# R pour l'économétrie

Abdoul Oudouss Diakite

# Contents

# Bienvenue

Au cours des dernières années, le langage de programmation statistique R est devenu une partie intégrante des programmes des cours d'économétrie. Nous avons régulièrement constaté qu'une grande partie des étudiants, en particulier dans nos cours d'introduction à l'économétrie de premier cycle, n'ont jamais été exposés à aucun langage de programmation et ont donc des difficultés à s'engager seuls dans l'apprentissage de R. Avec peu d'expérience en statistiques et en économétrie, il est naturel que les débutants aient du mal à comprendre les avantages d'avoir des compétences en R pour apprendre et appliquer l'économétrie. Celles-ci incluent en particulier la capacité de mener, de documenter et de communiquer des études empiriques et d'avoir les installations nécessaires pour programmer des études de simulation, ce qui est utile, par exemple, pour comprendre et valider des théorèmes qui ne sont généralement pas facilement saisis en ruminant simplement sur des formules. En tant qu'économistes appliqués et économètres, tous ces derniers sont des capacités que nous apprécions et que nous souhaitons partager.

Il s'agit d'un script interactif dans le style d'un rapport de recherche reproductible et permet aux étudiants non seulement d'apprendre comment les résultats d'études de cas peuvent être reproduits avec R, mais renforce également leur capacité à utiliser les compétences nouvellement acquises dans d'autres applications empiriques.

# Part I

# I-Débuter avec R

# Chapter 1

# Introduction

R est un langage de programmation créé par les staticiens Ross Ihaka et Robert Gentleman. C'est un langage dédié aux statistiques, représentations graphiques, ainsi que tout ce qui se rattache au traitement et manipulation de données. C'est aussi un logiciel à accès libre (open-source) disponible sous la licence publique générale GNU (GNU General Public License).

Principalement écrit en C et Fortran, R est une implémentation du langage S qui supporte plusieurs paradigmes de programmation tel que : procédural, orienté objet, fonctionnel, réflexif, impératif, tableau.

Depuis sa création, le langage a fortement évolué grâce à la contribution de sa communauté d'utilisateurs notamment par la publications de packages et de tutoriels. Cette évolution a permis d'étendre les fonctionnalités de ce langage à la rédaction d'ouvrage avec bookdown, d'article et de présentations avec R markdown, à la représentation graphique avec ggplot2 etc. Rien que sur le CRAN (Comprehensive R Archive Network) on peut trouver plus de 18000 packages.

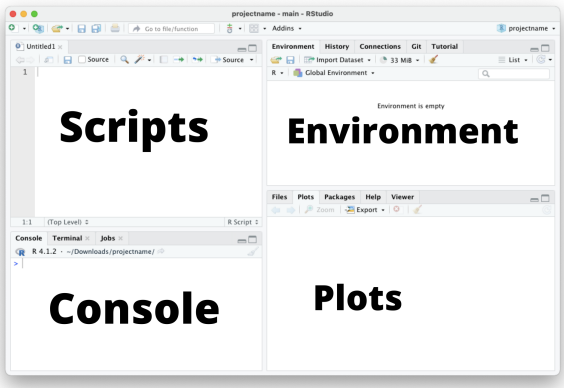
Dans le chapitre suivant nous allons voir les bases de la programmation avec R en utilisant le logiciel RStudio.

# Chapter 2

# les bases

A l'instar des autres langages, R a besoin d'un environnement de développement intégré (IDE) pour être utilisé. En plus d'une console interactive, l'IDE propose plusieurs fonctionnalités. Nous en verrons quelques-unes plus tard.
R est intégré par plusieurs logiciels tels que R lui-même, IntelliJ IDE, Rcode etc. Tout au long de cet ouvrage nous n'utiliserons que RStudio qui est l'un des IDE les plus célèbres de R.

## 2.1 Interface




L'interface RStudio se présente ainsi par défaut. Nous pouvons voir une console, un environnement de travail, une partie pour les scripts et une autre pour la visualisation des graphiques.

- La *console* peut servir à écrire une ligne d'instruction qui sera exécutée en appuyant sur Entrer.

- La partie dédiée au *scripts* permet d'écrire un ensemble de lignes de codes qui peuvent être exécutées par ordre voulu par l'utilisateur. En plus des scripts, on peut y visualiser nos tableaux de données qui sont décrits dans la section **??**.

