TP R 1: Introduction

Cours de Programmation

Vittorio Perduca, Master 1 Mathématiques et Applications

UFR Math-Info, Université Paris Descartes, septembre 2021

Table des matières

1.1 Présentation

R est un langage dédié aux statistiques développé à partir des années '90 à l'University of Auckland, Nouvelle-Zélande. Son implémentation principale est le logiciel open source (gratuit et modifiable) R (cran.r-project.org). R est normalement utilisé à partir de l'interface graphique et de développement RStudio (www.rstudio.com).

R est un langage interprété (comme Python) qu'on utilise à partir d'une ligne de commande :

```
print('Hello world!')
```

```
## [1] "Hello world!"
```

Alternativement, on peut exécuter un script, c'est à dire une suite de commandes qui se trouvent dans un fichier dont l'extension est `.R`.

La communauté des utilisateurs `R` est très active dans le monde scientifique (statistiques, science des données, bioinformatique, sciences sociales,...) et de plus en plus dans les entreprises. L'un des avantages de `R` est la richesse de *packages* développés par les utilisateurs et développeurs qu'on peut installer pour augmenter ses capacités dans des domaines très variés des statistiques.

Par ailleurs, `R` dispose d'une documentation très complète. On peut accéder à l'aide en tapant `?` suivi par la *fonction* sur la quelle on souhaite se renseigner :

```
?rnorm
```

On peut aussi trouver énormément d'informations sur le web : en cherchant sur Google un problème lié à une tâche `R` on trouve presque toujours une réponse (souvent dans des threads ouverts sur le site Cross Validated).

Le but de cette introduction tutorielle est de vous apprendre l'utilisation élémentaire de `R`. Quoique les objets et commandes de bases y soient introduits, cette introduction n'est pas une référence complète au langage `R`, vous serez donc amenés à utiliser l'aide et à chercher des informations sur le web. Avant de passer aux exercices, vous êtes encouragés à taper les commandes et à en comprendre le résultat : la meilleur façon d'apprendre un langage est d'écrire et débugger beaucoup de lignes de code !

1.2 Prise en main de `RStudio`

La fenêtre de `RStudio` se divise généralement en quatre sous-fenêtres : en partant de haut à gauche et an allant en sens horaire on trouve :

1. un éditeur de texte pour les scripts
2. l'espace de travail ou d'historique de commandes
3. le navigateur de fichiers, graphiques, packages, documentations
4. la console `R`, c'est à dire la ligne de commande.

Typiquement on tape le code dans l'éditeur et on l'exécute ensuite dans la console. Pour cela on place le curseur dans la ligne qu'on veut exécuter et on envoie la commande à la console à l'aide de la combinaison `cmd Retour` dans OS et `ctrl Retour` sous Windows.

1.3 Packages

Pour installer un package :

```
install.packages("dplyr")
#on telecharge et installe le package dplyr utilisé dans la manipulationd des données
```

Une fois un package installé, il faudra le charger en mémoire à chaque fois une nouvelle session est ouverte :

```r
library(dplyr) #sans guillemets!
#require(dplyr) #fonction équivalente
```

1.4 Références

Une excellente référence en français est le livre de Vincent Goulet *Introduction à la programmation en R*, qu'on peut télécharger gratuitement sur le site du CRAN :

https://cran.r-project.org/doc/contrib/Goulet_introduction_programmation_R.pdf

Une référence complète est l'introduction officielle sur le site du CRAN :

https://cran.r-project.org/doc/manuals/R-intro.html

1.5 Version

Version beta, merci de signaler toute erreur à vittorio.perduca@u-paris.fr

2. Bases du langage

2.1 Commandes

Il y a deux type de commandes en `R` : les expressions et les affectations.

Expression

```
cos(pi)
```

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

Affectations et expressions

```
x <- 1+2 # ou x=1+2
x
```

```
## [1] 3
```

```
y = 4
x == y
```

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

A l'aide de `;` on peut taper deux commandes sur la même ligne avant leur exécution :

```
e <- exp(1); log(e)
```

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

Quelques exemples d'opérateurs arithmétiques et booléens :

```
3*4; 12/3; 2^3; sqrt(16)
```

```
## [1] 12
```

```
## [1] 4
```

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

```
## [1] 4
```

```
1==2; 1!=1
```

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

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

```r
FALSE & TRUE # et
```

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

```r
FALSE | TRUE # ou
```

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

2.2 Modes, longueurs et classes de données

Dans R, tout est un *objet*. Le *mode* spécifie ce qu'un objet peut contenir. Les modes principaux sont :

- `numeric` : nombres réels
- `character` : chaînes de caractères
- `logical` : valeurs logiques vrai/faux
- `list` : liste, collection d'objets
- `function` : fonction

Les objets de mode `numeric`, `character` et `logical`, sont des objets *simples* qui peuvent contenir des données d'un seul type. Au contraire, les objets de mode `list` sont des objets spéciaux qui peuvent contenir d'autres objets.

