

# CM1 Langage R, Rstudio (et ChatGPT)

## Introduction et objectifs

L'objectif de cette séance est une brève présentation du langage de programmation R et de l'interface Rstudio. Il est essentiel de prendre en main cet outil très rapidement pour la réussite des travaux pratiques. Bien entendu, le contenu de cette séance est insuffisant pour cela. Voici quelques pointeurs utiles et libres pour débiter en R. Il est fondamental de les parcourir avant de débiter les travaux pratiques.

- R pour les débutants (Emmanuel Paradis)
- Ouvrage en français (Vincent Goulet)

Les manuels de R sont plus austères mais on peut les trouver utiles.

- Introduction à R
- Liste des fonctions de base.

On peut aussi faire un bon usage de la documentation en ligne, de ressources sur la toile et d'application telles que Open AI ChatGPT/Codex ou Copilot.

## Langage R

Le langage R est un langage de programmation interprété. Il repose sur l'exécution dynamique du programme par un autre programme interprète écrit en C. À l'origine, R a été développé pour des applications en analyse des données. Bien qu'il dépasse largement ce cadre, il reste particulièrement intéressant pour la science des données et les probabilités appliquées.

Le langage s'appuie sur la notion de **vecteur**, simplifiant le recours aux structures itératives (boucle for, while, etc.). Il n'y a pas de typage ou de déclaration obligatoire des variables. Le langage permet d'écrire des programmes courts, comportant quelques lignes de code seulement et le temps de développement est réduit.

## Utilisation de R

Comme tout programme R doit être installé sur le système avant de pouvoir l'utiliser. Il s'agit d'un logiciel libre qu'il est possible de télécharger à partir du site CRAN pour une installation personnelle.

R est en premier lieu une application offrant une invite de commande symbolisée par un "prompt" (>). L'utilisation de R en ligne de commande est toutefois restreinte, et réservée à des opérations très simples. Par exemple, on peut effectuer des opérations de calcul telles que calculer  $\log(3)$

```
log(3)
```

```
## [1] 1.098612
```

ou calculer la moyenne d'un vecteur de nombres

```
# x contient 100 variables aléatoires uniformément réparties entre 0 et 1  
x <- runif(100)
```

```
# la fonction mean() calcule la moyenne du vecteur  
help(mean)  
mean(x)
```

```
## [1] 0.4647211
```

R peut être utilisé sous l'invite du système en mode de traitement par lot ("batch processing") grâce à la commande système **R CMD BATCH**. Dans la suite, nous utiliserons essentiellement l'exécution de scripts (quelques commandes de R successives) à partir de l'interface **Rstudio**. Retenons que les principaux modes d'utilisation de R sont les suivants

- *Utilisation en ligne de commande* : une commande simple est utilisée en mode interactif. La suite des commandes est enregistrée dans le fichier *.Rhistory*, et peut être retrouvée par la commande `R history()`.
- *Utilisation d'un script* : un programme court ayant l'extension **.R** ou **.r** stocké sur le disque. Les scripts peuvent être exécutés par la commande `source()` ou en mode batch par **R CMD BATCH**.
- *Utilisation d'une fonction (routine)* : une suite d'instructions effectuant un traitement spécifique bien identifié et pouvant être réutilisée.
- *Utilisation d'une bibliothèque R* : un programme écrit en R (et pouvant utiliser d'autres langages) appelé "package" qui rend disponible des fonctions à d'autres projets de programmation en conservant l'intégrité de son implémentation. Nous utiliserons par exemple des bibliothèques de "réseaux neuronaux" qui nous permettront de programmer en quelques lignes de code R des méthodes d'apprentissage probabiliste très puissantes.

Le répertoire de travail (workspace) de R est le dossier par défaut dans lequel le logiciel va rechercher des fichiers de script ou de données et va sauvegarder l'espace de travail dans le fichier **.RData**. Le répertoire de travail est déterminé au lancement de R.

Notons que lorsque l'on quitte R, le logiciel propose de sauver les objets créés pendant la session dans le fichier **.Rdata**. En acceptant et en relançant le logiciel, on retrouvera la session dans l'état où on l'a quittée.