- Les graphes peuvent être visibles dans la partie *Plots* de notre interface. Cette partie est un panneau contenant les onglets *Viewer* pour les pages html, *Files* pour naviguer dans les fichiers, *Packages* pour gérer les packages installés et *Help* pour chercher de l'aide.

- La dernière partie c'est à dire *Environment* est consacrée à la gestion de l'environnement de travail. Elle permet de voir les variables créés lors de notre session mais aussi d'avoir une idée sur leur structure

## 2.2 Premiers codes

Et si on écrivait notre premier code ? Dans la section précédente nous avons présenté brièvement l'interface de RStudio, place maintenant à notre première instruction. Commencez par effectuer une petite opération d'addition (`2+3`) sur votre console puis appuyez sur Entrer.

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

Le résultat obtenu est tout naturellement `5`. Vous constatez que `[1]` précède le résultat de l'opération, en effet l'affichage se fait par défaut sous forme d'une liste**??**.
La console peut être utilisée comme une calculatrice et supporte toutes les opérations arithmétiques telles que la soustraction(`-`), l'addition(`+`), la multiplication(`*`), la division décimale(`/`), la division entière(`%/%`), le modulo[1] (`%%`). Le symbole (`#`) sert à écrire une ligne de commentaire.

```
1+2 #addition

1-2 #Soustraction

1/2 #Division decimale

1%/%2 #Division entiere
```

[1] modulo : Cet opérateur renvoi le reste de la division entre deux nombres

```r
1%%2 #Modulo
```

On peut aussi effectuer des assignations sans déclarer les variables au préalable comme l'indique le code suivant.

```r
x = 1 #affection de la valeur 1 à x
y <- 2 #affection de la valeur 2 à y
x+y #somme de x et y (1+2)
#> [1] 3
```

## 2.3 Données sur R

Il existe 6 principaux types simples de données sur R sont : *logical*, *integer*, *double*, *complex*, *character*, *raw*.
Il arrive qu'une structure de données se compose de types simples données, c'est ce que nous allons étudier dans cette section.

### 2.3.1 Vecteur

#### Définition

Le vecteur est un objet de base de R qui correspond à une liste d'éléments. Ses propriétés fondamentales sont :

- Dimension unitaire (les vecteurs sont unidimensionnel)
- Éléments de même type (Toutes les valeurs contenues dans un vecteur sont de même type)
- Longueur égale au nombre d'éléments

#### Création

La fonction la plus classique pour créer un vecteur est `c(...)`. Elle prend comme argument les éléments du vecteur. Dans le code suivant, nous allons créer un vecteur contenant les valeurs de 1 jusqu'à 5 puis nous allons le nommer `myvector`

```r
myvector <- c(1,2,3,4,5) #Création
myvector #Affichage
#> [1] 1 2 3 4 5
```

Maintenant que nous avons un vecteur, il est naturel de se demander comment accéder aux éléments de ce dernier. Facile ! Il suffit de mettre entre crochet (`[]`), juste après le nom de votre vecteur, l'indice de l'élément voulu sachant que sur R le comptage commence par 1 au lieu de 0. Par exemple, dans la cellule suivante, le code permet d'afficher le quatrième élément c'est-à-dire celui qui a pour indice 4 de `myvector`.

```
myvector[4]
#> [1] 4
```

Vous pouvez aussi supprimer un élément d'un vecteur en essayant de l'afficher avec l'opposé de son indice. Supprimons le premier élément de `myvector` !

```
myvector[-1]
#> [1] 2 3 4 5
```

Il se peut qu'on veuille créer une séquence de valeurs avec un pas spécifié. Un exemple concret c'est de vouloir créer un vecteur nommé `evenVector` contenant tous les nombres pairs compris entre 0 et 100. L'utilisation de la fonction `c()` rendrait le travail fastidieux. La fonction `seq()` est plus adaptée à notre situation. Elle prend comme argument from(le début de la séquence), to(la fin de la séquence), by(le pas de la séquence), etc. Pour en savoir plus vous pouvez exécuter la commande `?seq()`.

```
evenVector <- seq(from = 0, to = 100, by = 2) #Création
evenVector #Affichage
#>  [1]   0   2   4   6   8  10  12  14  16  18  20  22  24  26
#> [15]  28  30  32  34  36  38  40  42  44  46  48  50  52  54
#> [29]  56  58  60  62  64  66  68  70  72  74  76  78  80  82
#> [43]  84  86  88  90  92  94  96  98 100
```

