 valeurs numériques, allant de 2 à 4.4.

4.1. Médiane et quantiles

Si on range les n observations par ordre croissant, la **médiane** est la valeur de x telle que 50% des observations ont une valeur supérieure et 50% une valeur inférieure.

La fonction **median()** fournit cette valeur (ici 3, ce qui signifie que la moitié des observations de sw.set ont une valeur inférieure à 3 et l'autre moitié une valeur supérieure).
La fonction **quantile()** fournit par défaut les **quartiles**, c'est-à-dire les valeurs du minimum (0%), du premier quartile Q1 (25%), de la médiane ou deuxième quartile (50%), du troisième quartile Q3 (75%) et du maximum (100%). L'intervalle interquartile (**Q3-Q1**) est un bon indicateur de la dispersion de la distribution.
Dans l'appel de la fonction quantile(), on peut ajouter le paramètres **probs=** pour obtenir des quantiles autres que les quartiles, par exemple les déciles (découpage en 10 classes d'effectifs égaux) :

```
median(iris$Sepal.Width)
[1] 3
quantile(iris$Sepal.Width)
  0%  25%  50%  75% 100%
 2.0  2.8  3.0  3.3  4.4
quantile(iris$Sepal.Width, probs=seq(0,1,0.1))
  0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
2.00 2.50 2.70 2.80 3.00 3.00 3.10 3.20 3.40 3.61 4.40
```

4.2. La moyenne et la somme

La moyenne (arithmétique) est calculée en sommant toutes les valeurs de x et en divisant par le nombre d'observations. Elle est donnée par la fonction **mean(x)**. La fonction **sum()** fait la somme de toutes les valeurs du vecteur. On retrouve la moyenne avec la formule sum(x)/length(x) :

```
mean(iris$Sepal.Width)
[1] 3.057333
sum(iris$Sepal.Width)
[1] 458.6
sum(iris$Sepal.Width) / length(iris$Sepal.Width)
[1] 3.057333
```

Par défaut, la moyenne donne le même coefficient de pondération à toutes les valeurs de x. Pour attribuer des poids différents selon les observations, il faut disposer d'un vecteur w de même longueur que x contenant les poids des observations et utiliser la fonction **weigthed.mean()** :

```
w <- rep(1,150)
length(w)
[1] 150
weighted.mean(iris$Sepal.Width, w)
[1] 3.057333
```

Pour une distribution unimodale symétrique, le mode se confond avec la moyenne. Ce n'est pas le cas lorsque la distribution est unimodale dissymétrique ou plurimodale. La moyenne, qui est un résumé fréquemment utilisé, n'est donc représentative que si la distribution des données s'y prête. Il est toujours préférable de représenter la distribution par un histogramme ou un boxplot (voir plus loin). En règle générale, la médiane est considérée comme plus robuste que la moyenne, car peu sensible aux valeurs extrêmes.

On notera que la function **colMeans()** appliquée à un tableau de données numériques donne la moyenne de toutes les variables globalement sur l'ensemble des observations. La fonction **colSums()** en fait la somme. Symétriquement, les fonctions **rowMeans()** et **rowSums()** calculent la moyenne ou la somme par ligne.

```
colMeans(iris[,1:4])
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

4.3. La variance et l'écart-type

En complément de la moyenne, qui fournit un indicateur de la valeur centrale de la distribution, il est intéressant d'avoir également un indicateur de la **dispersion** de cette distribution.

La variance (fonction **var()**) est la moyenne des carrés des écarts entre chaque valeur de x et la moyenne. On en déduit l'écart-type (fonction **sd()**), sa racine carrée, qui a l'avantage de s'exprimer dans la même unité que la moyenne et que les données :

```
var(iris$Sepal.Width)
[1] 0.1899794
sqrt(var(iris$Sepal.Width))
[1] 0.4358663
sd(iris$Sepal.Width)
[1] 0.4358663
```

4.4. La fonction summary()

La fonction **summary**(), déjà évoquée plus haut, permet d'obtenir un résumé de la description statistique univariée d'une ou plusieurs variables, adaptée selon qu'elles sont continues ou discrètes. Elle est utile pour vérifier rapidement que les données que l'on vient d'importer sous R sont correctes.

```
summary(iris$Sepal.Width)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.800   3.000   3.057   3.300   4.400
```

```
summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
 Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

4.5. Régression linéaire avec la fonction lm()

La fonction **lm**(), pour *linear model* permet de faire la régression linéaire d'une variable continue y en fonction d'une ou plusieurs variables x1, x2… qui peuvent être continues ou discrètes. Cette régression est de la forme y=ax+b, ou b est l'ordonnée à l'origine et a la pente. Comme pour toutes les fonctions R, le résultat de lm() est une liste qui peut être affichée dans la console ou placée dans un objet R. La régression linéaire simple (une seule variable explicative) s'écrit lm(y~x). Dans l'exemple suivant, on teste la régression entre y=iris$Petal_Length et x=iris$Sepal_Length et le résultat est placé dans un objet m :

```
m <- lm(Petal.Length~Sepal.Length, data=iris)
```

```
summary(m)
Call:
lm(formula = Petal.Length ~ Sepal.Length, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-2.47747 -0.59072 -0.00668  0.60484  2.49512

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  -7.10144    0.50666  -14.02   <2e-16 ***
Sepal.Length  1.85843    0.08586   21.65   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.8678 on 148 degrees of freedom
Multiple R-squared:   0.76,   Adjusted R-squared:  0.7583
F-statistic: 468.6 on 1 and 148 DF,  p-value: < 2.2e-16
```

Le résultat rappelle la formulation du modèle, fournit la distribution des résidus (qui doit respecter une loi normale), donne les coefficients du modèle, l'ordonnée à l'origine b (intercept) et la pente a, avec les tests de significativité associés (ici les deux coefficients sont significativement différents de 0), le coefficient de détermination R^2 et sa valeur ajustée.

Les éléments de la liste sont accessibles :

```
m$coef
(Intercept) Sepal.Length
   -7.101443     1.858433
```

Dans le cas où on a plusieurs variables explicatives x1, x2… elle seront séparées par des + pour obtenir un modèle additif ou des * pour prendre en compte l'interaction. Ci-dessous on ajoute au modèle précédent la variable discrète iris$Species. Dans le cas d'une variable discrète, la première modalité par ordre alphabétique (setosa ici) est prise comme référence. L'ordonnée à l'origine et la pente indiquées sur les deux premières lignes correspondent à cette modalité de référence (ici b=0 .8031 et a=0.1316).
On trouve sur les deux lignes suivantes la correction à apporter à l'ordonnée à l'origine pour les deux autres modalités (pour versicolor, l'ordonnée à l'origine est 0.8031- 0.6179 soit 0.1852). Pour aucune des 3 variétés l'ordonnée à l'origine n'est significativement différente de zéro.
Puis les deux dernières lignes indiquent l'interaction entre les x, soit la pente pour versicolor et virginica. Dans ce modèle seules les interactions sont très significativement différentes de 0, indiquant une différence de pente significative entre setosa et chacune des autres variétés

```
m1 <- lm(Petal.Length~Sepal.Length*Species, data=iris)
```

```
summary(m1)
Call:
lm(formula = Petal.Length ~ Sepal.Length * Species, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-0.68611 -0.13442 -0.00856  0.15966  0.79607

Coefficients:
                             Estimate Std. Error t value Pr(>|t|)
(Intercept)                    0.8031     0.5310   1.512    0.133
Sepal.Length                   0.1316     0.1058   1.244    0.216
Speciesversicolor             -0.6179     0.6837  -0.904    0.368
Speciesvirginica              -0.1926     0.6578  -0.293    0.770
Sepal.Length:Speciesversicolor  0.5548     0.1281   4.330 2.78e-05 ***
Sepal.Length:Speciesvirginica   0.6184     0.1210   5.111 1.00e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.2611 on 144 degrees of freedom
Multiple R-squared:  0.9789,  Adjusted R-squared:  0.9781
F-statistic:  1333 on 5 and 144 DF,  p-value: < 2.2e-16
```

Exercice 4:

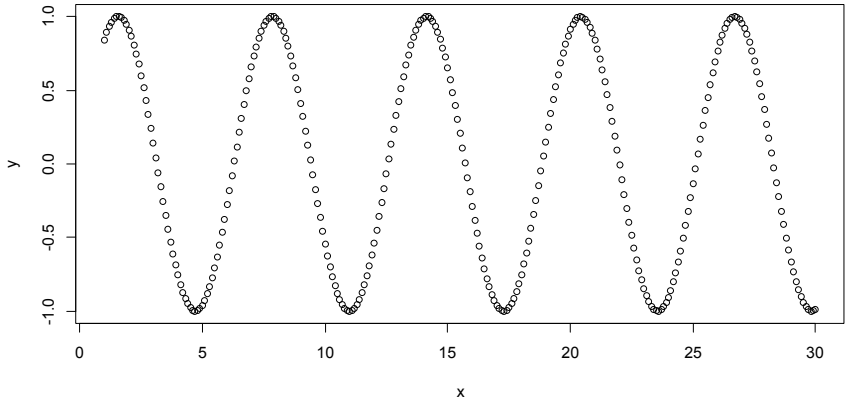
- Reprenez le jeu de données Plantgrowth vu plus haut. Utilisez les fonctions que l'on vient de voir pour faire une description statistique de la colonne weight : étendue de la distribution, moyenne, variance, écart-type, médiane et quartiles.
- Sur ce même jeu de données faire une régression de la variable weight en fonction de la variable group. Quelles conclusions en tirez-vous ?

5. Les graphiques avec R

5.1. Gestion des graphiques sous R

Prenons un exemple simple, qui trace un graphe en xy (*scatterplot*) à partir d'une série de données x en abscisse et y en ordonnée. Par défaut, sous RStudio, le graphe s'affiche dans la fenêtre **Plots** en bas à droite. On peut naviguer parmi les différents graphiques créés avec les flèches à droite ou à gauche du menu de cette fenêtre. On peut aussi faire un zoom ou exporter le graphique en différents formats (JPEG, EPS, PDF…) ou le coller dans le presse-papier (*Copy to Clipboard*) avec **Export**.

```
x <- seq(1,30,0.1)
y <- sin(x)
plot(x,y, type="p")
```



On peut aussi, avant l'appel à la fonction graphique, ouvrir une nouvelle fenêtre graphique avec la fonction **x11()** qui fonctionne sur Linux et Windows ou son équivalent **windows()** (Windows uniquement), ou encore **quartz()** sous MacOS. Le graphique s'affichera toujours dans la dernière fenêtre graphique ouverte qui devient le périphérique (*device*) actif. Pour le sauvegarder sous forme de fichier ou dans le presse-papier, utiliser le menu contextuel en se positionnant sur la fenêtre graphique et en cliquant sur le bouton de droite de la souris.

On peut également ouvrir directement un fichier graphique avec la fonction **pdf()**, ou **jpeg()**… Taper **?Devices** pour connaître les différents périphériques utilisables.

```
windows()           # ouvre une fenêtre graphique
x11()               # idem
pdf("mygraph.pdf") # ouvre un fichier pdf dans répertoire courant
```

Pour connaître la liste des périphériques ouverts :

```
dev.list()
windows windows     pdf
      2       3       4
```

Pour connaître le numéro du périphérique actif :

```
dev.cur()
pdf
4
```

et pour changer le périphérique actif :

```
dev.set(3)
windows
3
```

Pour générer un graphe dans un fichier pdf, y placer le résultat du plot et le fermer :

```
pdf("mygraph.pdf")
plot(x,y, type="l") # crée le graphe dans le périphérique actif
dev.off()           # ferme le périphérique actif
windows
      2
```

Vérifier ensuite dans le répertoire de travail qu'un fichier **mygraph.pdf** a bien été créé. Il est indispensable de fermer le périphérique par **dev.off()** pour pouvoir ensuite l'ouvrir avec Acobat Reader.

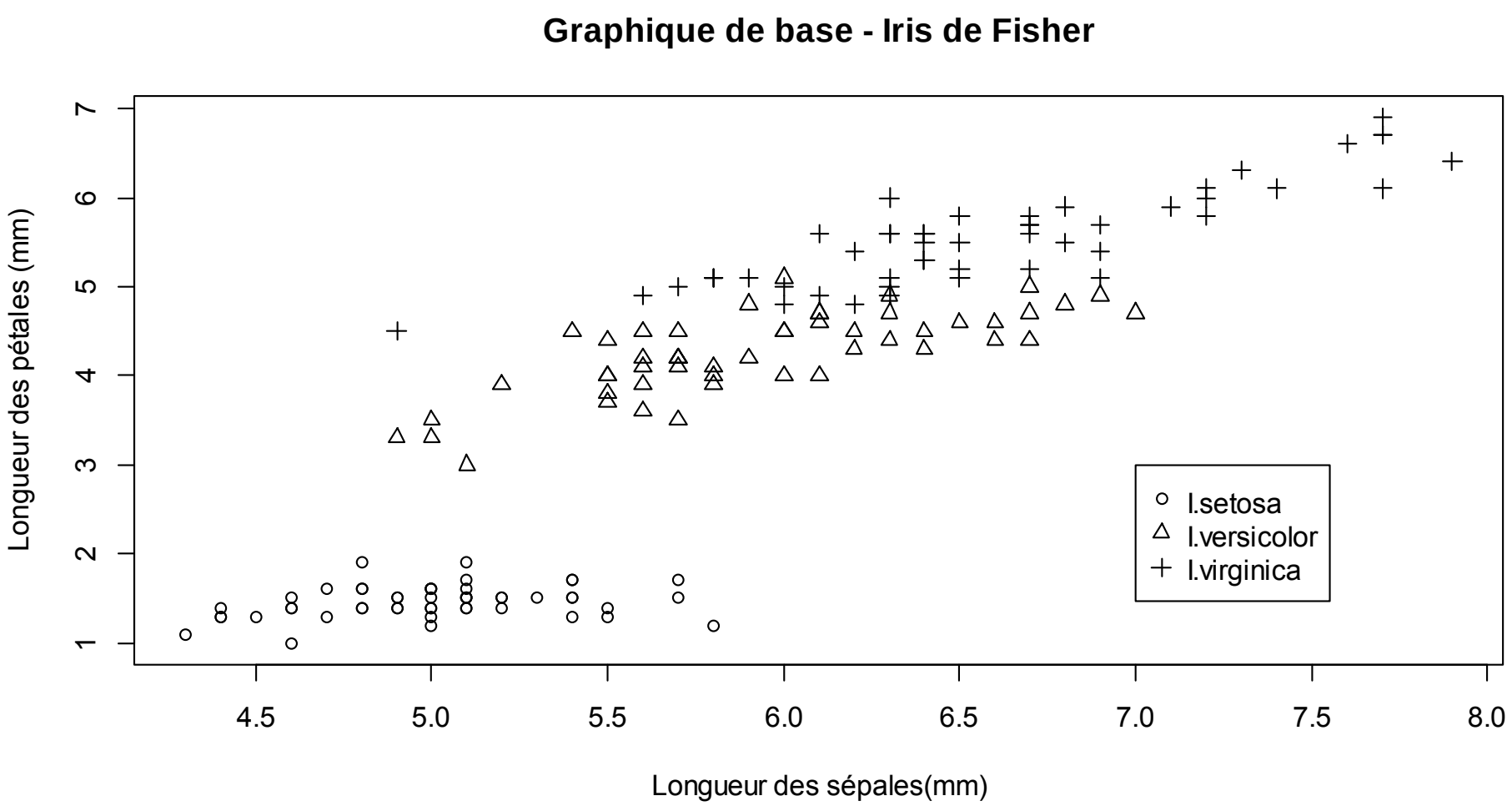
Il est possible de partitionner une fenêtre graphique en utilisant différentes fonctions de R comme **split.screen()**, **layout()**, ou **par(mfrow=c(nblig,nbcol))** avec nblig et nbcol donnant le nombre de fenêtres graphiques souhaitées verticalement et horizontalement.

