

424 lines (309 loc) · 14.2 KB

# Révisions R - Mémo

## Sommaire

- [barplot](#)
- [colMeans](#)
- [colnames](#)
- [cor](#)
- [data](#)
- [density](#)
- [fread](#)
- [file.exists](#)
- [grid](#)
- [hist](#)
- [lines](#)
- [lm](#)
- [legend](#)
- [matplot](#)
- [mean\(x\), sd\(x\)](#)
- [nchar](#)
- [order](#)
- [paste](#)
- [paste0](#)
- [pairs](#)
- [par](#)
- [pie](#)
- [plot](#)
- [prop.table](#)
- [read.csv](#)
- [rep](#)
- [round](#)
- [rainbow](#)
- [rowSums](#)
- [sapply](#)
- [seq](#)
- [system.time](#)
- [subset](#)

- [tapply](#)
- [trimws](#)

## Les fonctions

---

- `barplot(x, names.arg, col, main, xlab, ylab)` crée un diagramme en barres  
→ Exemple : `barplot(table(iris$Species), col="steelblue", main="Répartition des espèces")`
- `colMeans(x)` calcule la moyenne de chaque colonne d'une matrice ou `data.frame`  
→ Exemple : `colMeans(iris[,1:4])` calcule les moyennes des colonnes numériques
- `colnames(x)` retourne les noms des colonnes d'une matrice ou `data.frame`  
→ Exemple : `colnames(iris)` affiche les noms des colonnes
- `cor(x)` donne une matrice de corrélation pour une `dataframe/dataset`  
→ Exemple : `cor(iris[,1:4])` calcule la matrice de corrélation des 5 premières données quantitatives d' `iris`

■ **ATTENTION** : ne prend en compte que des données quantitatives

- `data(name)` charge un dataset pré-chargé dans R → Exemple : `data(iris)` charge le dataset `iris`
- `density(x, bw, adjust, kernel, trim, from, to)` calcule la densité de probabilité d'un vecteur `x`, où :
  - `x` est le vecteur de données pour lequel on souhaite estimer la densité
  - `bw` spécifie la méthode de sélection de la largeur de bande (bandwidth)
  - `adjust` permet d'ajuster la largeur de bande
  - `kernel` définit le type de noyau à utiliser (par exemple, "gaussian", "epanechnikov", etc.)
  - `trim` indique si l'on doit tronquer les valeurs en dehors de l'intervalle
  - `from` et `to` définissent l'intervalle sur lequel la densité est estimée

→ Exemple :

```
dens_perf <- density(Perf)
lines(dens_perf, col="darkblue", lwd=2)
```



- `fread(file_name, encoding, colClasses)` fait la même chose que `read.csv` mais est plus intelligente (elle n'a pas besoin qu'on lui tienne la main en lui précisant le séparateur etc...)

→ Exemple

```
data <- fread("data.csv",
  encoding="UTF-8",
  colClasses = c("numeric","character")
)
```



- `file.exists(filename)` vérifie si un fichier existe

→ Exemple : `file.exists("data.csv")` retourne TRUE ou FALSE

- `grid()` affiche la grille sur le graphique
- `hist(data, col, main, seq, freq)` crée un histogramme avec :
  - `data` est le jeu de données à afficher
  - `col` est la couleur des batons sur le graphique
  - `main` est le titre du graphique
  - `seq` est l'espacement des batons (réguliers ou non)
  - `freq` pour savoir si oui ou non on veut comparer l'histogramme à des densités (typiquement des densités gaussiennes).

→ Exemple :

```
Perf <- data$Perf[-1]

min_val <- round(min(Perf),2)
max_val <- round(max(Perf),2)
batons <- seq(min_val,max_val,by=0.01) # Ici le pas est régulier

hist(Perf, batons, col="skyblue", main="Histogramme du S&P500")
```



- `lines(x, y, col, lwd, lty)` ajoute des lignes sur un graphique déjà tracé :
  - `x, y` : coordonnées des points à relier
  - `col` : couleur
  - `lwd` : épaisseur du trait
  - `lty` : style du trait

→ Exemple

```
plot(x, y, type = "n") # crée la fenêtre sans tracer les points
lines(x, y, col = "red", lwd = 2)
```



- `lm(y~x, data)` construit une régression linéaire où : - `data` est la *data.frame* qui contient les données - `x` est le nom de la colonne de la variable explicative - `y` est le nom de la colonne de la variable à expliquer

→ Exemple : `reg <- lm(y ~ x, data=mes_donnees)` effectue cette tâche et affecte la valeur à la variable `reg`, avec `mes_donnees` comme la *data.frame* qui nous intéresse.