Si le pas de la séquence est de 1, on peut utiliser à la place de `seq()` l'opérateur `:` de premier terme le début de la séquence et de second terme la fin de la séquence. L'exemple qui suit permet de créer le vecteur `myvector` contenant tous les entiers de 1 à 5.

```
myvector <- 1:5
myvector
#> [1] 1 2 3 4 5
```

Il est possible de créer un vecteur d'éléments répétitifs avec la fonction `rep()`. Supposons que nous voulons créer le vecteur `repvector` contenant 5 fois de suite tous les entiers de 1 à 10, nous allons donner en premier argument à la fonction `rep()` l'objet à répéter (`1:10`) et comme second argument le nombre de répétitions(`5`).

```
repvector <- rep(1:10,5)
repvector
#>  [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8
#> [19]  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6
#> [37]  7  8  9 10  1  2  3  4  5  6  7  8  9 10
```

### 2.3.2 Facteur

#### Définition

Le facteur (*factor*) est un vecteur de valeurs d'une variable catégorielle. Très souvevent, les variables qualitatives sont catégorielles c'est le cas du sexe(Homme, Femme), des questions directes(Oui, Non), etc.C'est d'ailleurs la raison de l'existance de cet objet sur R qui est très utiles dans certaines representations graphiques.Le caractère principal de *factor* est qu'il dispose de niveaux appelés `levels`. Ces derniers sont uniques et peuvent avoir des valeurs qui ne sont pas contenus par le facteur.

#### Création

Pour créer un facteur, on commence par créer un vecteur puis avec la fonction `factor()` nous pouvons le transformer en objet de type facteur. Par défaut, les niveaux des facteurs sont les modalités prises par le vecteur. Pour modifier les niveaux on utilise l'argument `levels` de la fonction `factor()` pour spécifier notre vecteur de niveaux. On se propose de transformer en facteur le vecteur `animal` de modalités *chat*, *souris*, *chien* en un facteur de niveaux *chat*, *souris*, *chien* et *rat*.

```
animal <- c("souris","souris","chien","chat","chien","chat","souris","chat","chat","chien")
myfactor <- factor(animal,levels = c('chat','souris','chien','rat'))
myfactor
#>  [1] souris souris chien  chat   chien  chat   souris chat
#>  [9] chat   chien
#> Levels: chat souris chien rat
```

### 2.3.3 Liste

#### Définition

Une liste est un objet pouvant contenir des éléments de tous types. L'homogénéité du type des éléments n'est pas obligatoire dans une liste c'est à dire qu'elle peut contenir des listes, des vecteurs, des matrices, des

fonctions etc. On peut nommer les éléments d'une liste lors de sa création en effectuant des affections.

### Création

La création d'une liste se fait avec la fonction `list()` qui prend en argument les éléments à concaténer.

```
mylist = list(num = c(1,2,3), char = 'character')
mylist
#> $num
#> [1] 1 2 3
#>
#> $char
#> [1] "character"
```

On peut accéder à un élément par son nom en utilisant le symbole `$` (`malist$nomElement`). L'accès à l'élément `num` de `mylist` peut se faire de la manière suivante :

```
mylist$num
#> [1] 1 2 3
```

Un autre moyen d'accéder à un élément d'une liste c'est par son indice mis entre deux crochets (`[[index]]`) juste après le nom de la liste. On peut reprendre l'accès à l'élément `num` par indexation.

```
mylist[[1]]
#> [1] 1 2 3
```

On peut modifier l'élément `num` de mylist en lui affectant une nouvelle valeur.

```
mylist$num = 1:10
mylist$num
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

L'ajout d'un nouvel élément dans `mylist` peut aussi se faire facilement. Si on se propose d'ajouter un élément nommé `logical` qui reçoit initialement `TRUE` on peut procéder ainsi :

```
mylist$logical = TRUE
mylist
#> $num
```

```
#>  [1]  1  2  3  4  5  6  7  8  9 10
#>
#> $char
#> [1] "character"
#>
#> $logical
#> [1] TRUE
```

### 2.3.4 Matrice

#### Définition

Une matrice est un un tableau dont les colonnes sont des vecteurs de même type et de même taille. Autrement dit, la matrice est un objet de deux dimensions dont tous les éléments sont de type homogène. R ne considère pas un vecteur comme une matrice colonne ou ligne, ce sont deux types de structures différentes.

#### Création

Une matrice colonne se crée avec la fonction `cbind()` et la matrice ligne par `rbind()`. Pour créer une matrice de plusieurs colonnes on utilise la fonction `matrix()`. Vous pouvez avoir une documentation complète de ces fonctions en exécutant la commande faisant précéder d'un point d'interrogation le nom de votre fonction ( *Exemple* : `?cbind()`) dans votre console.