### Exemple de données manipulées dans R

Nous souhaitons *simuler* des tirages aléatoires pris dans un ensemble fini de valeurs numérotées (penser au lancer d'un dé non nécessairement équilibré) selon une loi de probabilité donnée. Une loi de probabilité est une suite de nombres positifs associés à chaque valeur numérotée dont la somme est égale à 1.

**ChatGPT, écris un code R pour simuler une loi de probabilité discrète.**

Réponse : Bien sûr ! Voici un exemple de code R pour simuler une loi de probabilité discrète.

```
# Définition des probabilités
# Remplacez ces valeurs par les probabilités de votre choix
probabilites <- c(0.2, 0.3, 0.1, 0.4)

# Générer un échantillon aléatoire en utilisant les probabilités
echantillon <- sample(1:length(probabilites),
                      size = 100,
                      replace = TRUE,
                      prob = probabilites)

# Afficher les valeurs de l'échantillon
print(echantillon)

##      [1] 2 1 4 2 3 2 2 3 4 3 2 4 4 4 3 2 4 3 2 3 1 3 4 1 3 4 4 2 4 4 4 2 2 4 3 4 1
##     [38] 3 4 4 4 3 4 2 4 4 4 3 2 3 4 2 2 3 1 1 2 4 1 2 4 4 2 1 1 2 4 4 2 4 4 4 4 1
##     [75] 4 2 3 2 4 4 2 4 4 2 2 4 3 4 1 1 3 2 4 4 2 4 4 4 2 4

# Afficher une table des valeurs de l'échantillon
table(echantillon)

## echantillon
##  1  2  3  4
## 12 27 17 44
```

Explication du code :

- La première étape consiste à définir les probabilités de la distribution discrète. Dans cet exemple, j'ai utilisé un vecteur probabilités contenant quatre valeurs, mais vous pouvez les remplacer par les probabilités correspondant à votre distribution.
- Ensuite, la fonction `sample()` est utilisée pour générer un échantillon aléatoire à partir des valeurs de la distribution. Les arguments `size` (taille de l'échantillon), `replace` (indiquant si les échantillons sont tirés avec remplacement) et `prob` (les probabilités correspondantes) sont spécifiés.
- Enfin, l'échantillon généré est affiché à l'aide de la fonction `print()`.

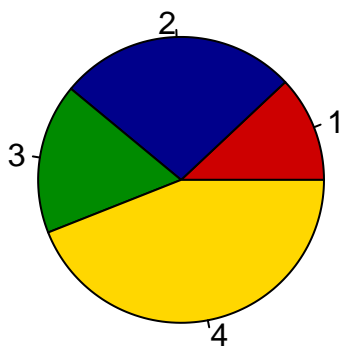
Merci l'a(m)i, nous pouvons compléter cette réponse avec un peu plus de couleur. Par exemple, nous pouvons représenter l'échantillon de valeurs simulés par un camembert (pie) ou par un diagramme en barre.

```
# partitionne la fenetre graphique
par(mfrow = c(1,2))

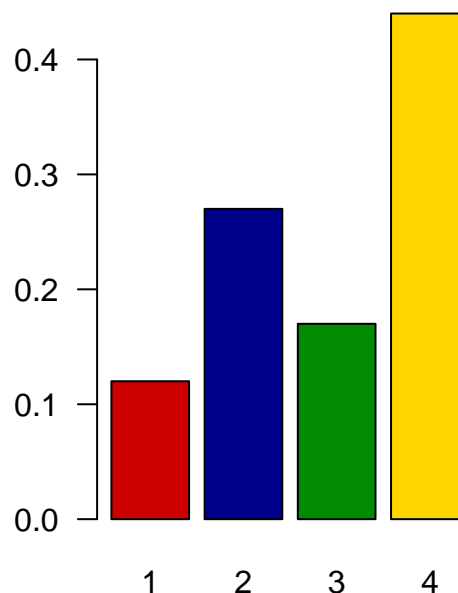
# un camembert est une tarte (pie) en anglais
pie(table(echantillon),
    col = c("red3", "darkblue", "green4", "gold"),
    main = "Probabilités empiriques")

# diagramme en barre
barplot(table(echantillon)/100,
    col = c("red3", "darkblue", "green4", "gold"),
    main = "Probabilités empiriques", las = 1)
```