**Attention : bien faire attention aux noms des variables `y` et `x`. Il faut que ces noms de colonnes apparaissent dans la *data.frame* !**

- `legend(position, legend, pch, lwd, bg)` configure la légende sur le graphique, où :
  - `position` indique la position de la légende sur le graphique, généralement : `topleft`, `topright`, `bottomleft`, `bottomright`
  - `legend` est le titre de la légende (peut être un vecteur, cf. exemples ci-dessous)
  - `pch` est le type de point (peut être un vecteur)
  - `lwd` est l'épaisseur du trait (peut être un vecteur)
  - `bg` est la couleur du *background*

→ Exemple 1: `legend("topleft", legend="Jeu de données", pch=1,lwd=2)` ajoute une légende en haut à gauche avec les paramètres ci-contre.

→ Exemple 2

```
# Exemple vectorisé
noms_legendes <- colnames(mes_donnees)[-1]
couleurs_legendes <- c("#c0392b", "#bdc3c7", "#f39c12", "#2980b9", "#2c3e50")

legend("topleft",
      legend=noms_legendes,
      lty=1, lwd=2,
      col=couleurs_legendes,
      bg = "#ecf0f1")
```

- `matplot(x, y, type, pch, col, lty, lwd, xlab, ylab, main, ylim)` affiche plusieurs fonctions (chaque colonne de `y` est considérée comme une fonction de `x`) :

- `x` : vecteur des abscisses (ou NULL pour utiliser l'indice des lignes)
- `y` : matrice ou data.frame (chaque colonne devient une fonction à dessiner)
- `type`, `col`, `lty`, `lwd` : apparence des courbes
- `xlab`, `ylab`, `main`, `ylim` : étiquettes, titre et limites d'axe

→ Exemple : affichage de trois fonctions différentes

```
# Création des données
x <- seq(0, 2*pi, length.out = 200)
f1 <- sin(x)
f2 <- sin(2*x) / 2
f3 <- (cos(x))^2
Fonctions <- cbind(f1, f2, f3)

# Tracé de plusieurs fonctions en une seule commande
matplot(x, Fonctions, type = "l", lty = 1, lwd = 2,
      col = c("steelblue", "tomato", "darkgreen"),
      xlab = "x", ylab = "f(x)", main = "Exemples de fonctions")

legend("topright", legend = c("sin(x)", "sin(2x)/2", "cos^2(x)"),
      col = c("steelblue", "tomato", "darkgreen"), lty = 1, lwd = 2)
```

Différence avec `plot` : `matplot` est très pratique pour dessiner simultanément plusieurs colonnes d'une matrice/data frame, cela évite d'appeler plusieurs fois `plot` et lignes en gros.

- `mean(x)`, `sd(x)` permettent de calculer la moyenne et l'écart-type de `x`

- `nchar(x)` retourne la longueur de chaque chaîne de caractères

→ Exemple : `nchar("hello")` retourne 5

- `order(x)` donne une permutation des indices dans la version triée de `x` → Mézamor, cela permet de réindexer les vecteurs en une version triée :

```
x <- c(2,1,3)
```

```
x <- x[order(x)] # x devient triée !
```

→ On remarquera que l'on peut trier des datasets etc... *selon une seule colonne* :

```
data <- data[order(Mois), ] # on trie les données selon l'ordre chronologique
```



- `paste(x, y, sep, collapse)` concatène des chaînes de caractères, où :
  - `x`, `y` sont les chaînes à concaténer (peuvent être des vecteurs)
  - `sep` précise le séparateur entre les éléments, par défaut " "
  - `collapse` concatène les résultats avec ce séparateur si on travaille avec des vecteurs

→ Exemple :

```
paste("Bonjour", "monde", sep = " ") # retourne "Bonjour monde"  
paste(c("a", "b", "c"), collapse = "-") # retourne "a-b-c"
```



- `paste0(x, y)` concatène des chaînes de caractères **sans séparateur** (équivalent à `paste(x, y, sep="")`), où :
  - `x`, `y` sont les chaînes à concaténer (peuvent être des vecteurs)
  - `collapse` concatène les résultats avec ce séparateur si on travaille avec des vecteurs

→ Exemple :

```
paste0("Bonjour", "monde") # retourne "Bonjourmonde"  
paste0(c("a", "b", "c"), collapse = "-") # retourne "a-b-c"
```



- `pairs(x)` crée une matrice de nuages de points pour explorer les corrélations entre variables