```
#matrice colonne
colMatrix <- cbind(1:5)
colMatrix
#>      [,1]
#> [1,]    1
#> [2,]    2
#> [3,]    3
#> [4,]    4
#> [5,]    5
```

```
#matrice ligne
rowMatrix <- rbind(1:5)
rowMatrix
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    2    3    4    5
```

```r
#matrice
Matrix <- matrix(c(x= (1:5), y = rep(1,5)),nrow = 5)
Matrix
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    2    1
#> [3,]    3    1
#> [4,]    4    1
#> [5,]    5    1
```

On peut accéder aux éléments de la matrice par leurs indices. Comme on le fait en maths, il faut d'abord mettre le numéro de la ligne puis le numéro de la colonne séparée par une virgule. On peut afficher une ligne (respectivement une colonne )toute entière en spécifiant seulement l'indice de la ligne (respectivement la colonne).

```r
# Accès à l'élément de la ligne 4 et de la colonne 1
Matrix[4,1]
#> [1] 4
```

```r
# Accès à  la ligne 2
Matrix[2,]
#> [1] 2 1
```

```r
# Accès à la colonne 3
Matrix[,2]
#> [1] 1 1 1 1 1
```

### 2.3.5 Tableau de données ou Data frame

#### Définition

Un tableau de données (data frame) est comme la matrice, un objet de deux dimensions sauf qu'il peut contenir des colonnes de types différents. Chaque colonne doit contenir des éléments de même type. La data.frame est un objet très utilisé sur R et ce sera le cas dans les chapitres suivants de ce livre.

#### Création

La fonction `data.frame()` permet de créer un tableau de données. Elle prend en argument des vecteurs de même longueur. Il en existe d'autres arguments pour cette fonction, pour en savoir plus vous pouvez exécuter `?data.frame()`.

```r
x = c(12,67,13)
y = c('A','B','C')
tableau = data.frame(x,y)
tableau
#>    x y
#> 1 12 A
#> 2 67 B
#> 3 13 C
```

L'accès à un élément peut se faire de la même manière qu'avec les matrices. Pour accéder à une colonne par son nom on utilise le symbole comme dans la section liste. Les codes suivants permettent d'accéder à l'élément "B" du `tableau` de plusieurs façons.

```r
#Indexiation
tableau[2,2]
#> [1] "B"
```

```r
#Par le nom de la colonne
tableau$y[2]
#> [1] "B"
```

```r
tableau[['y']][2]
#> [1] "B"
```

## 2.4 Les boucles et conditions

### 2.4.1 Les boucles

Les boucles permettent de gérer des instructions répétitives. Sur R il est plus pratique d'utiliser la vectorisation, mais pour débuter, les boucles feront l'affaire.

#### for

La boucle for permet d'itérer sur un vecteur de longueur connu d'avance. La syntaxe est la suivante :

```r
for (variable in vector) {
  statements
}
```

Un exemple de création d'un vecteur de 5 éléments contenant les carrés des nombre compris entre 1 et 5

```
#Initialisation
carrevector = 0
#Boucle
for (i in 1:5) {
  carrevector[i]=i^2
}
#affichage
carrevector
#> [1]  1  4  9 16 25
```

### While

Si le nombre d'itérations n'est pas connu d'avance alors que la condition d'arrêt si, il est préférable d'utiliser la boucle while. Elle permet de répéter une instruction tant qu'une condition est satisfaite. la syntaxe est la suivante :

```
while (condition) {
  statements
}
```

Avec la boucle while on peut chercher le premier élément supérieur à 10 dans `carrevector`. Les deux conditions d'arrêts sont alors : l'élément(`x`) supérieur à 10 et(`&`) l'indice(`j`) supérieur à la longueur du vecteur(`5`).