5.2. Quelques fonctions graphiques

5.2.1. La fonction plot()

Nous commençons par un exemple, où nous allons reprendre le jeu de données des iris de Fisher. La fonction **plot()** sert à afficher le graphe croisant une variable numérique avec une autre. Ses paramètres indispensables sont x, correspondant à l'axe horizontal et y pour l'axe vertical. D'autres paramètres optionnels permettent de gérer le figuré (*pch*=), les labels des axes (*xax*= et *yax*=), demettre un titre (*main*=). Pour une liste complète des options, ***?plot***. Ici, la variable alphanumérique *Species* permet de différencier les 3 variétés d'iris constituant le tableau et est utilisée comme variable illustrative et la fonction secondaire ***legend()*** permet d'ajouter une légende.

```
# Chargement des données au départ de la librairie interne 'base'
data(iris)

# Affichage de l'aide concernant le jeu de données 'iris'
help(iris)

# Résumé du tableau de données iris (statistiques descriptives)
summary(iris)

# Nuage de points avec en abscisse la variable Sepal_Length,
# en ordonnée Petal.Length, des symboles différents
# selon l'espèce avec l’argument pch=
# un titre général avec main=
# et des titres d'axes personnalisés avec xlab= et ylab=
plot( x=iris$Sepal.Length,
    y=iris$Petal.Length,
    pch=(1:3)[iris$Species],
    main="Graphique de base - Iris de Fisher",
    xlab="Longueur des sépales(mm)", ylab="Longueur des pétales (mm)")

# Ajout d'une légende au graphique
help(legend)
legend(7, 3, c("I.setosa", "I.versicolor", "I.virginica"),pch=1:3)
```


Graphique de base - Iris de Fisher
Longueur des pétales (mm)
1
2
3
4
5
6
7
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
Longueur des sépales(mm)
I.setosa
I.versicolor
I.virginica

Une autre syntaxe possible est une formule de la forme y~x :

```
plot(iris$Petal.Length~iris$Sepal.Length)
```

ou

```
plot(Petal.Length~Sepal.Length, data=iris)
```

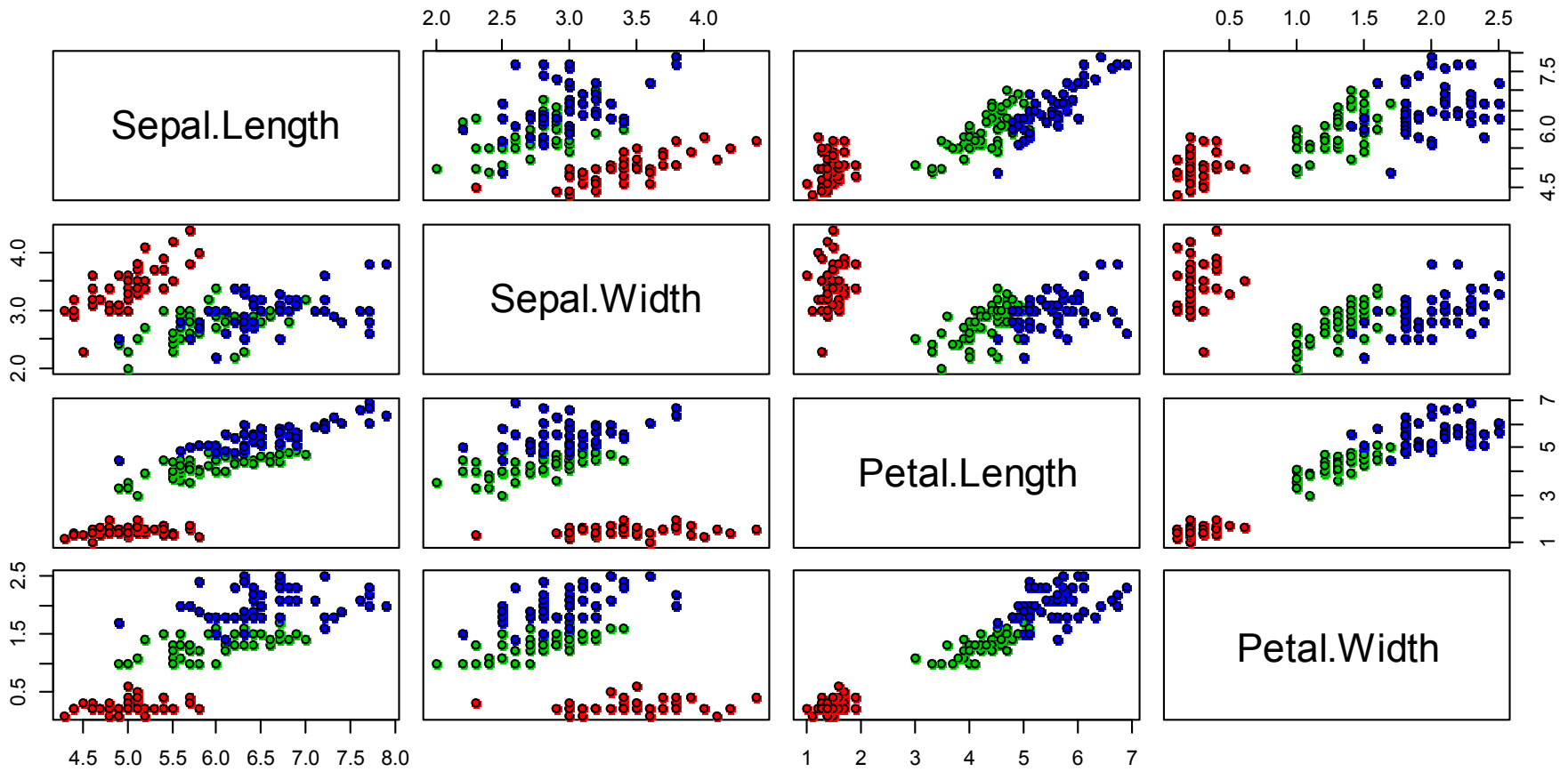
Les avantages de cette dernière formulation sont (i) que les labels des axes donnent directement le nom de la colonne (sans iris$ devant) et (ii) que cette syntaxe est identique à celle des fonctions permettant de faire des modèles comme la fonction lm() vue plus haut :

```
lm(Petal.Length~Sepal.Length, data=iris)
```

5.2.2. La fonction pairs()

La fonction **pairs()** sert à explorer les données en affichant une matrice de graphes croisant toutes les variables numériques deux à deux et de réaliser ainsi un **Draftsman display**. Ici on l'applique aux quatre variables numériques du data.frame Iris. On utilise le même figuré pour les trois espèces (pch=21 qui correspond à un cercle dont on peut colorer le contour avec col= et l'intérieur avec bg=). Ici la couleur intérieure est choisie parmi trois couleurs dont les noms sont prédéfinis dans R et associée à une des 3 variétés selon leur ordre alphabétique : rouge pour setosa, vert pour versicolor et bleu pour virginica.

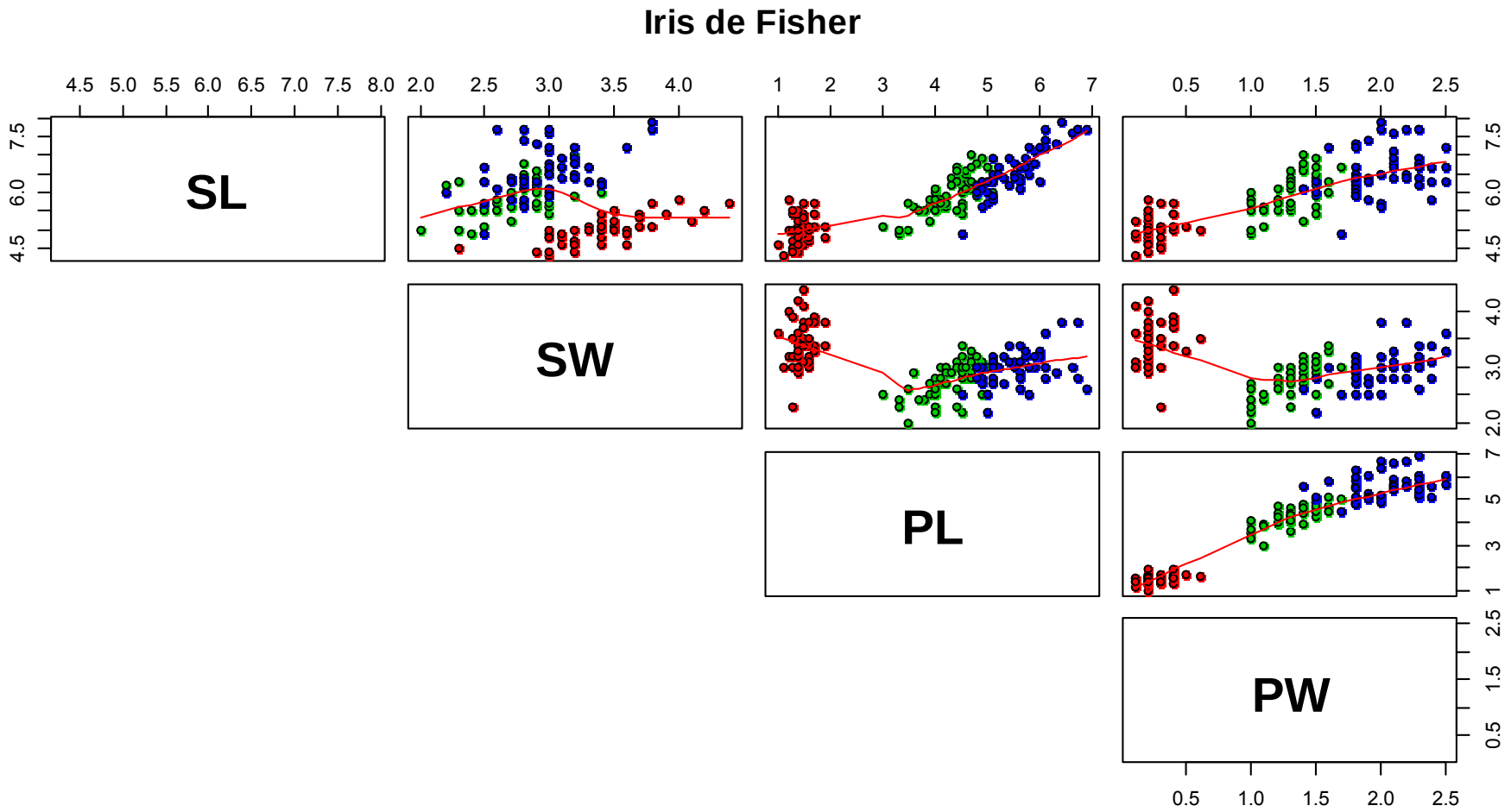
```
pairs(iris[1:4], main = "Iris de Fisher", pch = 21, bg = c("red", "green3",
"blue")[iris$Species], col="black")
```


Iris de Fisher
Sepal.Length
Sepal.Width
Petal.Length
Petal.Width
2.0 2.5 3.0 3.5 4.0
0.5 1.0 1.5 2.0 2.5
4.5 6.0 7.5
2.0 3.0 4.0
1 3 5 7
0.5 1.5 2.5
4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0
1 2 3 4 5 6 7

Pour améliorer le graphique précédent :

- on supprime les graphes situés en dessous de la diagonale avec *lower.panel=NULL*
- on ajoute une courbe lissée sur ceux situés au dessus de la diagonale avec *upper.panel=panel.smooth*
- on modifie les labels apparaissant sur la diagonale avec *labels=*
- *font.labels=2* définit des caractères gras pour ces labels (1=normal, 3=italique…)
- *cex.labels*=2.5 définit des caractères 2.5 fois plus grand que la normale

```
pairs(iris[1:4], main = "Iris de Fisher", pch = 21,
    bg = c("red", "green3", "blue")[iris$Species],
    lower.panel=NULL,
    upper.panel=panel.smooth,
    labels=c("SL","SW","PL","PW"),
    font.labels=2,
    cex.labels=2.5)
```

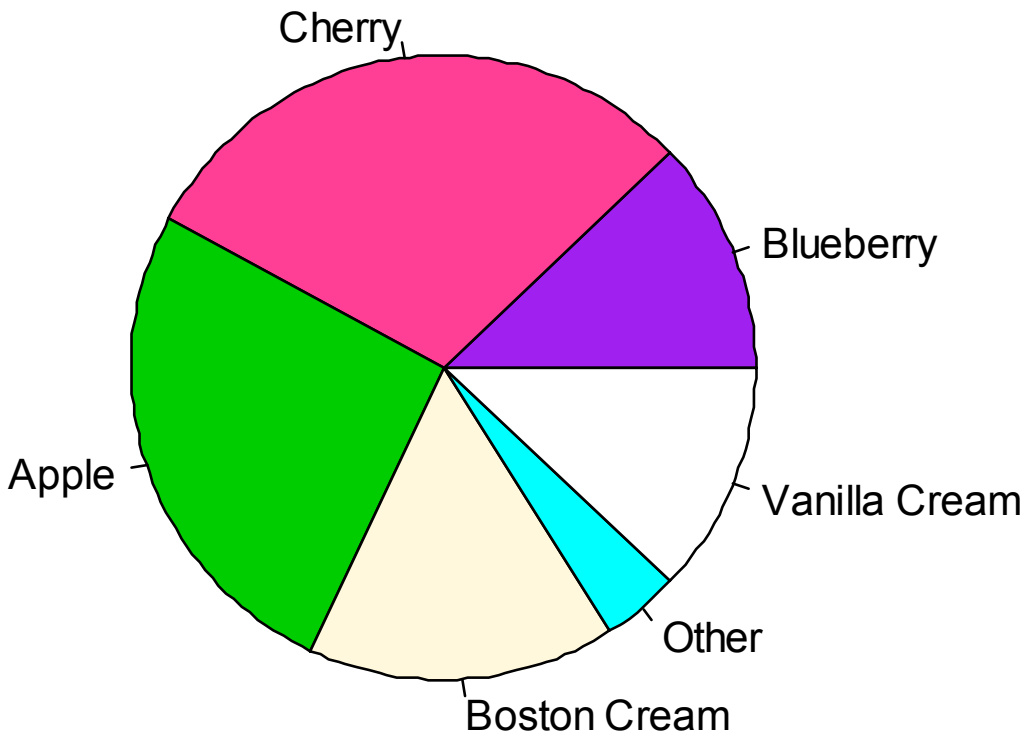


Nous verrons en fin de ce document comment améliorer encore ce Draftsman display en ajoutant le coefficient de corrélation entre les variables deux à deux.

5.2.3. La fonction pie()

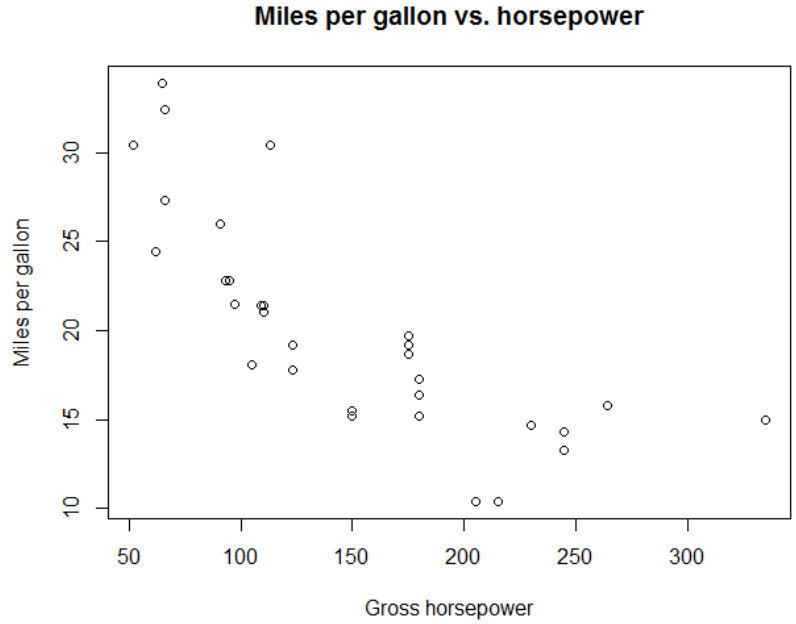
La fonction **pie()** permet de tracer un diagramme en camembert (pie chart) pour représenter une série de valeurs représentant des proportions. Statistiquement, ce graphique est trompeur car l'oeil humain n'est pas performant pour juger des aires relatives et il est préférable de réaliser plutôt un diagramme en bâton sur ce type de données (voir plus bas la fonction barplot). Voici un exemple de ce type de graphe à partir d'un petit jeu de données simulé correspondant aux proportions des ventes de différents parfums de crème glacée.

```
# Génération du vecteur de données et attribution de noms avec names()
pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
names(pie.sales) <- c("Blueberry", "Cherry",
    "Apple", "Boston Cream", "Other", "Vanilla Cream")
pie.sales
    Blueberry        Cherry         Apple  Boston Cream         Other Vanilla Cream
         0.12          0.30          0.26          0.16          0.04          0.12
pie(pie.sales, main="Ice cream",
    col = c("purple", "violetred1", "green3", "cornsilk", "cyan", "white"))
```


Ice cream
Cherry
Blueberry
Apple
Vanilla Cream
Other
Boston Cream

Exercice 5

- Chargez le jeu de données de R mtcars et affichez son descriptif
- Faites un draftsman display. Modifiez la taille et la police des labels. N'affichez que les graphes au-dessus ou au-dessous de la diagonale. N'afficher le graphe que pour les 7 premières colonnes
- Puis faites un scatterplot de la variable mpg en function de la variable hp, avec un titre et des labels personnalisés sur les axes.