**Probabilités empiriques**



**Probabilités empiriques**



### Principales classes d'objets

Dans le langage R, toute forme de donnée est appelée *objet*. Les variables, les fonctions, les opérateurs, et même le symbole représentant le nom d'un objet est un objet. Les objets possèdent au minimum un mode (ou classe) et une longueur. Les principales classes d'objet sont les suivantes

- **numeric**: un vecteur contenant des valeurs numériques,
- **character** : un vecteur contenant des caractères ou des chaînes de caractères,

- **logical** : un vecteur contenant des valeurs logiques ou binaires,
- **matrix** : un vecteur contenant des valeurs numériques, recodé comme une matrice comportant  $n$  lignes et  $p$  colonnes,
- **list** : une liste d'objets potentiellement hétérogène, contenant par exemple des vecteurs, des matrices, des caractères,
- **data.frame** : une liste d'objets de même longueur organisée en tableau,
- **function** : une fonction.

Par exemple, supposons que la variable **x** prenne la valeur 8.

```
x <- 8
x
```

```
## [1] 8
```

Nous voyons que **x** est un vecteur dont la première valeur est égale à 1.

```
x[1]
```

```
## [1] 8
```

Nous vérifions que le vecteur **x** est de longueur 1 et qu'il est de la classe 'numeric'.

```
length(x)
```

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

```
class(x)
```

```
## [1] "numeric"
```

Pour créer une liste, on peut utiliser la fonction `list`. La plupart des objets manipulés par les bibliothèques R sont des listes. Les différents attributs d'une liste sont accessibles par le symbole dollar (\$) ou par le double crochet.

```
mes_betes <- list(animal = c("chat", "chien"),
                  sex = c("M", "F"),
                  age = c(2, 8),
                  vaccine = c(TRUE, TRUE))
```

Pour obtenir les attributs de la liste "mes\_betes", on pourra utiliser le symbole dollar

```
mes_betes
```

```
## $animal
## [1] "chat" "chien"
##
## $sex
## [1] "M" "F"
##
## $age
## [1] 2 8
##
## $vaccine
## [1] TRUE TRUE
```

Ainsi, pour obtenir l'argument `animal`, on peut faire

```
mes_betes$animal
```

```
## [1] "chat" "chien"
```

On pourra alors corriger les erreurs éventuelles. Si le second animal est un raton laveur, il suffit de modifier l'argument de la liste

```
mes_betes$animal[2] <- "raton laveur"
mes_betes
```

```
## $animal
## [1] "chat"      "raton laveur"
##
## $sex
## [1] "M" "F"
##
## $age
## [1] 2 8
##
## $vaccine
## [1] TRUE TRUE
```

Remarquons que tous les attributs ont la même longueur. Plutôt que le format de liste, nous pouvons organiser les données sous la forme d'un tableau (`data.frame`). Un objet de type `data.frame` est une liste particulière organisée en colonnes. Un `data.frame` peut être créé de la manière suivante.

```
# les noms sont donnés aux colonnes du tableau de données
mes_betes <- data.frame(animal = c("chat", "raton laveur"),
                        sex = c("M", "F"),
                        age = c(2, 8),
                        vaccine = c(TRUE, TRUE))
mes_betes
```

```
##      animal sex age vaccine
## 1      chat  M   2    TRUE
## 2 raton laveur F   8    TRUE
```

Dans la suite, nous serons amenés à manipuler des tableaux de données contenant des statistiques sur le sport (football) et nous nous interrogerons sur l'existence de loi de probabilités décrivant les scores des matchs.

```
#install.packages("worldfootballR")
library(worldfootballR)
df <- fb_match_results(country = c("FRA"),
                       gender = "M",
                       season_end_year = 2022)
head(df)
```

```
##      Competition_Name Gender Country Season_End_Year      Round Wk Day
## 1      Ligue 1          M      FRA      2022 Regular season  1 Fri
## 2      Ligue 1          M      FRA      2022 Regular season  1 Sat
## 3      Ligue 1          M      FRA      2022 Regular season  1 Sat
## 4      Ligue 1          M      FRA      2022 Regular season  1 Sun
## 5      Ligue 1          M      FRA      2022 Regular season  1 Sun
## 6      Ligue 1          M      FRA      2022 Regular season  1 Sun
##      Date Time      Home HomeGoals Home_xG      Away AwayGoals Away_xG
## 1 2021-08-06 21:00 Monaco         1      1.5      Nantes         1      0.2
## 2 2021-08-07 17:00 Lyon          1      1.4      Brest          1      0.8
## 3 2021-08-07 21:00 Troyes        1      0.9      Paris S-G         2      1.2
## 4 2021-08-08 13:00 Rennes        1      0.7      Lens             1      2.1
## 5 2021-08-08 15:00 Bordeaux      0      0.7      Clermont Foot     2      3.1
## 6 2021-08-08 15:00 Strasbourg    0      0.3      Angers            2      0.9
```