→ Exemple : `pairs(iris[,1:4])` affiche tous les croisements possibles

- `par(mfrow, mar)` configure les paramètres graphiques, où :
  - `mfrow` divise la fenêtre en plusieurs sous-graphiques (ex: `c(2,2)` pour 2x2)
  - `mar` définit les marges du graphique

→ Exemple : `par(mfrow=c(2,2))` crée une grille de 4 graphiques

- `pie(x, labels, col)` crée un diagramme circulaire

→ Exemple : `pie(c(30,40,30), labels=c("A","B","C"), col=rainbow(3))`

- `plot(x,y,pch,col,xlab,ylab)` trace `y` en fonction de `x`. Les autres paramètres sont facultatifs :
  - `pch` est le type de point (*point character* littéralement)
  - `col` est le type de couleur utilisée(s) (peut être un vecteur pour insérer différentes couleurs, cf. exemples)
  - `xlab` est le *label* des abscisses
  - `ylab` est le *label* des ordonnées

→ Exemple 1: `plot(iris$Petal.Length, iris$Petal.Width, col = "forestgreen", pch=16)` trace `iris$Petal.Width` en fonction de `iris$Petal.Length`.

→ Exemple 2 :

```
# Création d'un vecteur de couleurs
couleurs <- rep("#16a085", nrow(iris))

indice_versicolor <- iris$Species == "versicolor"
indice_virginica <- iris$Species == "virginica"

couleurs[indice_versicolor] <- "#c0392b"
couleurs[indice_virginica] <- "#8e44ad"

# Tracé avec différentes couleurs

plot(mes_donnees$x, mes_donnees$y, col = couleurs, pch=16, xlab="Largeur des pétales", ylab = "
```

- `prop.table(x)` convertit un tableau en proportions (pourcentages)

→ Exemple : `prop.table(table(iris$Species))` affiche les proportions de chaque espèce

- `read.csv(file_name, header, sep, encoding, colClasses)` lit le contenu d'un fichier `.csv` (un tableur quoi), où :
  - `file_name` est le nom du fichier
  - `header` précise si oui ou non on a une ligne d'entête (avec le nom des colonnes par exemple)
  - `sep` précise le séparateur, généralement `,` ou `;`
  - `encoding` précise l'encodage du fichier, généralement `"UTF-8"`
  - `colClasses` précise le type de chaque colonne

→ Exemple :

```
data <- read.csv("data.csv",
  header = TRUE,
  sep = ",",
  encoding = "UTF-8",
  colClasses = c("numeric", "character")
)
```

- `rep(value, longueur)` crée un vecteur de longueur `longueur` avec `value` à chaque indice.

→ Exemple : `rep("red", 5)` crée un vecteur de `"red"` de longueur 5.

- `round(number, number_digits)` fournit une approximation de `number` avec `number_digits` chiffres après la virgule

→ Exemple : `round(2.555555, 2)` renvoie `2.56`

- `rainbow(n)` crée un vecteur de `n` couleurs de l'arc-en-ciel

→ Exemple : `plot(1:10, col=rainbow(10))` trace 10 points avec des couleurs différentes

- `rowSums(x)` calcule la somme de chaque ligne d'une matrice ou `data.frame`

→ Exemple : `rowSums(iris[,1:4])` somme les valeurs par ligne

- `sapply(x, function)` applique une fonction à chaque élément et simplifie le résultat

→ Exemple : `sapply(iris, class)` retourne le type de chaque colonne

- `seq(start,end,by)` crée un vecteur comprenant les nombres de `start` à `end` avec un pas régulier de `by` . → Exemple : `seq(1,3,1)` renvoie le vecteur `1 2 3`

- `system.time(expr)` mesure le temps d'exécution d'une expression

→ Exemple : `system.time(sum(1:1000000))` affiche le temps écoulé

- `subset(x, condition)` extrait un sous-ensemble de données selon une condition spécifiée :

- `x` est le vecteur ou la `data.frame` à filtrer
- `condition` est la condition logique à appliquer

→ Exemple : `subset(iris, Species == "setosa")` retourne les lignes d'iris où l'espèce est "setosa"

- `tapply(x,cat_group_by,function_x)` : même opération que le `GROUP BY` en SQL pour ensuite appliquer à chaque catégorie la fonction. → Exemple : `tapply(iris$Sepal.Length, iris$Species, mean)` calcule la moyenne de la longueur des sépales, groupée par espèce.

- `trimws(x)` enlève les espaces inutiles (en début ou en fin de mot) pour chaque chaîne de caractères contenue dans `x` .

→ Exemple

```
data$name_country <- trimws(data$name_country) # Enlève les espaces inutiles dans les noms de
```

Ça sauve bien des situations !