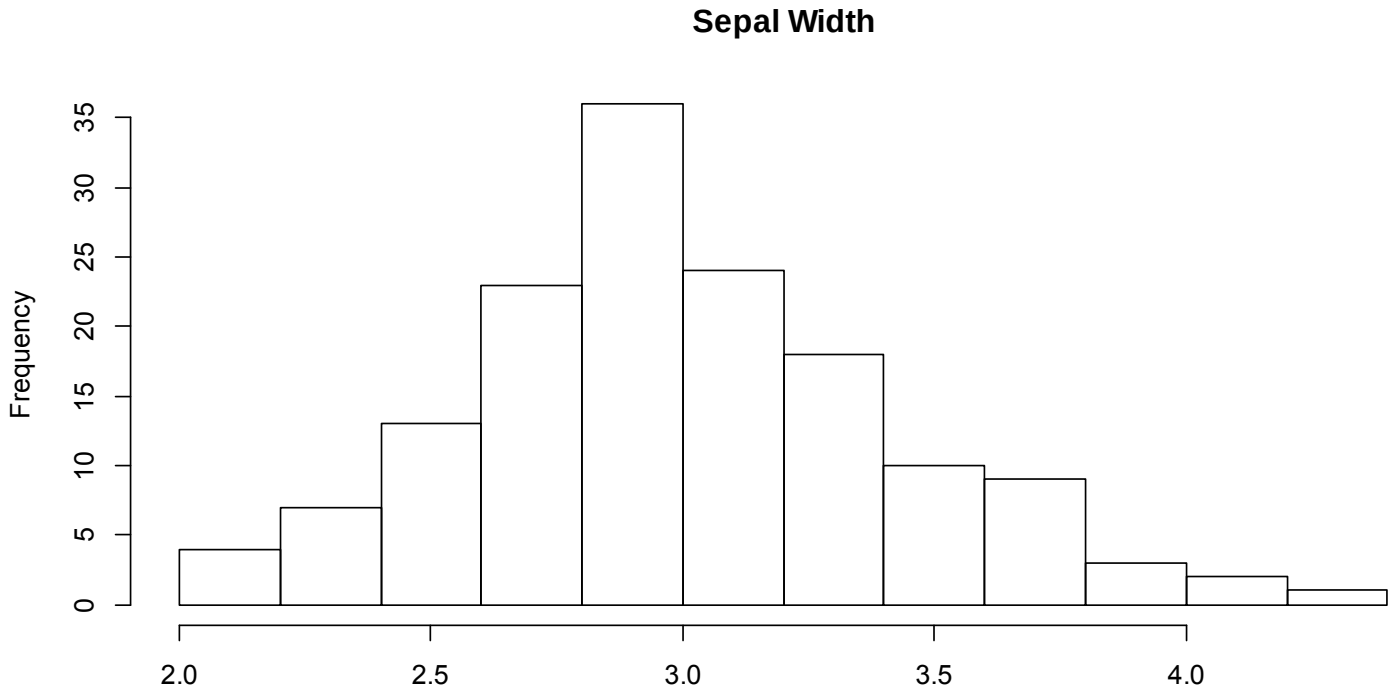


5.2.4. La fonction hist()

La fonction **hist()** permet de tracer un histogramme pour une variable numérique. L'histogramme permet d'identifier le (ou les) **mode**(s) de la distribution, la (ou les) valeurs observées dont la fréquence (nombre d'observations) est maximale. Une distribution **unimodale** possède un seul maximum de fréquence (courbe en cloche). Une distribution **plurimodale** comporte plusieurs maximums de fréquence.
Les intervalles sont égaux par défaut mais peuvent être définis de différentes manières. Si on ne le précise pas, le nombre optimal d'intervalles est calculé par R selon l'algorithme de Sturges, ici des intervalles de 0.5. Ce nombre peut être imposé par l'utilisateur grâce à l'option *breaks*=. On peut définir un titre (*main*=), un label pour l'axe des abscisses (*xlab*=)

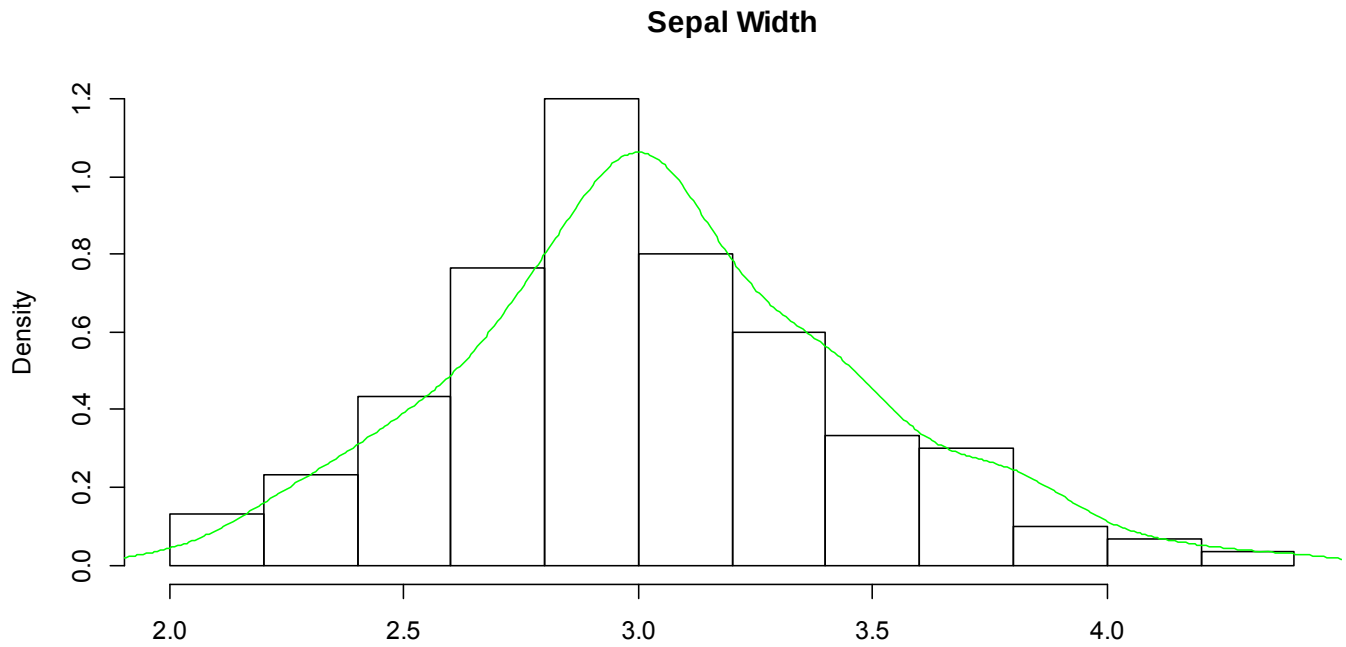
On applique ici la fonction hist() à la colonne Sepal.Width du data.frame iris.

```
hist(iris$Sepal.Width, main="Sepal Width", xlab="")
```



A la place des fréquences, on peut afficher des **densités** avec l'option *freq=F* et ajouter par-dessus la fonction de densité *density()* avec la fonction *lines()*

```
hist(iris$Sepal.Width, main="Sepal Width", xlab="", freq=F)
lines(density(iris$Sepal.Width), col="green")
```



Les résultats de la fonction hist() apparaissent par défaut sur une fenêtre graphique, mais peuvent aussi être placés dans une liste R :

```
hist1 <- hist(iris$Sepal.Width)
hist1
```

`$breaks` (les bornes des intervalles)

```
 [1] 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4
```

`$counts` (le nombre de valeurs par intervalle)

```
 [1] 4  7 13 23 36 24 18 10  9  3  2  1
```

`$density` (valeurs de la fonction de densité)

```
 [1] 0.13333333 0.23333333 0.43333333 0.76666667 1.20000000 0.80000000 0.60000000
0.33333333 0.30000000
[10] 0.10000000 0.06666667 0.03333333
```

`$mids` (valeur centrale des intervalles)

```
 [1] 2.1 2.3 2.5 2.7 2.9 3.1 3.3 3.5 3.7 3.9 4.1 4.3
```

`$xname` (nom de la variable x, tel qu'il est affiché sur le graphe)

```
[1] "iris$Sepal.Width"
```

`$equidist` (vrai si les intervalles sont équidistants)

```
[1] TRUE
```

5.2.5. La fonction barplot()

Cette function génère des diagrammes en bâtons (horizontaux ou verticaux) sur des variables **discrètes**.

Nous allons appliquer cette fonction sur le jeu de données **VADeaths** de R (taux de mortalité en Virginie en 1940 – en lignes des classes d'âge, en colonnes des catégories combinant homme/femme et rural/urbain) : *The death rates are measured per 1000 population per year. They are cross-classified by age group (rows) and population group (columns). The age groups are: 50–54, 55–59, 60–64, 65–69, 70–74 and the population groups are Rural/Male, Rural/Female, Urban/Male and Urban/Female.*

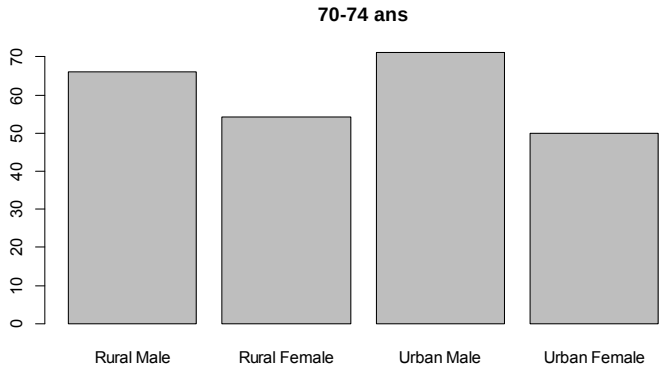
```
data(VADeaths)
dim(VADeaths)
[1] 5 4
VADeaths
      Rural Male Rural Female Urban Male Urban Female
50-54       11.7          8.7       15.4          8.4
55-59       18.1         11.7       24.3         13.6
60-64       26.9         20.3       37.0         19.3
65-69       41.0         30.9       54.6         35.1
70-74       66.0         54.3       71.1         50.0
```

NB : Ce jeu de données est de type matrice (*matrix*) et non tableau de données (*data.frame*). La fonction **barplot()** ne peut s'appliquer qu'à des objets de type **vecteur ou matrice** :

```
is.matrix(VADeaths)
[1] TRUE
is.data.frame(VADeaths)
[1] FALSE
```

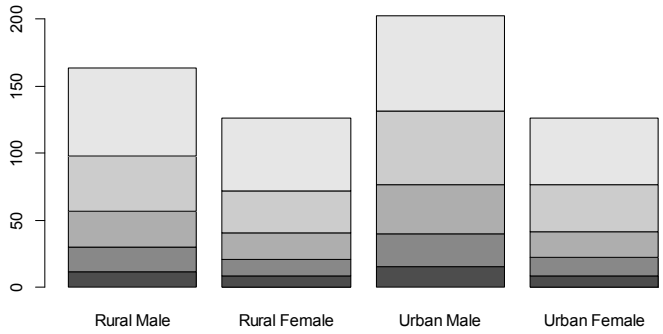
Extraire la dernière ligne de ce jeu de données, correspondant à la classe d'âge 70-74 ans, dans un vecteur indépendant et appliquer la fonction barplot à ce vecteur pour représenter la mortalité dans cette classe d'âge :

```
VADeaths.7074 <- VADeaths[5,]
barplot(VADeaths.7074, main="70-74 ans")
```



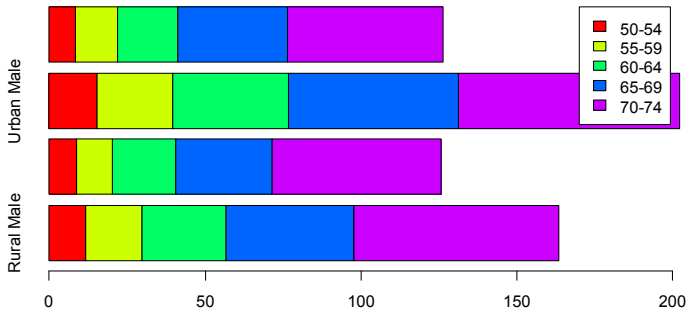
Lorsque l'on applique la fonction **barplot()** à la matrice entière, chaque colonne est représentée par une barre, où les 5 classes d'âge sont distinguées par des grisés différents. Par défaut, la légende n'est pas affichée :

```
barplot(VADeaths)
```



Les barres peuvent être tracées horizontalement (*horiz=TRUE*). On peut aussi modifier la couleur (*col=*) et afficher la légende correspondante (*legend=*) :

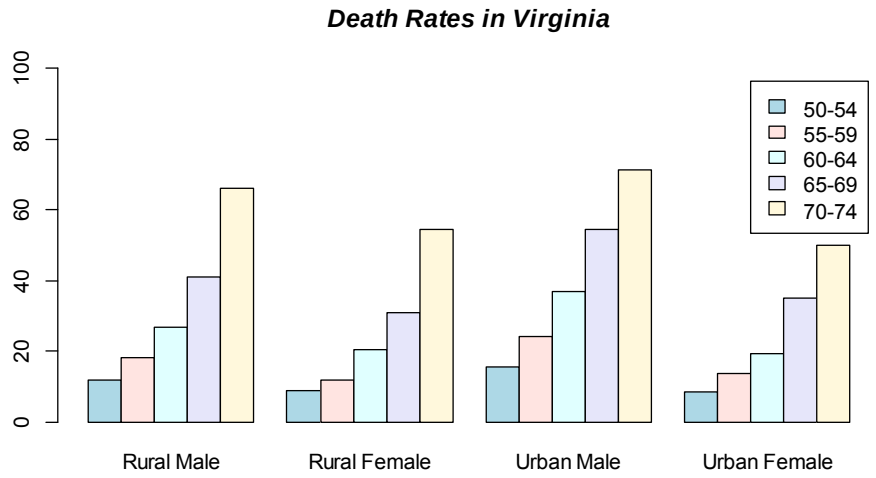
```
barplot(VADeaths, horiz=T, col=rainbow(5),legend=rownames(VADeaths))
```



Notez ici l'utilisation de **col=rainbow(n)** qui choisit automatiquement n couleurs sur une palette pré-définie des couleurs de l'arc-en-ciel. D'autres palettes existent. Taper **?colors** pour plus d'information.

Une autre façon de distinguer les classes d'âge est d'utiliser l'option *beside=TRUE* qui juxtapose les barres au lieu de les empiler. On note au passage une autre façon de spécifier les couleurs par leur nom. Taper *colors()* dans la console pour avoir la liste des noms de couleur sous R.