##	Attendance	Venue	Referee	Notes	MatchURL
## 1	7500	Stade Louis II.	Antony Gautier		
## 2	29018	Groupama Stadium	Mikael Lesage		
## 3	15248	Stade de l'Aube	Amaury Delerue		
## 4	22567	Roazhon Park	Bastien Dechepy		
## 5	18748	Stade Matmut-Atlantique	Florent Batta		
## 6	23250	Stade de la Meinau	Jeremy Stinat		

##	MatchURL
## 1	<a href="https://fbref.com/en/matches/5a71bd1f/Monaco-Nantes-August-6-2021-Ligue-1">https://fbref.com/en/matches/5a71bd1f/Monaco-Nantes-August-6-2021-Ligue-1</a>
## 2	<a href="https://fbref.com/en/matches/22f9fbae/Lyon-Brest-August-7-2021-Ligue-1">https://fbref.com/en/matches/22f9fbae/Lyon-Brest-August-7-2021-Ligue-1</a>
## 3	<a href="https://fbref.com/en/matches/081fcc93/Troyes-Paris-Saint-Germain-August-7-2021-Ligue-1">https://fbref.com/en/matches/081fcc93/Troyes-Paris-Saint-Germain-August-7-2021-Ligue-1</a>
## 4	<a href="https://fbref.com/en/matches/efa9bd32/Rennes-Lens-August-8-2021-Ligue-1">https://fbref.com/en/matches/efa9bd32/Rennes-Lens-August-8-2021-Ligue-1</a>
## 5	<a href="https://fbref.com/en/matches/3bd58e2c/Bordeaux-Clermont-Foot-August-8-2021-Ligue-1">https://fbref.com/en/matches/3bd58e2c/Bordeaux-Clermont-Foot-August-8-2021-Ligue-1</a>
## 6	<a href="https://fbref.com/en/matches/d5c51de9/Strasbourg-Angers-August-8-2021-Ligue-1">https://fbref.com/en/matches/d5c51de9/Strasbourg-Angers-August-8-2021-Ligue-1</a>

Nous cherchons à récupérer le score à domicile, c'est à dire, le nombre de buts marqués par l'équipe qui reçoit, pour chaque match de la saison de championnat de France 2021-2022.

```
colnames(df) # les noms des colonnes
```

```
## [1] "Competition_Name" "Gender"          "Country"         "Season_End_Year"
## [5] "Round"           "Wk"             "Day"             "Date"
## [9] "Time"            "Home"           "HomeGoals"       "Home_xG"
## [13] "Away"            "AwayGoals"      "Away_xG"         "Attendance"
## [17] "Venue"           "Referee"        "Notes"           "MatchURL"
```

*# On peut obtenir la colonne "HomeGoals" de la manière suivante*

```
but_a_domicile <- df[, "HomeGoals"]
```

*# Méthode alternative : le \$ désigne un attribut de liste*

```
but_a_domicile <- df$HomeGoals
```

*# Méthode alternative : attach/detach*

```
attach(df)
but_a_domicile <- HomeGoals
detach(df)
```

La fonction `table` crée un histogramme des données de manière automatique

```
table(but_a_domicile) # help(table)
```

```
## but_a_domicile
##  0  1  2  3  4  5  6
## 84 128 92 41 22 10 5
```

## Manipulations vectorielles élémentaires

Le langage R permet de nombreuses manipulations vectorielles. Nous décrivons ci-dessous deux exemples de manipulations vectorielles élémentaires. Dans le premier exemple, nous chercherons à extraire les éléments d'un vecteur satisfaisant une condition particulière (valeurs supérieures à deux). Dans le deuxième exemple, nous remplacerons certains éléments d'une matrice par des valeurs nulles.