```
# Initialisation
j = 1
# Boucle
while (carrevector[j]<=10 & j<= 5){
  j=j+1
}
x=carrevector[j]
# Affichage
x
#> [1] 16
```

## 2.4.2 Les conditions

Elles permettent d'exécuter des instructions si une ou des conditions sont satisfaites. la syntaxe la plus simple est :

```
if (condition) {
  #statements
}else{
```

```
  #statements
}
```

**Exemple** :

```
num = 10
if (num > 0) {
  print('Ce nombre est positif')
}else{
  print('Ce nombre est negatif')
}
#> [1] "Ce nombre est positif"
```

## 2.5 Les fonctions

Si dans un projet vous aurez à utiliser plusieurs fois un bloc d'instructions, l'idéal c'est de créer des fonctions. Pour cela, on a besoin de `function()` que l'on assigne au nom de notre fonction. Les fonctions peuvent retourner une valeur ou bien exécuter seulement un ensemble d'instructions.

```
mafonction  <- function(arguments) {
  instructions
  return(valuer)
}
```

Pour créer la fonction `addition` qui somme deux éléments `x` et `y` on procède ainsi :

```
addition <- function(x,y){
  return(x+y)
}
#Ou bien
addition <- function(x,y){
  x+y
}
```

Pour utiliser la fonction il suffit de l'appeler en indiquant les valeurs de ses arguments entre parenthèses.

```
# somme de 10 et de 4
addition(10,4)
#> [1] 14
```

## Liens utils

**?**
https://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf

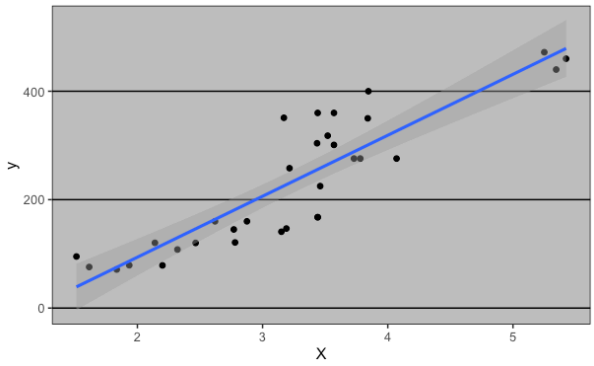
# Part II

# II-Regressions

# Chapter 3

# La régression linéaire simple

## 3.1 Introduction




La régression linéaire simple est une méthode statistique permettant de trouver une relation linéaire entre une variable explicative $X$ et une variable à expliquer $y$. Ce modèle consiste à considérer $y$ comme une fonction affine de $X$. En d'autre terme, la régression linéaire a pour but de trouver une droite ajustée au nuage de points de $y$ en fonction de $X$.
Dans ce chapitres nous allons voir en détail le modèle linéaire simple ainsi que son application avec R. Nous utiliserons les données de income disponible sur github. Vous pouvez télécharger le jeux de données avec la fonction `read_csv()` disponible dans le package `readr` comme suit :

```
library(readr)
income = read_csv("https://github.com/AODiakite/r4econometrics/blob/master/Data/income.data.csv")
```

Table 3.1: Données pour la régression linéaire simple : income( niveau de revenu par 10 000 dollars ), happiness(score du bonheur entre 0 et 10), nombre d'observations(498)

| income | happiness |
|---|---|
| 3.862647 | 2.314489 |
| 4.979381 | 3.433490 |
| 4.923957 | 4.599373 |
| 3.214372 | 2.791114 |
| 7.196409 | 5.596398 |
| 3.729643 | 2.458556 |
| 4.674517 | 3.192992 |
| 4.498104 | 1.907137 |
| 3.121631 | 2.942450 |
| 4.639914 | 3.737942 |

## 3.2 Modélisation mathématique

L'ajustement affine de $y$ par $X$ stipule que que y peut s'écrire comme équation d'une droite :

$$y = \beta_0 + \beta_1 X \tag{3.1}$$

- $y(y_1, y_2, \dots, y_n)$ : variable à expliquer, variable dépendante, variable endogène, variable réponse
- $X(x_1, x_2, \dots, x_n)$ : variable explicative,variable exogène, Variable régresseur
- $\beta_0$ : l'ordonnée à l'origine, coefficient inconnu
- $\beta_1$ : la pente de la droite, coefficient inconnu

En réalité sauf dans le cas d'un modèle parfait, la liaison linéaire (**??**) entre y et X est perturbée par un bruit $\epsilon$. l'équation du modèle devient alors :

$$y = \beta_0 + \beta_1 X + \epsilon \tag{3.2}$$

La variable aléatoire $\epsilon$ est indépendante de $X$ et est supposée suivre une loi normale de moyenne 0 et d'écart type $\sigma$ :

$$(\epsilon_1, \epsilon_2, \dots, \epsilon_n) = \epsilon \sim \mathcal{N}(0, \sigma^2)$$