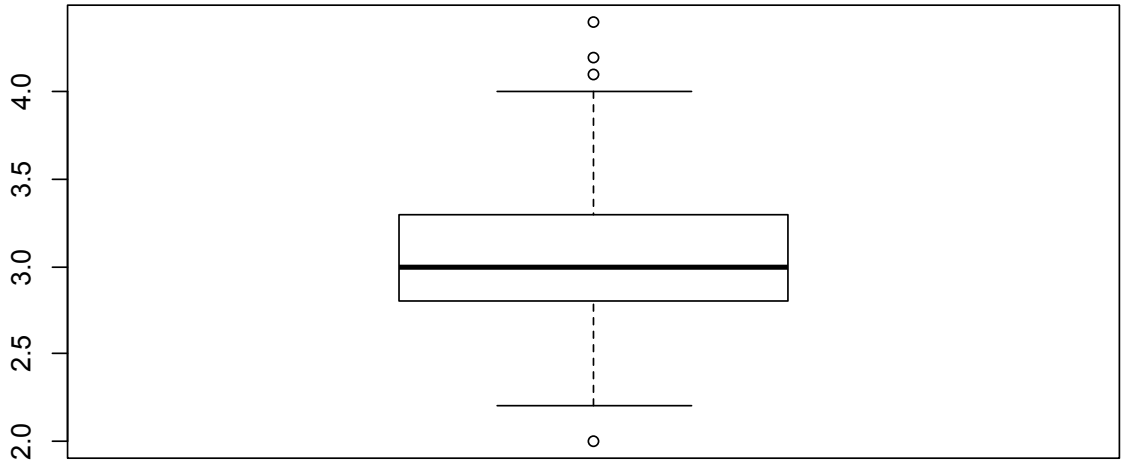
```
barplot(VADeaths, beside = TRUE,
        col = c("lightblue", "mistyrose", "lightcyan",
                "lavender", "cornsilk"),
        legend = rownames(VADeaths), ylim = c(0, 100))
title(main = "Death Rates in Virginia", font.main = 4)
```



5.2.6. La fonction boxplot()

La « boite à moustache » ou *box-and-whisker plot* (fonction **boxplot()** de R) permet de représenter graphiquement la distribution d'un vecteur de données. On applique la fonction aux données iris$Sepal.Width.

```
boxplot(iris$Sepal.Width)
```

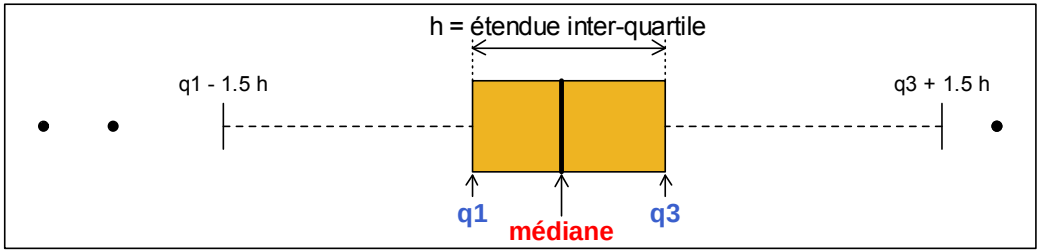


En désactivant l'affichage du graphique (*plot=FALSE*), on obtient dans la console les éléments de la liste générée par la fonction boxplot(). On peut aussi placer le résultat dans un objet R.

```
boxplot(iris$Sepal.Width, plot=F)
$stats
     [,1]
[1,]  2.2
[2,]  2.8
[3,]  3.0
[4,]  3.3
[5,]  4.0

$n
[1] 150

$conf
         [,1]
[1,] 2.935497
[2,] 3.064503

$out
[1] 4.4 4.1 4.2 2.0
```

```
$group
[1] 1 1 1 1

$names
[1] "1"
```

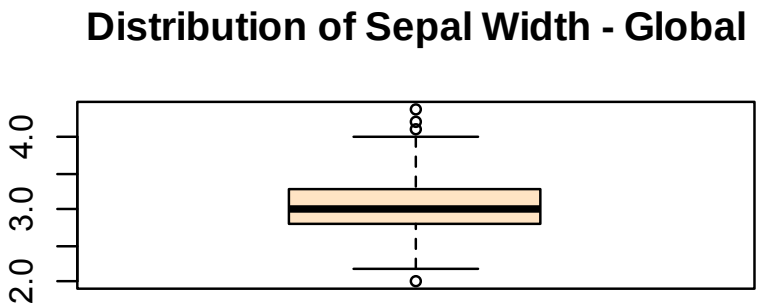
La **médiane** (3) est donnée par le troisième élément de *$stats*. Les quartiles Q1 (2.8) et Q3 (3.3) sont donnés respectivement par le second et le quatrième élément de *$stats*. **L'étendue inter-quartile,** défini comme l'intervalle entre Q1 et Q3 contient 50% des observations. Il est visualisé par la **boîte**. Les **moustaches** (traits prolongeant la boîte en haut et en bas ici jusqu'à à 2.2 et 4) s'étendent jusqu'au point le plus extrême dont la distance à la bordure de la boîte n'excède pas 1.5 fois l'intervalle interquartiles. Les points situés au-delà des moustaches sont des **outliers** donnés par *$out*. Il est judicieux de vérifier dans les données s'il ne s'agit pas de points aberrants et si c'est le cas de les corriger ou les supprimer des données. L'option *outline=F* permet de ne pas afficher les outliers.

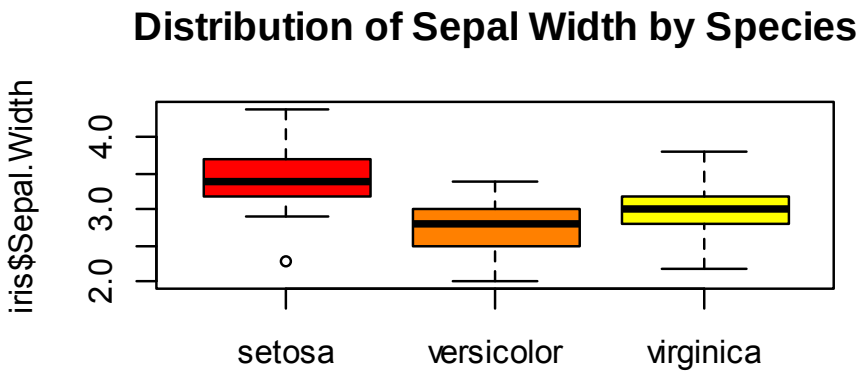


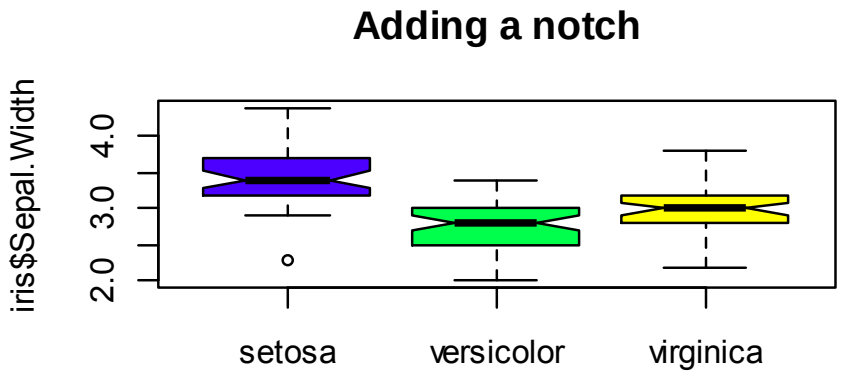
On peut gérer la couleur avec *col*=, ajouter une encoche (notch) autour de la médiane avec *notch*=*T* ou présenter les boxplots horizontalement avec *horizontal*=*T*. Les valeurs peuvent être groupées selon une variable discrète en utilisant le ~. Si les encoches de deux boites ne se superposent pas, on peut en conclure que les deux médianes sont différentes.

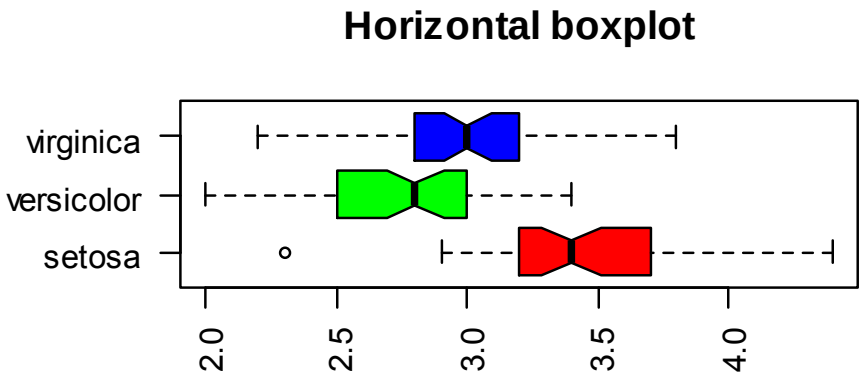
L'exemple ci-dessous est appliqué à la colonne Sepal.Width du data.frame iris :

```
par(mfrow=c(2,2))
boxplot(iris$Sepal.Width, main="Distribution of Sepal Width - Global",
col="bisque")
boxplot(iris$Sepal.Width~iris$Species, main="Distribution of Sepal Width by
Species", col=heat.colors(3), xlab="")
boxplot(iris$Sepal.Width~iris$Species, main="Adding a notch", col=topo.colors(3),
notch=T, xlab="")
boxplot(iris$Sepal.Width~iris$Species, main="Horizontal boxplot", col=rainbow(3),
notch=T, horizontal=T, xlab="", ylab="", las=2)
```


Distribution of Sepal Width - Global
2.0
3.0
4.0


Distribution of Sepal Width by Species
iris$Sepal.Width
2.0
3.0
4.0
setosa
versicolor
virginica


Adding a notch
iris$Sepal.Width
2.0
3.0
4.0
setosa
versicolor
virginica

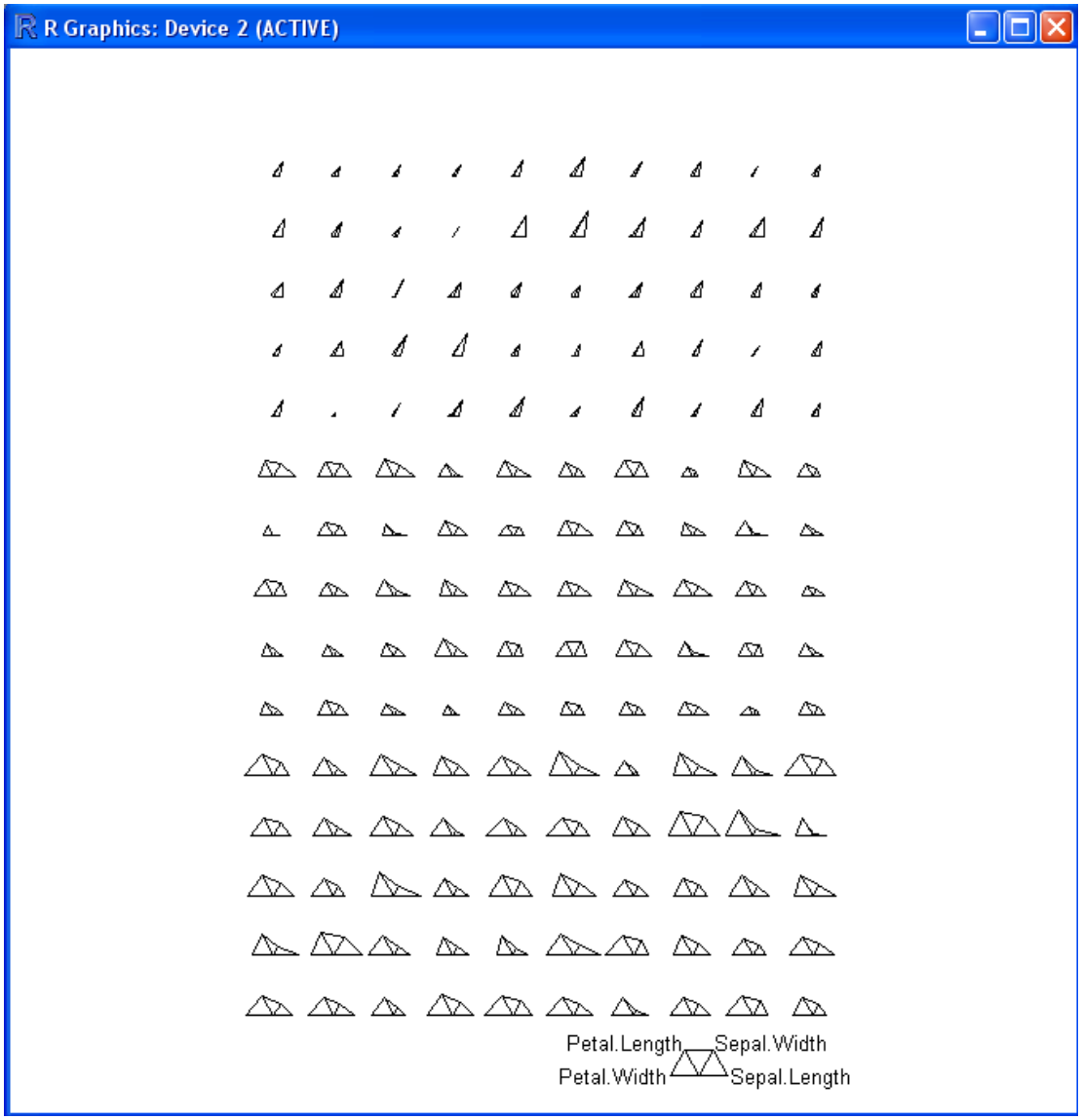

Horizontal boxplot
virginica
versicolor
setosa
2.0
2.5
3.0
3.5
4.0

5.2.7. La fonction stars() : tracé de radarplot

Un diagramme en radar permet de représenter en 2 dimensions des données multivariées (au moins 3 variables) sur des axes partant du même point. Ce type de graphique est utile uniquement si les variables sont dans des unités de mesures comparables (par exemple pour représenter les notes d'un élève dans toutes les matières, toutes les notes étant entre 0 et 20). Sous R on utilise la fonction **stars().**

Le premier exemple porte sur les données iris. Les 150 individus sont représentés selon leurs valeurs pour les 4 variables quantitatives (légende en bas du graphe). Les 50 premiers correspondent à la variété *setosa* que nous avons déjà identifiée plus haut comme présentant des longueurs inférieures aux deux autres variétés.