Tout d'abord, considérons le vecteur suivant (notons que `c()` permet de créer un vecteur)

```
x <- c(1,6,7,1,0,9,2,3) # c() permet de créer un vecteur
x
```

```
## [1] 1 6 7 1 0 9 2 3
```

Supposons que l'on cherche à *extraire les valeurs plus grandes que 2* du vecteur `x`. Dans ce cas, nous pouvons créer un vecteur logique

```
(x > 2)
```

```
## [1] FALSE TRUE TRUE FALSE FALSE TRUE FALSE TRUE
```

et obtenir le résultat de la manière suivante.

```
x[x > 2]
```

```
## [1] 6 7 9 3
```

Nous obtenons les indices vérifiant la condition (`x>2`) avec la fonction `which`.

```
which(x > 2)
```

```
## [1] 2 3 6 8
```

Nous voyons que le calcul vectoriel est équivalent au code suivant utilisant une boucle `for` et un test

```
y <- NULL # initialiser la variable résultat par l'objet "NULL"
n <- length(x)

for (i in 1:n){ # 1:n est un vecteur de valeurs de 1 à n
  if (x[i] > 2){
    y <- c(y, x[i])
  }
}
print(y) # on pourrait simplement écrire "y"
```

```
## [1] 6 7 9 3
```

Notre deuxième exemple est un peu plus subtil. Nous créons une matrice de taille 9 fois 11 contenant les 99 premiers entiers et nous cherchons à *remplacer les valeurs paires de cette matrice par des valeurs nulles*.

```
M <- matrix(1:99, nrow = 9, ncol = 11)
dim(M) # dimensions de la matrice M
```

```
## [1] 9 11
```

```
print(M[2,4]) # affiche l'élément [2,4] de la matrice
```

```
## [1] 29
```

Puisque les matrices sont codées comme des vecteurs, la solution s'écrit tout simplement de la manière suivante

```
M[M %% 2 == 0] <- 0 # le reste de la division (modulo) s'écrit "%%"
```

Nous pouvons vérifier le résultat

```
M
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
## [1,]    1    0   19    0   37    0   55    0   73    0   91
## [2,]    0   11    0   29    0   47    0   65    0   83    0
## [3,]    3    0   21    0   39    0   57    0   75    0   93
## [4,]    0   13    0   31    0   49    0   67    0   85    0
## [5,]    5    0   23    0   41    0   59    0   77    0   95
## [6,]    0   15    0   33    0   51    0   69    0   87    0
## [7,]    7    0   25    0   43    0   61    0   79    0   97
## [8,]    0   17    0   35    0   53    0   71    0   89    0
```

```
## [9,] 9 0 27 0 45 0 63 0 81 0 99
```

### Application à un jeu de données

La bibliothèque ‘*mlbench*’ (“machine learning benchmarks”) contient des jeux de données que nous pouvons analyser et à partir desquels nous pourrions faire des prédictions. L’exemple **BreastCancer** contient des données biomédicales concernant le cancer du sein (les données concernent 699 patientes). La fonction **data** permet de charger les données en mémoire sous forme de tableau (**data.frame**). Avec la fonction **head** nous pouvons visualiser les premières lignes du tableau ‘BreastCancer’.

```
#install.packages("worldfootballR")
library('mlbench')
data(BreastCancer)
head(BreastCancer)
```

```
##      Id Cl.thickness Cell.size Cell.shape Marg.adhesion Epith.c.size
## 1 1000025          5         1         1           1           2
## 2 1002945          5         4         4           5           7
## 3 1015425          3         1         1           1           2
## 4 1016277          6         8         8           1           3
## 5 1017023          4         1         1           3           2
## 6 1017122          8        10        10           8           7
##  Bare.nuclei Bl.cromatin Normal.nucleoli Mitoses      Class
## 1           1          3              1         1    benign
## 2           10         3              2         1    benign
## 3            2          3              1         1    benign
## 4            4          3              7         1    benign
## 5            1          3              1         1    benign
## 6           10          9              7         1 malignant
```

Nous cherchons à récupérer l’ensemble des diagnostics des patientes (colonne **Class**) et souhaitons compter le nombre de diagnostics bénins (“benign”).

```
colnames(BreastCancer) # les noms des colonnes
```

```
## [1] "Id"           "Cl.thickness" "Cell.size"    "Cell.shape"
## [5] "Marg.adhesion" "Epith.c.size" "Bare.nuclei"  "Bl.cromatin"
## [9] "Normal.nucleoli" "Mitoses"      "Class"
```

```
# On peut obtenir la colonne "Class" de la manière suivante
# On peut aussi remplacer "Class" par 11
```

```
status <- BreastCancer[, "Class"]
```

```
# Méthode alternative : le $ désigne un attribut de liste
status <- BreastCancer$Class
```

Le résultat est donné par la commande suivante

```
sum(status == "benign")
```

```
## [1] 458
```

Notons qu’il est aussi possible d’utiliser la fonction **table** qui crée un histogramme des données de manière automatique

```
table(status) # help(table)
```

```
## status
```



```
##      benign malignant
##      458           241
```

## Fonctions de base

La programmation en R s'appuie sur l'utilisation de fonctions et l'utilisation de l'aide en ligne `help()`. La structure d'une fonction est la suivante

```
f <- function(arg){
  # body
}
```

Des centaines de fonctions de base sont contenues dans les bibliothèques **base**, **stats**, **graphics**, etc, et sont chargées par défaut en lançant R. Il est impossible d'énumérer l'ensemble de ces fonctions. C'est en pratiquant régulièrement l'écriture de commandes, en utilisant l'aide, les tutoriaux en ligne (vignettes) et internet que l'on progresse le plus vite dans l'apprentissage des principales commandes et fonctions de R. Des applications web aux graphiques les plus élaborés en passant par des maths de haut niveau, **tout** est réalisable en R.

Voici quelques fonctions clés. Pour lire des données locales ou distantes ou pour en sauvegarder, nous pouvons utiliser les fonctions suivantes \* `read.table()`, `write.table()` \* `scan()`, `write()` \* `read.csv2()`, `write.csv()`, etc.

La documentation de ces fonctions est directement disponible dans R. Pour effectuer des calculs statistiques de base, nous pouvons utiliser les fonctions suivantes

- `mean()`, `median()`, `sd()`
- `table()`, `hist()`, `pie()`, etc.

Pour effectuer des tirages aléatoires, nous pouvons utiliser les fonctions suivantes

- `sample()` pour un tirage d'urne,
- `rnorm()` pour un tirage de loi normale,
- `runif()` pour un tirage de loi uniforme, etc.

La fonction `apply()` est particulièrement importante car elle permet d'appliquer une fonction aux lignes ou aux colonnes d'une matrice ou d'un tableau, évitant ainsi le recours à une boucle `for()`. Par exemple, nous souhaitons calculer la somme des valeurs de chaque colonne de la matrice 'M'.

```
apply(M, MARGIN = 2, FUN = sum)
```

```
## [1] 25 56 115 128 205 200 295 272 385 344 475
```

En R, il est aussi possible de calculer cette somme directement à l'aide de la fonction `colSums()`.

```
colSums(M)
```

```
## [1] 25 56 115 128 205 200 295 272 385 344 475
```

Finalement, notons l'usage de la fonction *pipe* de la bibliothèque **magrittr**. Un peu comme le "pipe" sous Unix/Linux, cet opérateur permet d'organiser séquentiellement les traitements à effectuer et gagne en lisibilité. Pour reproduire l'exemple précédent, nous pouvons écrire

```
library(magrittr)
```

```
M %>% colSums() # M est passée en argument à la fonction colSums
```

```
## [1] 25 56 115 128 205 200 295 272 385 344 475
```

Si on souhaite calculer la moyenne du sinus de 10000 nombres aléatoires pris entre 0 et 2 et arrondir le résultat à la troisième décimale, il suffit de lire la phrase à l'envers et d'écrire

```
runif(10000, 0, 2) %>% sin() %>% mean() %>% round(3)
```

```
## [1] 0.708
```

Cette opération évite les appels imbriqués et rend le code plus lisible. Elle est équivalente à la commande suivante.

```
round(mean(sin(runif(10000, 0, 2))), digit = 3)
```

```
## [1] 0.705
```

Remarquons que le résultat produit peut être un peu différent de celui de la ligne précédente, car il est dépendant de valeurs aléatoires.