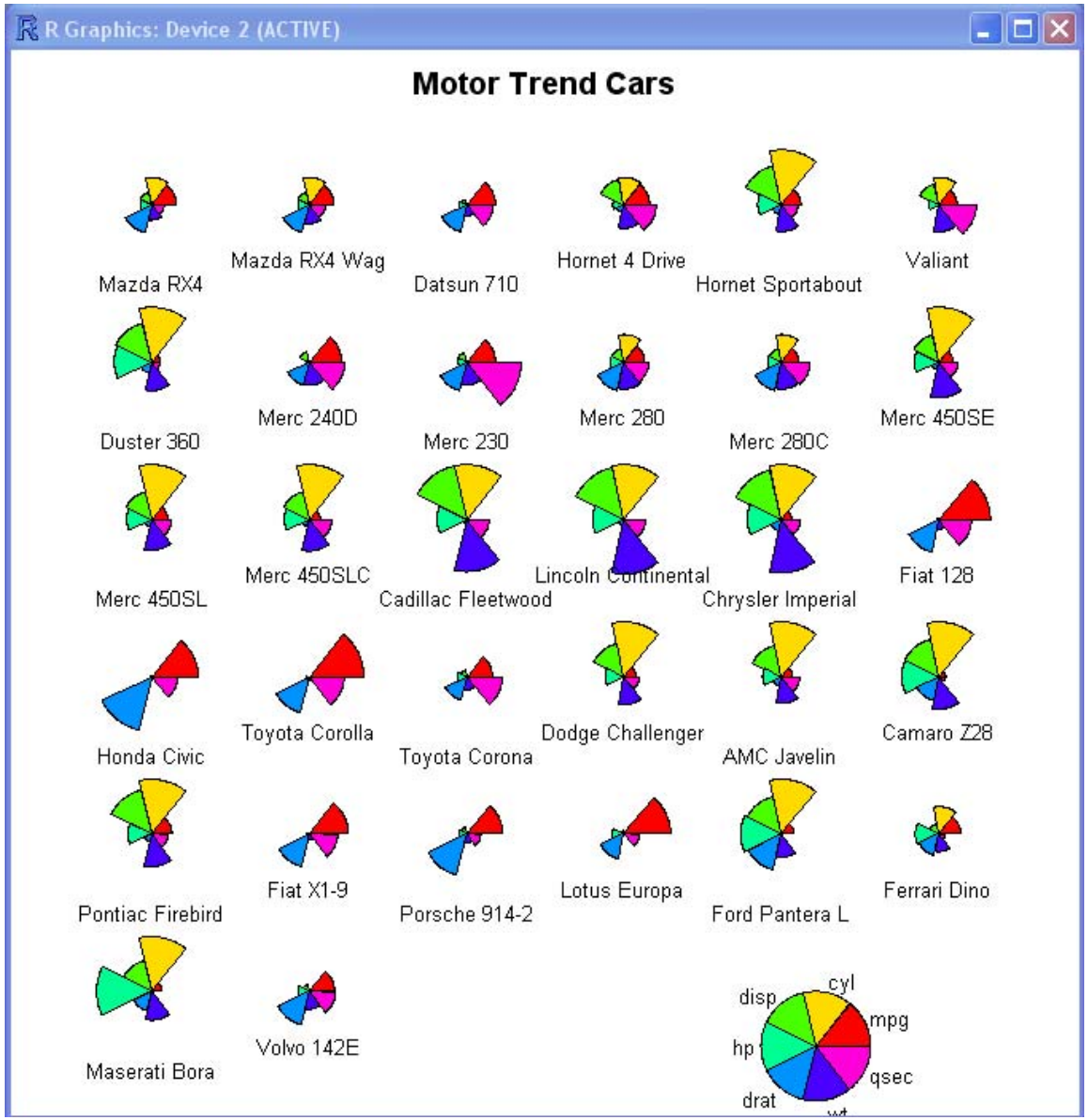
```
# Premier exemple sur les données des iris
# Les 4 variables quantitatives sont représentées pour chaque individu
# ncol=10 pour afficher les individus sur 10 colonnes
# full=FALSE pour représenter les variables sur un demi-cercle
# au lieu d'un cercle complet (full=TRUE qui est l'option par défaut)
# key.loc donne la position de la légende en (x,y)

stars(iris[,1 :4], key.loc = c(17,0), full=FALSE, ncol=10)
```


R Graphics: Device 2 (ACTIVE)
Petal.Length
Sepal.Width
Petal.Width
Sepal.Length

Le second exemple est un jeu de données reprenant les caractéristiques de 32 modèles de voitures des années 1973-1974. Les 7 premières variables sont les suivantes :

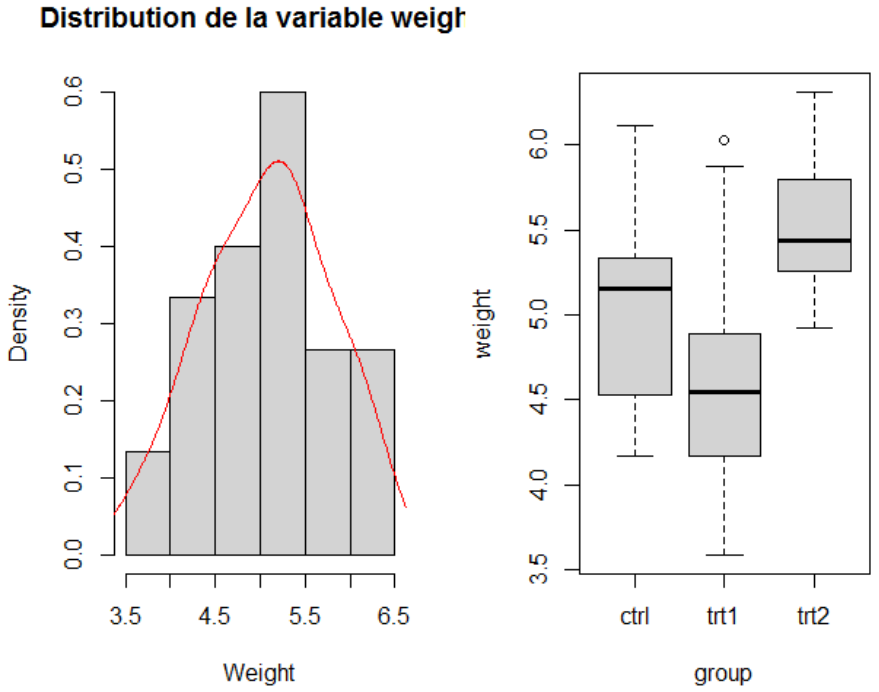
[, 1] mpg Miles/(US) gallon
[, 2] cyl Number of cylinders
[, 3] disp Displacement (cu.in.)
[, 4] hp Gross horsepower
[, 5] drat Rear axle ratio
[, 6] wt Weight (lb/1000)
[, 7] qsec 1/4 mile time

```
# Second exemple sur les données mtcars
# L'instruction palette() choisit 7 couleurs sur la palette rainbow.
# Les 7 variables sont représentées pour chaque modèle de voiture
# key.loc donne la position de la légende en (x,y)
# draw.segments=TRUE permet une représentation en « parts de camembert »
# de longueur proportionnelle à la variable. Essayer en mettant à FALSE

palette(rainbow(7))
stars(mtcars[, 1:7], len = 0.8, key.loc = c(12, 1.5), main = "Motor Trend Cars",
draw.segments = TRUE)
```


R Graphics: Device 2 (ACTIVE)
Motor Trend Cars
Mazda RX4
Mazda RX4 Wag
Datsun 710
Hornet 4 Drive
Hornet Sportabout
Valiant
Duster 360
Merc 240D
Merc 230
Merc 280
Merc 280C
Merc 450SE
Merc 450SL
Merc 450SLC
Cadillac Fleetwood
Lincoln Continental
Chrysler Imperial
Fiat 128
Honda Civic
Toyota Corolla
Toyota Corona
Dodge Challenger
AMC Javelin
Camaro Z28
Pontiac Firebird
Fiat X1-9
Porsche 914-2
Lotus Europa
Ford Pantera L
Ferrari Dino
Maserati Bora
Volvo 142E
disp
cyl
mpg
hp
qsec
drat

Exercice 6

- Reprenez le jeu de données PlantGrowth et faites l'histogramme de la colonne weight en superposant la courbe de densité.
- Toujours sur PlantGrowth, représentez un boxplot de weight pour chacun des 3 groupes. Discutez de ce résultat en le comparant avec le résultat du lm() de l'exercice 4.



5.2.8. Les fonctions de tracé de courbes de niveau

Plusieurs fonctions sous R permettent de tracer des courbes de niveau à partir d'une matrice de données contenant des données topographiques en grille régulière (données kriggées). Voir un exemple avec le jeu de données **volcano** qui contient des données topographiques sur le volcan Maunga Whau (New Zealand) selon une grille régulière de 10m par 10m.

```
data(volcano)
dim(volcano)
[1] 87 61
par(mfrow=c(1,3))
contour(volcano, main="contour : courbes de niveau")
image(volcano, main="image : représentation colorée")
persp(volcano, main="persp : représentation 3D")
```

contour : courbes de niveau

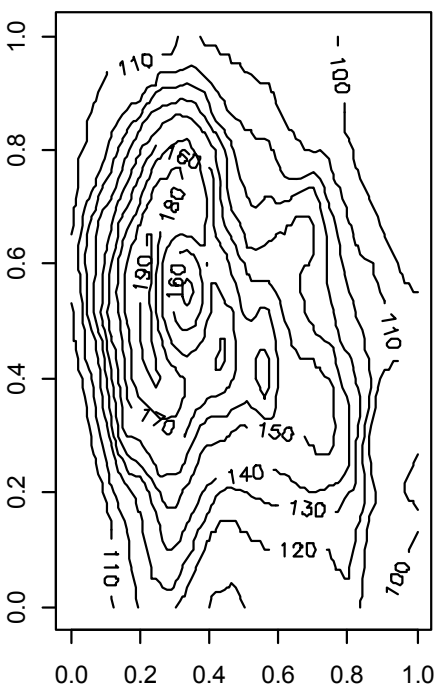
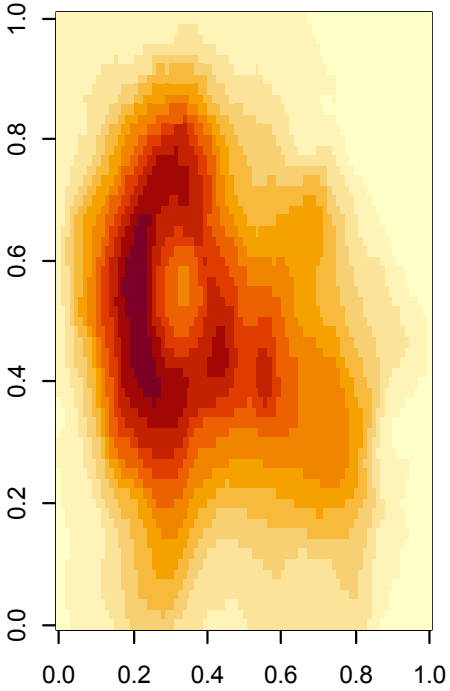
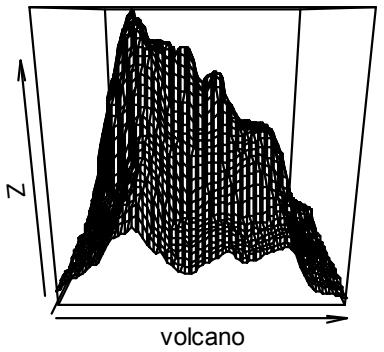


image : représentation colorée

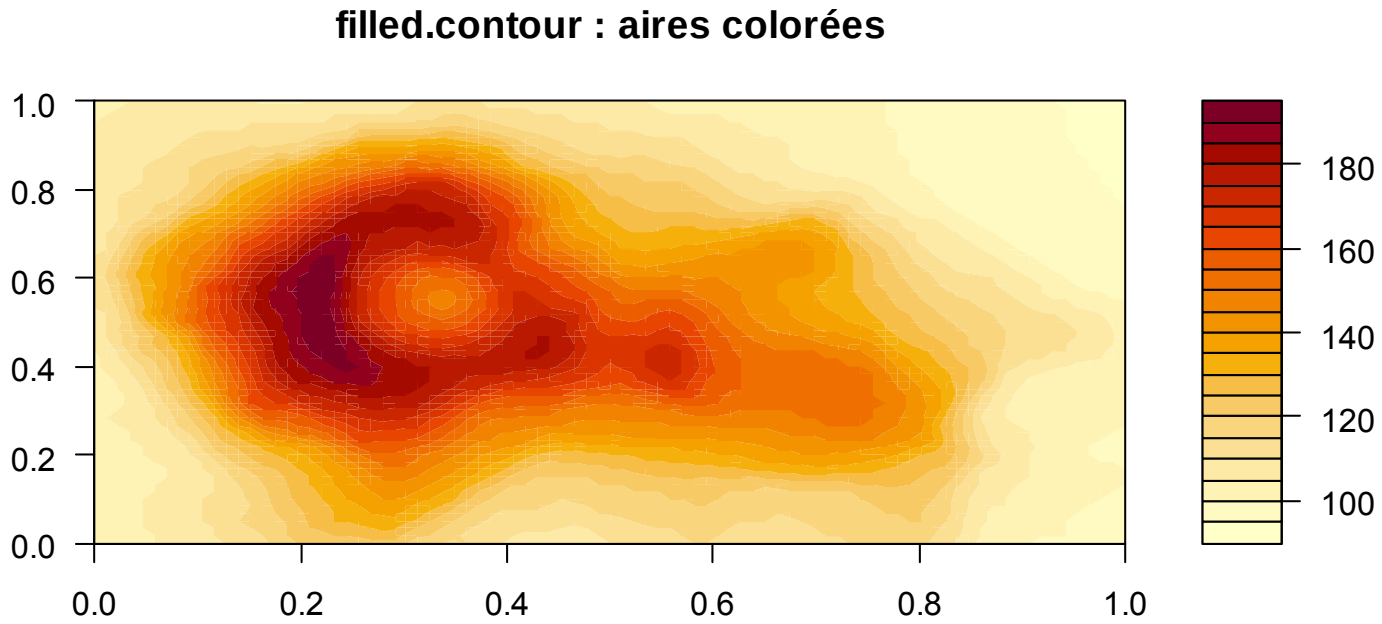


persp : représentation 3D



La fonction la plus aboutie pour ce type de données est **filled.contour()** qui affiche automatiquement une échelle de couleurs :

```
filled.contour(volcano, main="filled.contour : aires colorées")
```

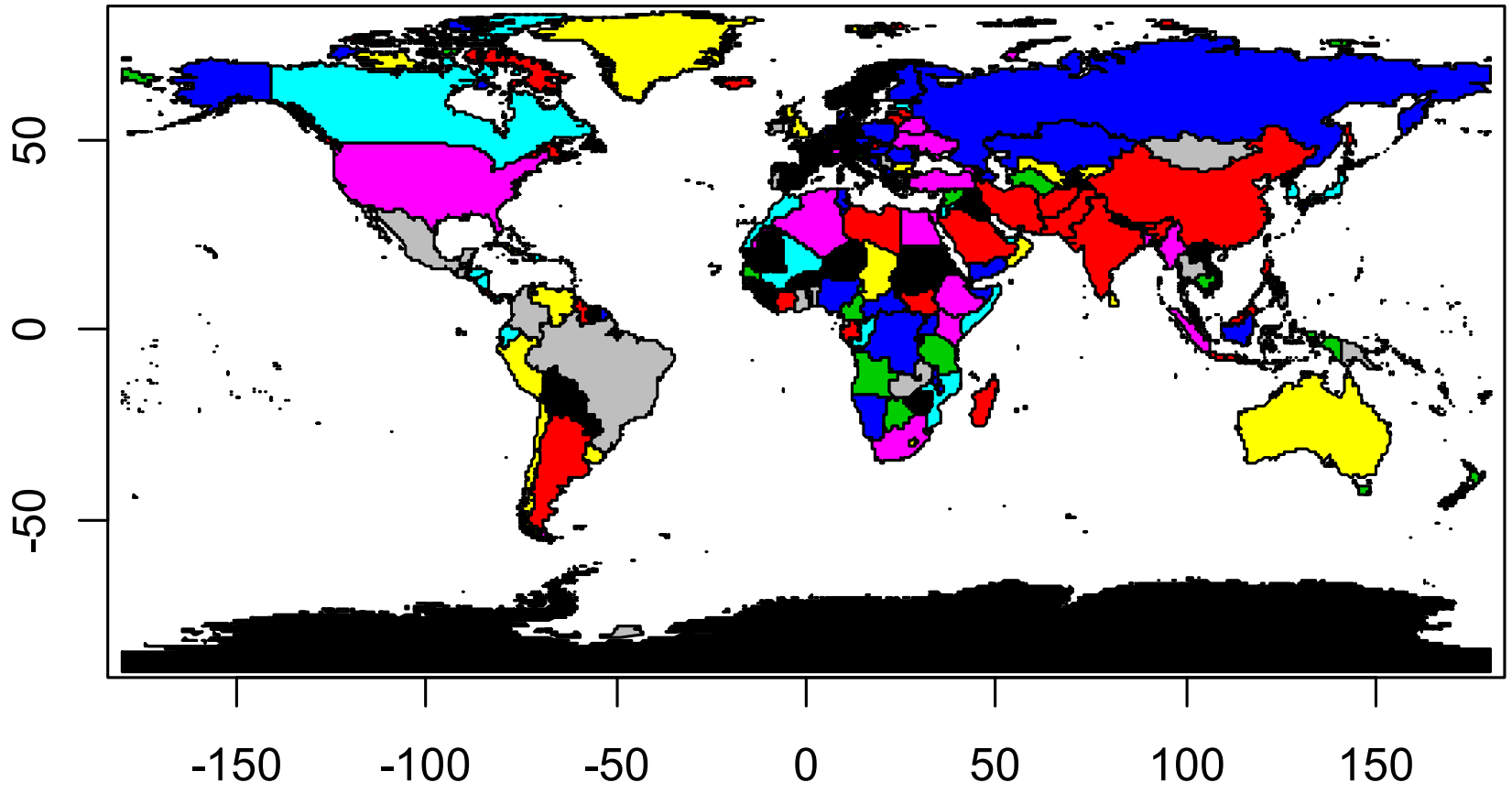


5.2.9. Les fonctions de cartographie

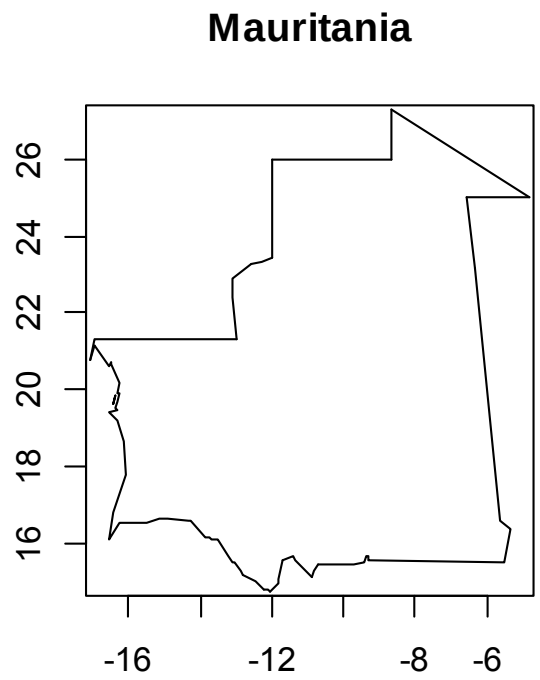
Il existe sous R des librairies spécifiques permettant de cartographier des données.

La librairie **maps** permet de récupérer certains fonds de carte, dont un fond de carte général du monde. La librairie **mapdata** contient des bases de données supplémentaires de cartes.

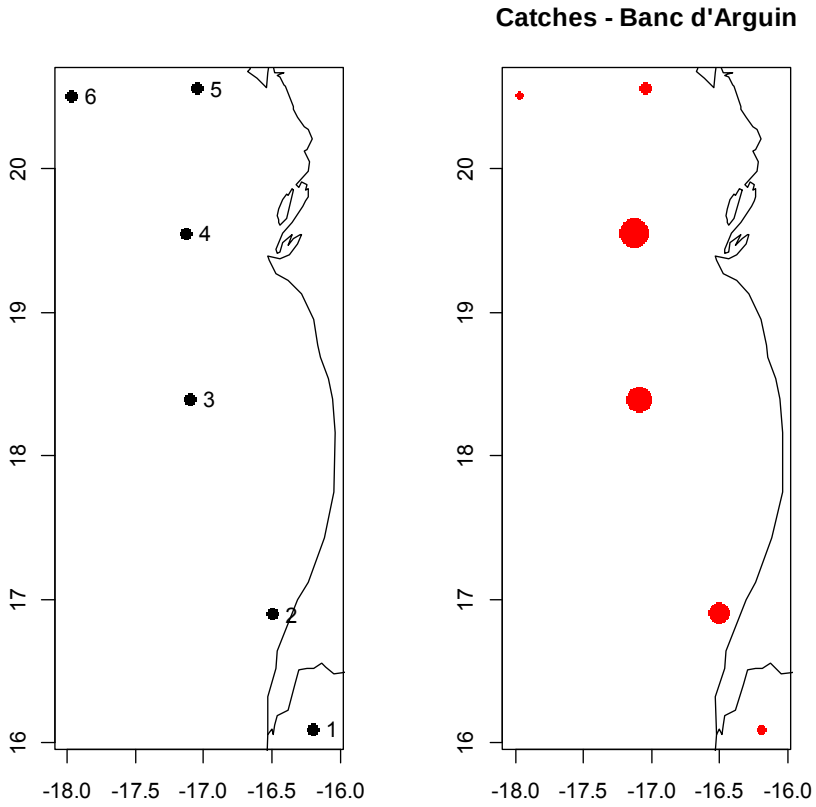
```
library(maps)
m <- map(wrap = c(-180,180), fill = TRUE, col = 1:10)
str(m)
List of 4
 $ x    : num [1:101107] -69.9 -69.9 -69.9 -70 -70.1 ...
 $ y    : num [1:101107] 12.5 12.4 12.4 12.5 12.5 ...
 $ range: num [1:4] -180 180 -89.9 83.6
 $ names: chr [1:1632] "Aruba" "Afghanistan" "Angola" "Angola:Cabinda" ...
 - attr(*, "class")= chr "map"
map(wrap = c(-180,180), fill = TRUE, col = 1:10)
map.axes()
```


50
0
-50
-150
-100
-50
0
50
100
150

```
library(mapdata)
map('worldHires', 'Mauritania')
map.axes()
title('Mauritania')
```



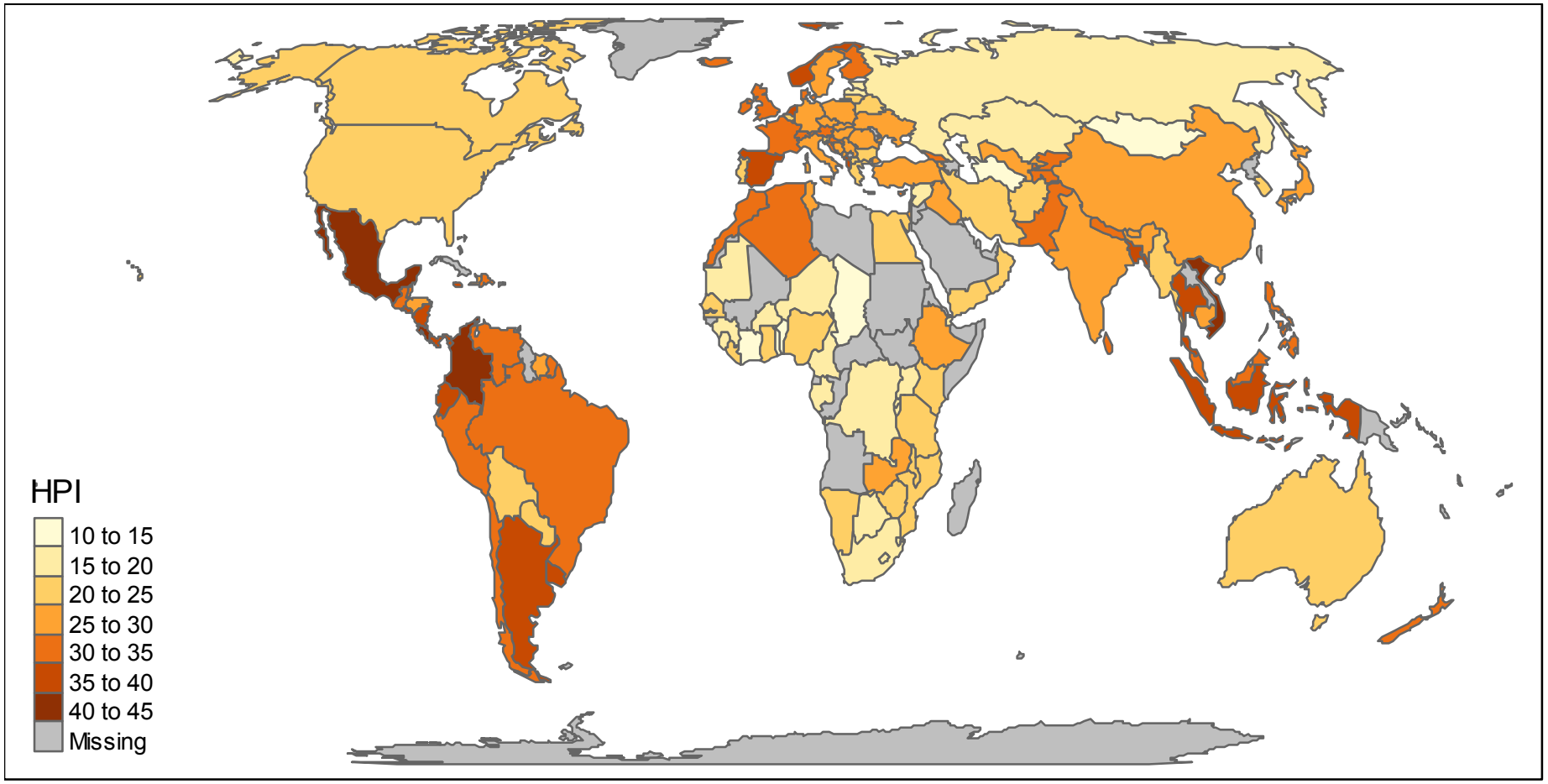
```
# Afficher des points sur un fond de carte
site <- seq(1,6,1)
lat <- c(16.1, 16.9, 18.4, 19.55, 20.56, 20.51)
lon <- c(-16.2, -16.5, -17.1, -17.13, -17.05, -17.97)
catch <- c(2000,4500,5400,6000,2500,1750)
dm <- cbind.data.frame(site,lat,lon,catch)

rlat <- c(min(dm$lat)-0.1, max(dm$lat)+0.1)
rlon <- c(min(dm$lon)-0.1, max(dm$lon)+0.2)

windows()
par(mfrow=c(1,2))
map('worldHires',xlim=rlon, ylim=rlat, col="black")
map.axes()
points(dm$lon,dm$lat,pch=16,cex=1.2)
text(dm$lon,dm$lat,dm$site, pos=4)
points(dm$lon,dm$lat,pch=16,cex=3*dm$catch/max(dm$catch), col="red")
title("Catches - Banc d'Arguin")

points(dm$lon,dm$lat,pch=16,cex=3*dm$catch/max(dm$catch), col="red")
```



Dans la librairie **maptools**, il y a un grand nombre de fonctions de cartographie qui permettent par exemple d'utiliser des fichiers issus de SIG (shapefiles).

D’autres librairies plus évoluées existent, comme **tmap** et **tmaptools** ou **ggmap** ou encore **leaflet** qui permettent de réaliser des cartes thématiques associées à des fonds d’images satellites, ou qui permet de faire des cartes interactives. Voir la documentation associée aux différentes librairies, celle de tmap est particulièrement bien faite. Ci-dessous un exemple de carte thématique du monde avec **tmap** :

```
library(tmap)
data("World")
tmap_mode("plot")
tm_shape(World) +
  tm_polygons("HPI")
```


HPI
10 to 15
15 to 20
20 to 25
25 to 30
30 to 35
35 to 40
40 to 45
Missing

5.3. Améliorer un graphe R

5.3.1. Les fonctions graphiques secondaires

Il existe diverses fonctions R permettant de compléter un graphique. Nous en citons seulement quelques-unes. Ces fonctions sont dites **secondaires** (low-level plotting commands) car elles ont une action sur un graphe déjà existant et les coordonnées spécifiées font référence au système de coordonnées de ce graphe :

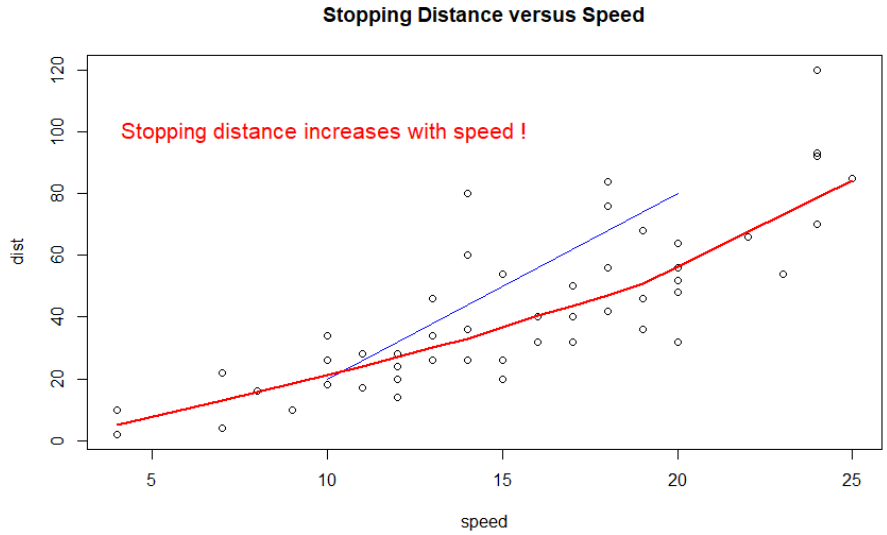
lines(x,y) : trace une ligne, qui peut être générée par une fonction mathématique, ou simplement un segment de ligne droite entre les deux points x et y.

text(x,y,labels) : affiche aux coordonnées x,y le texte spécifié dans labels. Par défaut, le texte est centré sur la position indiquée.

L'exemple ci-dessous illustre ces deux fonctions. On utilise le jeu de données **cars** de R qui contient 2 séries de valeurs : la distance de freinage (dist) et la vitesse (speed). On trace le graphe en (x,y) avec plot(), puis on superpose une ligne bleue entre deux positions sur le graphe, et une seconde ligne générée par la fonction de lissage **lowess(x, y)**. Le paramètre *lwd*= permet de modifier l'épaisseur de la ligne. Enfin on ajoute un texte.

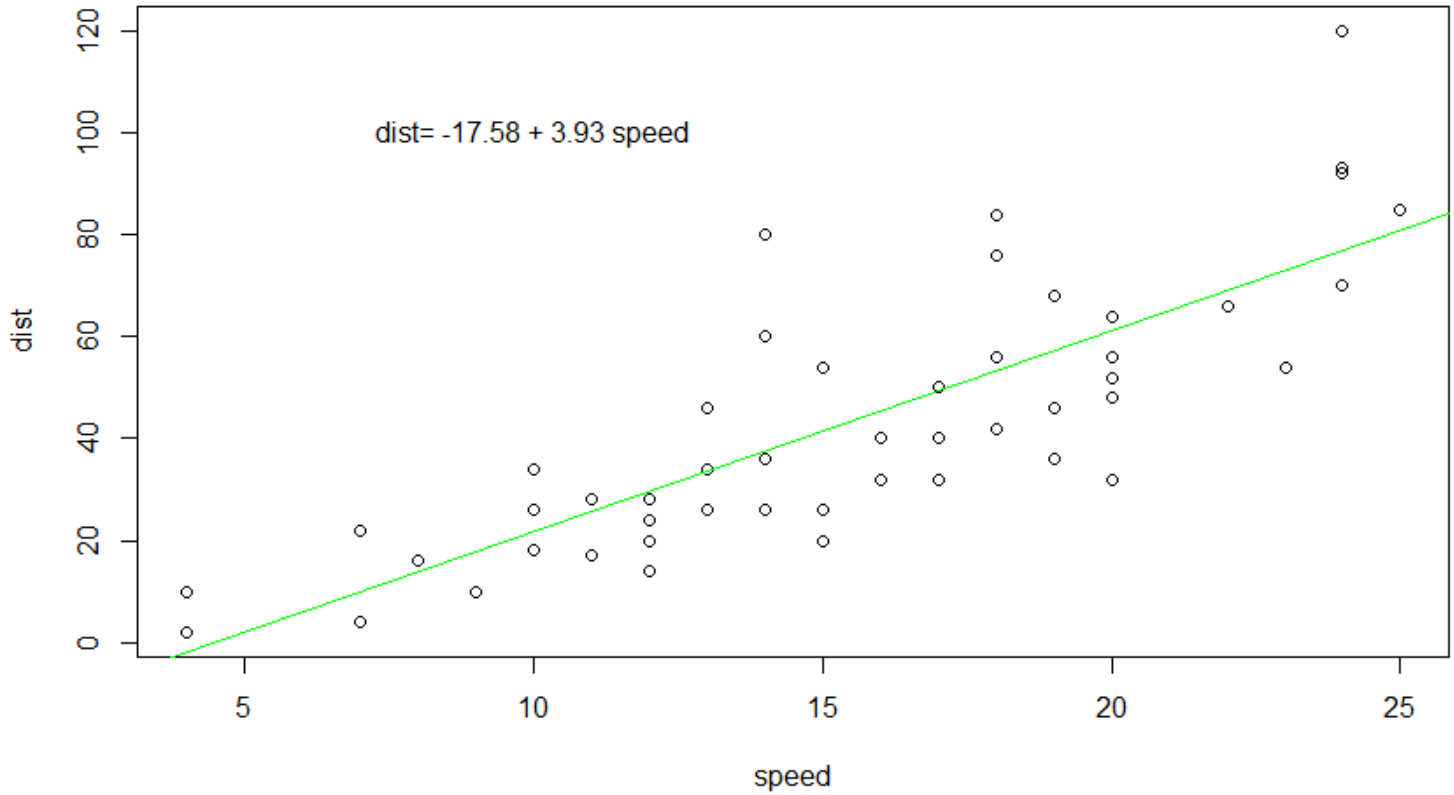
```
str(cars)
'data.frame':    50 obs. of  2 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17
```

```
plot(dist~speed, data=cars, main="Stopping Distance versus Speed")
lines(c(10,20), c(20,80), col="blue")
lines(lowess(cars$speed, cars$dist), col="red", lwd=2)
text(x=10,y=100,labels="Stopping distance increases with speed ! ", cex=1.3,
col="red")
```