## Bonnes pratiques pour les travaux pratiques

Programmer en R ou dans un autre langage doit permettre à un autre programmeur ou un autre utilisateur de comprendre l'organisation d'un projet, et généralement de faciliter la reproductibilité du travail effectué. Pour cela, une clé est l'organisation de son espace de travail. R possède de très bons outils pour cela (Rstudio).

Pour les travaux pratiques, il est important de créer un répertoire pour chaque TP. Le répertoire de travail d'un TP pourra contenir les sous-répertoires suivants

- Data : un répertoire contenant les données du TP (dans les TPs, les données seront chargées en mémoire).
- Codes : un répertoire contenant les scripts ou les cahiers de travail en Rmd.
- Documents : un répertoire contenant des ressources séparées, par exemple les documents de référence.
- Comptes\_Rendus : un répertoire contenant le fichier à rendre.

Il est de plus très important de respecter quelques recommandations en matière de nomenclature pour l'écriture de code. Quelques règles d'écriture peuvent être trouvées en suivant le lien suivant : [Coding style](#), mais ce n'est pas exhaustif.

Les codes étant en général très courts, on s'efforcera de commenter systématiquement les instructions des programmes à l'aide d'un symbole dièse. Pour les noms de variables, nous utiliserons la convention "Wickham" utilisant des mots en lettres minuscules séparés par des tirets. Par exemple, pour nommer une fonction permettant de dire "Coucou", on écrira

```
dire_coucou <- function(){  
  cat("Coucou \n")  
}
```

```
dire_coucou()
```

```
## Coucou
```

## Rstudio

Rstudio est un programme dont nous nous servirons pour faire les TPs. Il est essentiel de se familiariser rapidement avec l'interface. Deux avantages intéressants que nous tirerons de ce programme est 1) la possibilité d'associer un projet à un espace de travail, 2) la possibilité de rédiger les TPs et les comptes rendus de TP de manière interactive en même temps que l'on effectue le TP. Pour cela, nous utiliserons en utilisant un cahier de travail, **Notebook**.

R studio est plus qu'une interface graphique. Ce programme permet de coder en R (ou dans un autre langage), d'ouvrir des scripts ou des cahiers, de gérer ses fichiers, de consulter l'aide, d'installer des bibliothèques, d'afficher des graphiques, de gérer le versionnage du projet (git), etc.

## Projets

Rstudio permet d'associer un “projet” à un espace de travail. Le projet peut être créé dans l'onglet **File**, en sélectionnant **File > New Project** et en suivant les instructions de l'interface.

L'intérêt d'un projet est de pouvoir retrouver l'environnement de travail tel qu'on le laisse à la fermeture de l'application. Cela permet aussi de changer de projet très facilement et d'y revenir plus tard.

## Notebooks (Rmd)

Les cahiers (**Notebooks**) sont le mode de travail interactif que nous privilégierons en TP. Un cahier est un programme écrit en langage **Rmarkdown**. **Rmarkdown** est un langage de balise permettant de mélanger du texte et du code, que l'on exécute à la volée en mode interactif. Il est aussi possible d'intégrer des images, des liens web, des commandes html ou des commandes latex pour inclure des équations.

On peut “compiler” un fichier Rmarkdown pour une sortie en format PDF ou html. En TPs, il vous sera demandé de fournir obligatoirement un compte rendu au format html, pour que l'enseignant puisse le visualiser à partir de l'application de dépôt (TEIDE). Attention, tout autre format sera ignoré.

Il est inutile de connaître le langage Rmarkdown à fond. Des fichiers préparés vous permettront de faire les TPs en simplifiant la phase de rédaction. Une feuille d'aide contenant les quelques commandes utiles de Rmarkdown est disponible dans l'onglet **Help** de Rstudio.

L'ensemble de ce cours est écrit en **Rmarkdown**.