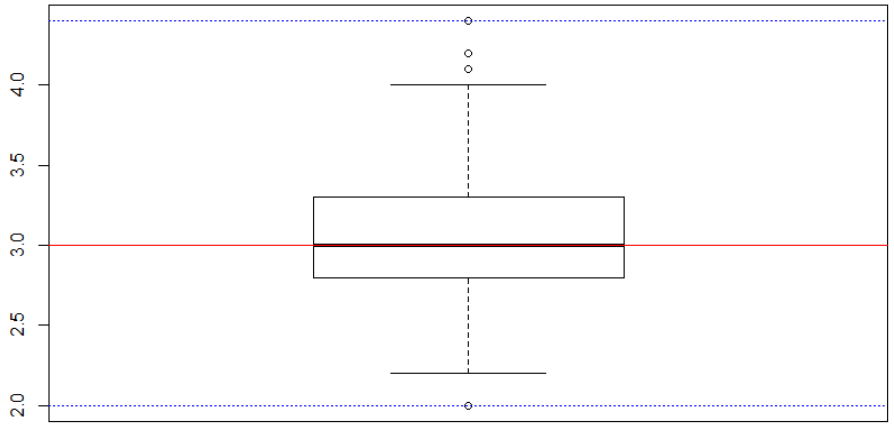
abline(a=, b=) : trace une droite de régression d'ordonnée à l'origine **a** et de pente **b** sur un nuage de points.
On reprend ci-dessous l'exemple précédent en traçant la droite de régression avec les coefficients a et b obtenus par la fonction *lm()*. On ajoute ensuite au graphe un texte qui donne l'équation de la droite, en utilisant la fonction *paste()* qui permet de concaténer des chaines de caractère. La fonction *round(x,2)* arrondit un nombre réel x à deux décimales.

```
m <- lm(dist~speed, data=cars)
coef(m)
ordon <- coef(m)[1]
pente <- coef(m)[2]
plot(dist~speed, data=cars)
abline(a=ordon, b=pente, lwd=1.5, col="green")
equa<-paste("dist=", round(ordon,2),"+", round(pente,2),"speed")
text(10,100, labels=equa)
```


dist= -17.58 + 3.93 speed
dist
speed
0
20
40
60
80
100
120
5
10
15
20
25

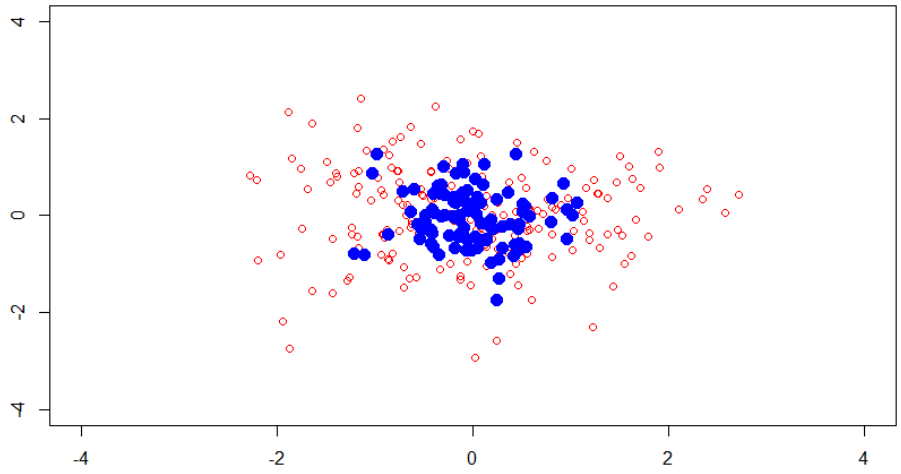
abline(h=) ou **abline(v=)** : La même fonction, associée à un paramètre h ou v, trace une droite horizontale ou verticale à la position indiquée respectivement sur l'axe de y ou des x. L'exemple ci-dessous trace sur le boxplot de iris$Sepal.Width une ligne continue rouge au niveau de la médiane et deux lignes pointillées (*lty=3*) bleues au niveau des valeurs minimales et maximales.

```
boxplot(iris$Sepal.Width)
abline(h=median(iris$Sepal.Width), col="red", lwd=1.5)
abline(h=max(iris$Sepal.Width), lty=3, col="blue", lwd=1.5)
abline(h=min(iris$Sepal.Width), lty=3, col="blue", lwd=1.5)
```



points(x,y) : affiche des points aux coordonnées x,y sur un graphe. L'exemple ci-dessous crée un plot vide (avec *type="n"*), puis ajoute des points dont les coordonnées sont générées aléatoirement selon une loi normale (avec *rnorm*). Les options *col*=, *cex*= et *pch*= permettent de paramétrer respectivement la couleur, la taille et le type de point.

```
plot(-4:4, -4:4, type = "n", xlab="", ylab="")
points(rnorm(200), rnorm(200), col = "red")
points(rnorm(100)/2, rnorm(100)/2, col = "blue", cex = 1.5, pch=19)
```

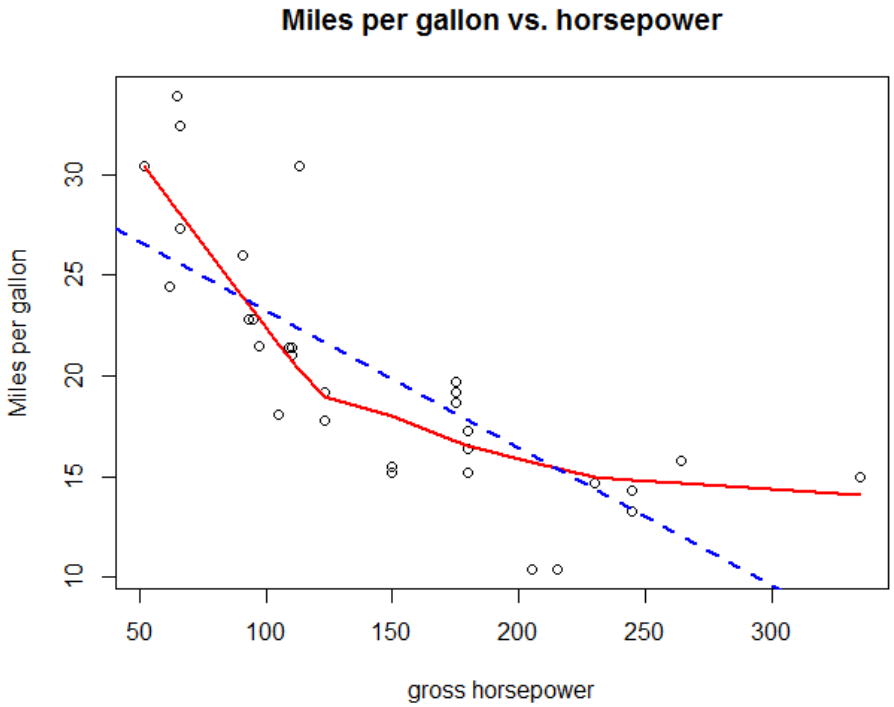


5.3.2. Les paramètres graphiques

La présentation des graphiques peut aussi être améliorée grâce aux paramètres graphiques, qui s'utilisent soit comme des options des fonctions graphiques, soit en arguments de la fonction **par()** qui permet de définir des paramètres graphiques de manière permanente. Une des options souvent utilisée est **mfrow(i,j)** déjà évoquée plus haut, pour partager la fenêtre graphique en i lignes et j colonnes, ce qui permet d'afficher plusieurs graphes sur la même fenêtre. Taper **?par** pour une liste complète des paramètres graphiques.

Exercice 7

- Reprenez le jeu de données mtcars et le plot de la variable mpg en function de la variable hp. Superposez la courbe lissée par la function loess (en rouge – trait plein) et la droite de regression (en bleu – trait pointillé).



6. Compléments de programmation R

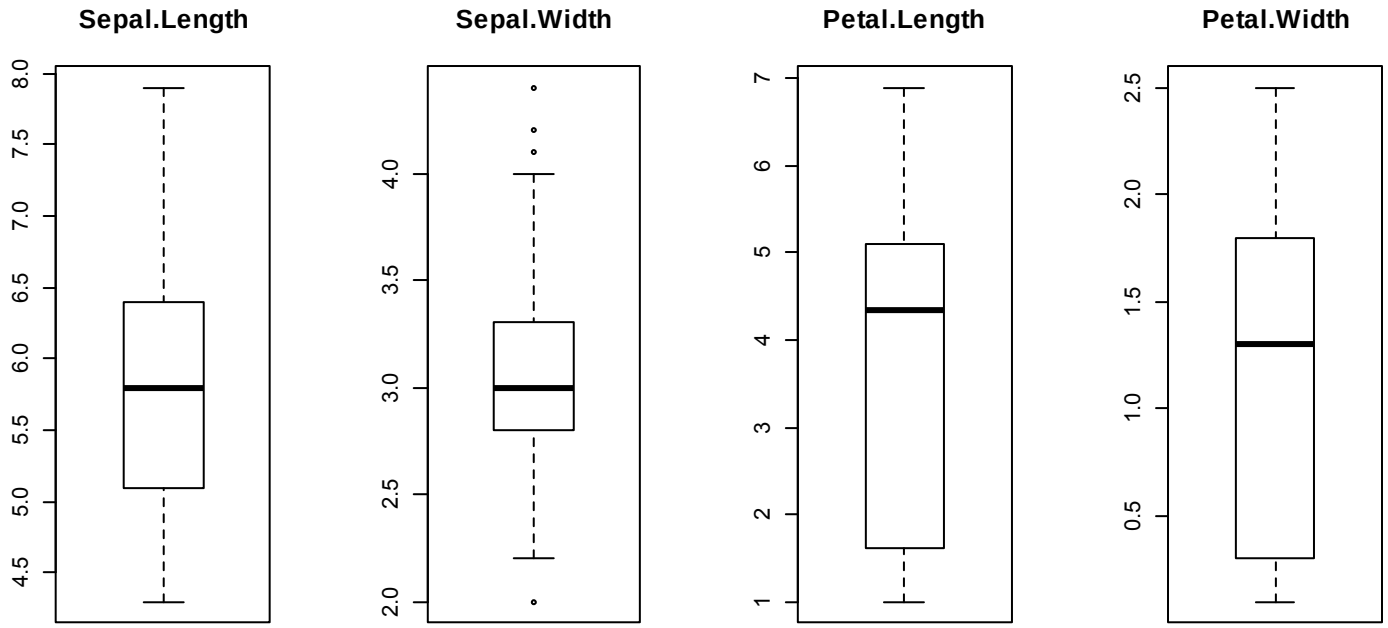
6.1. Les boucles for

R est un langage de programmation, qui possède donc des structures de contrôle comme les boucles for. Une boucle for permet d'exécuter automatiquement une série de commandes sur toutes les valeurs prises par un index. La liste de valeurs est un vecteur quelconque. Les commandes à exécuter pour chaque valeur de l'index sont encadrées par des { } :

```
for (i in liste_de_valeurs) {
        Commande 1
        Commande 2… }
```

Les instructions ci-dessous permettent de tracer des boxplots pour chacune des 4 premières colonnes du tableau iris. On utilise le paramètre mfrow() pour afficher sur la même fenêtre les 4 graphes :

```
par(mfrow=c(1,4))
for (i in 1:4) { boxplot(iris[,i], main=colnames(iris)[i])}
```


Sepal.Length
Sepal.Width
Petal.Length
Petal.Width
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
2.0
2.5
3.0
3.5
4.0
1
2
3
4
5
6
7
0.5
1.0
1.5
2.0
2.5

6.2. Les boucles implicites

Il est souvent plus efficace d'utiliser les boucles implicites de R qui découlent de sa caractéristique évoquée plus haut : R travaille sur des vecteurs. Nous allons voir ici un certain nombre de fonctions qui permettent d'effectuer des calculs sur les objets R de manière itérative : les fonctions de la famille **apply**.

apply(X, MARGIN=, FUN=) : applique la fonction FUN à la **matrice** X, en lignes (MARGIN=1), ou en colonnes (MARGIN=2).

Dans l'exemple ci-dessous, on applique la fonction apply à la matrice iris_mat obtenue à partir des 4 premières colonnes du data.frame iris, pour calculer la moyenne (mean) et l'écart-type (sd) sur les colonnes (MARGIN=2).

```
iris_mat <- as.matrix(iris[,1:4])
apply(iris_mat,MARGIN=2,FUN=mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   5.843333     3.057333     3.758000     1.199333
apply(iris_mat,MARGIN=2,FUN=sd)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   0.8280661    0.4358663    1.7652982    0.7622377
```

lapply(L, FUN) : applique la fonction FUN à tous les éléments de la **liste** L. Dans l'exemple ci-dessous la liste L est composée de 3 éléments : a, vecteur numérique des entiers de 1 à 10, beta composé de 6 valeurs décimales et logic, composée de 4 logiques. On note que dans une liste les éléments n'ont pas forcément la même longueur ni le même type. On note également que le calcul de la moyenne d'un vecteur logique est possible, les valeurs FALSE étant considérées comme 0 et TRUE comme 1 :

```
L <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
lapply(L,mean)
$a
[1] 5.5

$beta
[1] 4.535125

$logic
[1] 0.5
```

tapply(X, index, FUN) : applique la fonction FUN à un **vecteur** X pour chacun des groupes définis par le facteur index. Dans l'exemple ci-dessous nous calculons la valeur moyenne de Sepal.Width pour chacune des variétés d'iris définies dans Species :

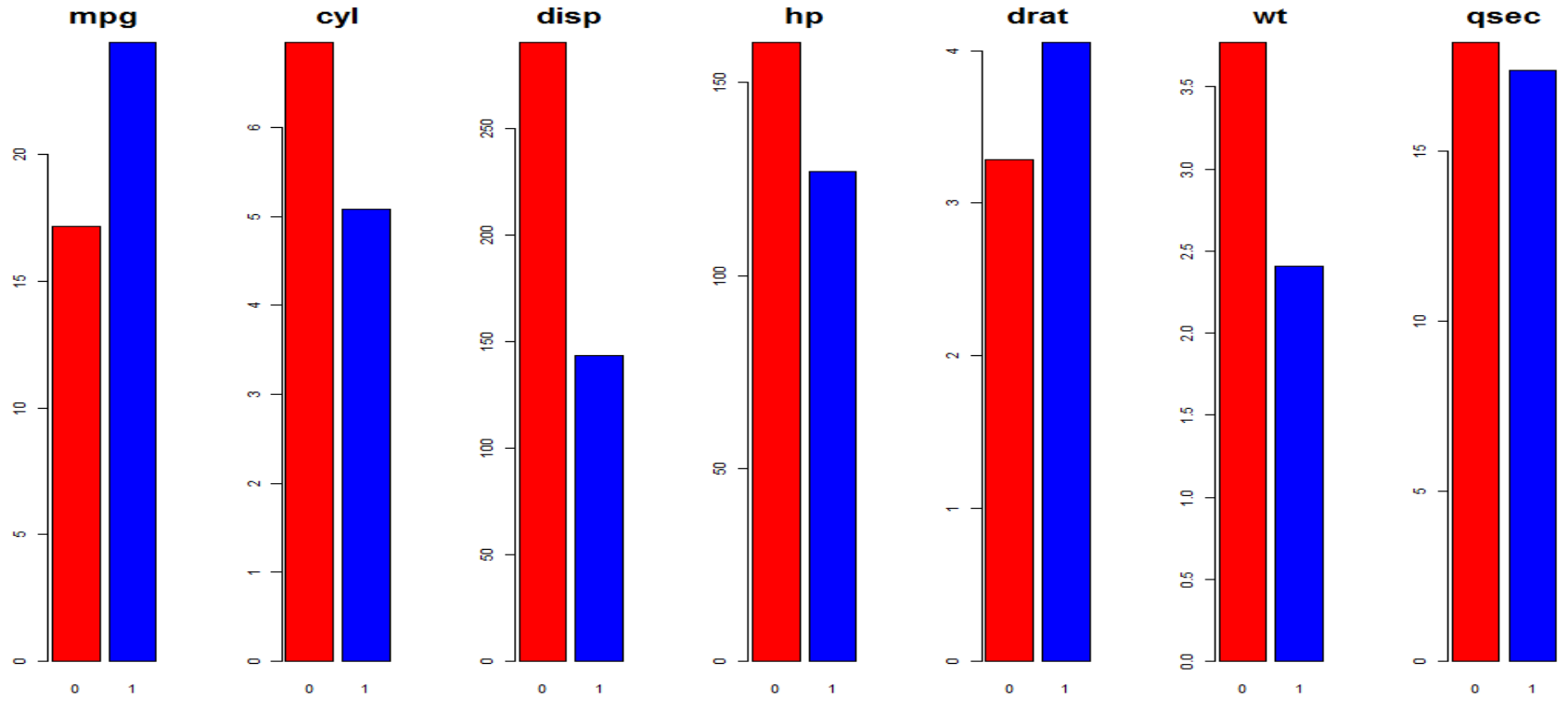
```
tapply(iris$Sepal.Width, iris$Species, mean)
    setosa versicolor  virginica
     3.428      2.770      2.974
```

Si on veut réaliser la même opération sur plusieurs variables numériques simultanément, par exemple les 4 mesures contenues dans le jeu de données iris, on utilisera la fonction **aggregate**, plus flexible, qui permet de spécifier plusieurs variables à traiter (ici les colonnes 1 à 4 d'iris correspondant aux variables numériques), plusieurs facteurs définis dans une liste (ici il n'y en a qu'un, la colonne 5 à laquelle on attribue le label Variété qui apparaîtra dans le résultat).

```
aggregate(iris[,1:4],by=list(Variété=iris[,5]), FUN=mean)
     Variété Sepal.Length Sepal.Width Petal.Length Petal.Width
1     setosa        5.006       3.428        1.462       0.246
2 versicolor        5.936       2.770        4.260       1.326
3  virginica        6.588       2.974        5.552       2.026
```

Exercice 8:

- Sur le jeu de données mtcars, calculer la valeur moyenne des variables 1 à 7 selon les modalités de la variable 'transmission' (0=automatic, 1=manual). Puis afficher le résultat sous forme de barplot en utilisant une boucle for.



6.3. Définir sa propre fonction

Jusqu'à présent, nous avons utilisé les fonctions définies dans R et ses librairies. Il est également possible de définir soi-même ses propres fonctions.
Dans le premier exemple, on définit une fonction très simple appelée **Salut**, qui affiche dans la console le mot Bonjour suivi du nom qui lui est passé en paramètre x (lors de l'appel de la fonction, bien mettre entre guillemets le nom à afficher). La fonction "cat" imprime dans la console et "\n" provoque un retour à la ligne après le texte affiché.

```
Salut <- function( x )
{
cat("Bonjour", x, "\n")
}

# Application
Salut("Le Monde")
Bonjour Le Monde
```

La fonction suivante **CoefVar** est un peu plus sophistiquée. Elle calcule le coefficient de variation sur un vecteur de données numériques. Notez l'utilisation de l'option na.rm=T dans les fonction sd() et mean() qui permet de traiter des vecteurs contenant des NA (valeurs manquantes). On peut tester ce qui se passe si on supprime cette option de l'une ou l'autre instruction. La fonction CoefVar renvoie la valeur du coefficient de variation grâce à l'instruction return() :

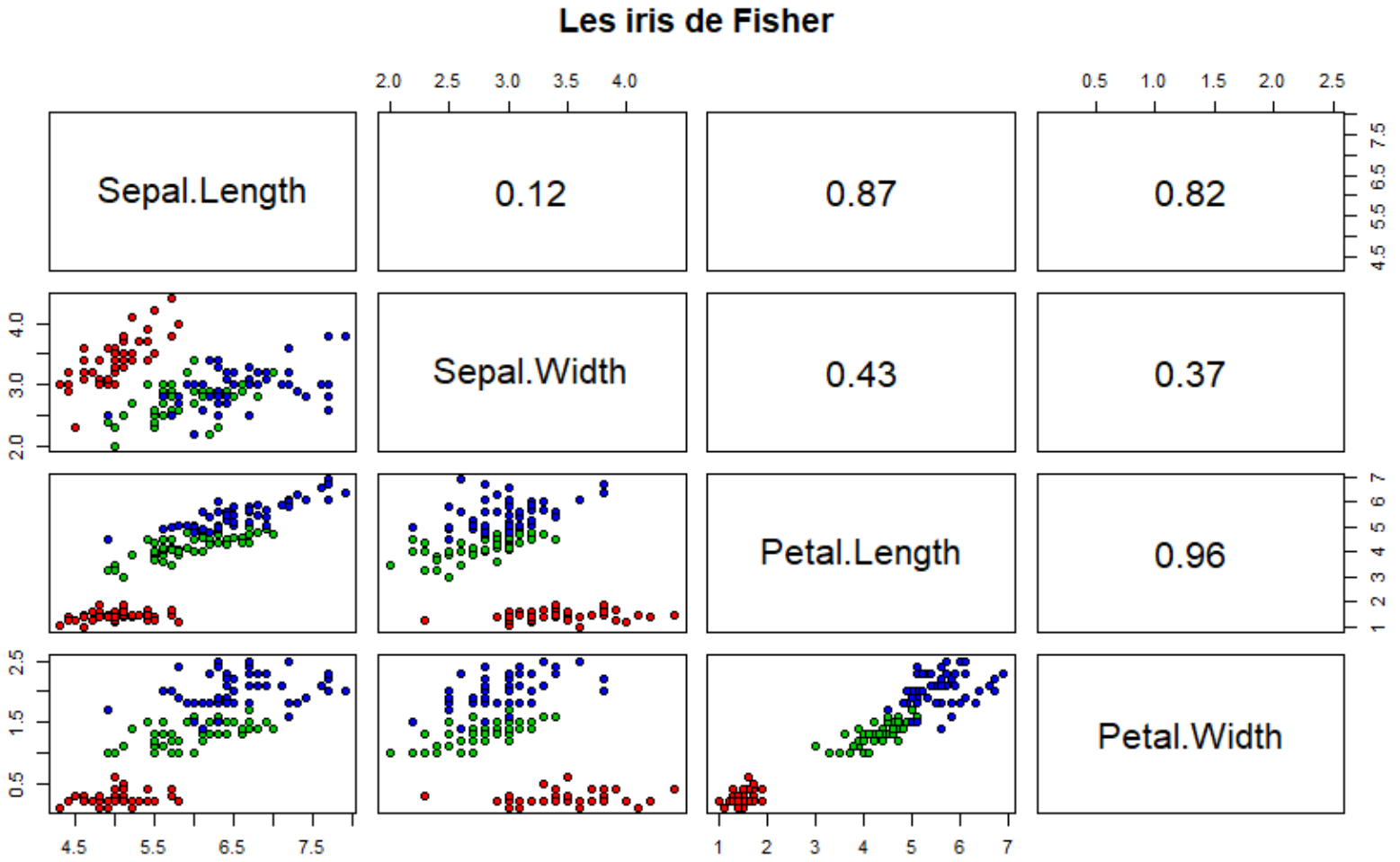
```
CoefVar <- function( datum )
{
# Fonction permettant de calculer le coefficient de variation
# Calcul de l'écart type
EcartType <- sd(datum, na.rm=T)
# Calcul de la moyenne
Moyenne <- mean(datum, na.rm=T)
# Calcul du coefficient de variation
CV <- 100 * EcartType / Moyenne
# Retourne le résultat
return(CV)
}
# Application à un jeu de données contenant un NA
TestData <- c( 1, 2, 3, 4, 5, NA )
A <- CoefVar( datum = TestData)
# Affichage du résultat en contrôlant le nombre de chiffres significatifs
print(A, digits=3)
[1] 52.7
```

Une fois définie, la nouvelle fonction peut être utilisée comme toute fonction R. Par exemple ici, on l'utilise comme fonction appliquée par une instruction aggregate au tableau iris :

```
ai <- aggregate(iris[,1:4],list(Variété=iris[,5]), CoefVar)
print(ai, digits=3)
     Variété Sepal.Length Sepal.Width Petal.Length Petal.Width
1     setosa         7.04        11.1        11.88        42.8
2 versicolor         8.70        11.3        11.03        14.9
3  virginica         9.65        10.8         9.94        13.6
```

Reprenons maintenant l'exemple du Draftsman display vu au §5.2.2. Nous allons définir une fonction que nous nommerons **panel.pearson** et qui sera passée en paramètre à l'argument upper.panel de la fonction prédéfinie **pairs()**, afin de remplacer les graphes situés au-dessus de la diagonale par la valeur du coefficient de corrélation de Pearson correspondant au couple de variables considéré.

```
panel.pearson <- function(x, y, ...) {
horizontal <- (par("usr")[1] + par("usr")[2]) / 2;
vertical <- (par("usr")[3] + par("usr")[4]) / 2;
text(horizontal, vertical, format(abs(cor(x,y)), digits=2) , cex=2)
}
# Application de la function :
pairs(iris[1:4], main = "Les iris de Fisher", pch = 21, bg =
c("red","green3","blue")[unclass(iris$Species)], upper.panel=panel.pearson)
```


Les iris de Fisher
Sepal.Length
0.12
0.87
0.82
Sepal.Width
0.43
0.37
Petal.Length
0.96
Petal.Width

6.4. Les dates sous R

6.4.1. Le type date

Les dates sont stockées comme un nombre de jours écoulés depuis le 1er janvier 1970 (valeurs négatives avant). Une date sous forme de chaine de caractères peut être convertie en date avec la fonction **as.Date()** qui, par défaut, attend une date au format aaaa-mm-dd :

```
datestring <- c("2007-06-22", "2004-02-13")
datestring
[1] "2007-06-22" "2004-02-13"
madate <- as.Date(datestring)
madate
[1] "2007-06-22" "2004-02-13"
```

En apparence c'est la même chose, mais le format date permet de faire des opérations, comme par exemple calculer le nombre de jours écoulés entre deux dates :

```
jours <- madate[1] - madate[2]
jours
Time difference of 1225 days
```

La fonction **Sys.Date()** renvoie la date du jour :

```
Sys.Date()
[1] "2023-03-16"
```

La fonction **date()** renvoie la date et aussi l'heure

```
date()
[1] "Thu Mar 16 17:15:01 2023"
```

Différents formats de date sont prédéfinis :

Format	Utilisation	Exemple
%Y	année au format long (4 chiffres)	2016
%y	année en format abrégé (2 chiffres)	16
%B	mois en format long	January
%b	mois au format abrégé	Jan
%m	mois au format numérique	01
%A	Jour de la semaine	Monday
%a	Jour de la semaine abrégé	Mon
%d	jour du mois au format numérique	07

Ces formats peuvent être utilisés pour l'affichage des dates. On peut y ajouter des séparateurs :

```
today <- Sys.Date()
format(today, format="%A %d %B %y")
[1] "jeudi 16 mars 23"
format(today, format="%d/%m/%y")
[1] "16/03/23"
```

Ils peuvent aussi être utilisés dans la fonction de conversion de chaine de caractères à date :

```
datestring2 <- c("05/01/1965", "16/08/1975")
madate2 <- as.Date(datestring2, "%d/%m/%Y")
format(madate2, "%A %d %B %y")
[1] "mardi 05 janvier 65" "samedi 16 août 75"
```

Pour extraire le jour, le mois ou l'année d'une date on utilisera aussi ces formats, qui renvoient des chaines de caractères et doivent donc être convertis en numériques si besoin :

```
annee <- as.numeric(format(madate2, "%Y")
mois <- as.numeric(format(madate2, "%m")
jour <- as.numeric(format(madate2, "%d")
```

Quelques fonctions applicables aux dates :

weekdays()	Jour de la semaine
months()	Mois au format caractère
quarters()	Trimestre

6.4.2. Le type POSIXct

Cette classe permet de tenir compte de l'heure en indiquant le nombre de secondes écoulées depuis le 1er janvier 1970 (et non de jours). La fonction **as.POSIXct()** permet de convertir une chaine de caractères en date de ce type. Elle attend 3 arguments : la chaine contenant la date, le format et un argument tz qui indique le fuseau horaire.

```
date <- "15 janvier 2017 12h57:10"
date_format <- "%d %B %Y %Hh%M:%S"
date_complete <- as.POSIXct(date,
                            format = date_format,
                            tz = "GMT")
[1] "2017-01-15 12:57:10 GMT"
```

Les formats listés ci-dessus pour le jour, le mois, l'année restent les mêmes. S'y ajoutent des formats pour les heures, minutes secondes :

Format	Utilisation	Exemple
%H	Heure au format 24h	16
%h	Heure au format 12h	4
%M	Minutes	36
%S	Secondes	14

Exercice 9:

- Mettre dans un objet **demain** de type Date la date de demain. Extraire de cet objet les différents éléments (jour, mois, année).
- Quel jour de la semaine êtes-vous ne(e) ?
- Calculer le nombre de jours écoulés entre le jour de votre naissance et aujourd'hui.

Bibliographie

Documents utiles :

R pour les débutants - Emmanuel Paradis (en français ou en anglais) - 77 pp.
https://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf

Introduction à l'environnement de programmation statistique R - Y. Brostaux - 22 pp.
https://www-polsys.lip6.fr/~safey/Enseignements/R/Documents/Brostaux-Introduction-au-R.pdf

simpleR – Using R for Introductory Statistics – John Verzani – 114 pp.
https://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf

Using R for Data Analysis and Graphics – JH Maindonald – 96 pp.
https://cran.r-project.org/doc/contrib/usingR.pdf

Sites Web :

Le site de référence : **The Comprehensive R Archive Network** http://cran.r-project.org/

Le moteur de recherche de R : http://www.rseek.org

Un manuel en ligne en anglais : **Quick-R** http://www.statmethods.net

Un manuel en ligne en français : **Aide mémoire R** http://www.duclert.org/

Introduction à R et au tidyverse – Julien Barnier (présente également l'approche tidyverse)
https://juba.github.io/tidyverse/

R news and tutorial: **R-bloggers** https://www.r-bloggers.com

Site de questions-réponses **Stackoverflow** : https://stackoverflow.com/

Aides-mémoire :

Short-refcard : en anglais (Tom Short) https://cran.r-project.org/doc/contrib/Short-refcard.pdf

Kauffmann_aide_memoire_R : en français, traduction de Short-refcard https://cran.r-project.org/doc/contrib/Kauffmann_aide_memoire_R.pdf

Pour aller plus loin..
Brève introduction au tidyverse

Tidyverse est une collection de paquets R adaptée au *Data Science*

En Data Science, une grosse partie du travail consiste au traitement, transformation, nettoyage et visualisation des données

Existence d'un déséquilibre entre disponibilités des paquets en modélisation de données (stat, modèles, etc..) et ressources pour la préparation de ces données

Il existe plusieurs extensions qui constituent le "coeur" du *tidyverse* :

- ggplot2 (visualisation)
- dplyr (manipulation des données)
- tidyr (remise en forme des données)
- purrr (programmation)
- readr (importation de données)
- tibble (tableaux de données)
- forcats (variables qualitatives)
- stringr (chaînes de caractères)

Traitement séquentiel de l'information grâce à l'utilisation de l'opérateur %>% appelé pipe pouvant se traduire par ET PUIS

Exemple : Nous souhaitons cuisiner des carottes

En base R

```
carottes_epluchees <- eplucher(carottes)
carottes_coupees <- couper(carottes_epluchees)
carottes_cuites <- cuire(carottes_coupees)

OU

carottes_cuites <- cuire(couper(eplucher(carottes)))
```

En tidyverse

```
carottes_cuites <- carottes %>%
                   eplucher() %>%
                   couper() %>%
                   cuire()
```

La suite dans une prochaine formation .. ☺