On peut accéder au mode d'un objet avec la fonction `mode()` :

```r
age=c(33,28, 33) # La fonction de concatenation c() permet de créer des vecteurs
mode(age)
```

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

```r
noms <- c('Daniel', 'Jehanne', 'Romain')
mode(noms)
```

```
## [1] "character"
```

```r
ma.liste <- list(Noms=noms, Age=age)
mode(ma.liste)
```

```
## [1] "list"
```

```r
mode(is.integer(pi))
```

```
## [1] "logical"
```

```r
mode(mode)
```

```
## [1] "function"
```

A part le mode, un objet a aussi une *longueur*, définie comme le nombre d'éléments qu'il contient :

```
length(age)
```

```
## [1] 3
```

```
length(noms)
```

```
## [1] 3
```

```
length(ma.liste)
```

```
## [1] 2
```

La *classe* d'un objet spécifie son comportement et donc sa façon d'interagir avec opérations et fonctions. Un exemple important sont les *data frame* : des liste spéciales dont les éléments ont tous la même longueur. La classe d'un data frame est différente de celle des listes génériques et les data frame ont un système d'indiçage qui n'existe pas pour les autres listes :

```
class(ma.liste)
```

```
## [1] "list"
```

```
mon.data.frame=data.frame(noms,age)
mode(mon.data.frame)
```

```
## [1] "list"
```

```
class(mon.data.frame)
```

```
## [1] "data.frame"
```

```
mon.data.frame[1,2] # pour extraire le 1e élément de la 2ème "colonne""
```

```
## [1] 33
```

```
# Essayer la commande suivante:
# ma.liste[1,2]
```

Un objet spécial est la valeur manquante `NA`. Par défaut, son mode est `logical`, cependant `NA` n'est ni `TRUE` ni `FALSE`. Pour tester si une valeur est manquante on utilisera la fonction `is.na()` :

```
NA==NA
```

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

```
is.na(NA)
```

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

```r
is.na(mean(c(1,4,NA)))
```

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

2.3 Vecteurs et matrices

2.3.1 Vecteurs

En R l'unité de base dans les calculs est le *vecteur* (un scalaire est considéré comme un vecteur de longueur un). La fonction la plus utilisée pour créer un vecteur est la concaténation :

```
prix <- c(150, 162, 155, 157); prix
```

```
## [1] 150 162 155 157
```

L'**indiçage** se fait par les crochets :

```
prix[1] # Le premier indice est toujours 1!!
```

```
## [1] 150
```

```
prix[c(1,3)]
```

```
## [1] 150 155
```

```
prix[-(1:2)] # pour extraire tous les éléments sauf le 1e et le 2e
```

```
## [1] 155 157
```

On peut aussi utiliser un vecteur d'indiçage booléen, les éléments extraits sont bien évidement ceux correspondants aux valeurs `TRUE`. Par exemple pour extraire les prix supérieurs à 156 :

```
prix > 156 # le vecteur booléen
```

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

```
prix[prix>156]
```

```
## [1] 162 157
```

Une alternative est donnée par la fonction `which()`qui rend les indices dont les éléments vérifient une condition logique :

```
which(prix>155)
```

```
## [1] 2 4
```

```
prix[which(prix>156)]
```

```
## [1] 162 157
```

On peut utiliser l'indiçage pour changer un élément :

```
prix[1] <- 0; prix
```

```
## [1]   0 162 155 157
```

Il est possible de donner des étiquettes aux éléments d'un vecteur et d'extraire des éléments sur la base de celles-ci :

```
names(prix) # NULL est un objet spécial de mode NULL qui se lit "pas de contenant"
```

```
## NULL
```

```
names(prix) <- c('model.1', 'model.2', 'model.3', 'model.4')
prix
```

```
## model.1 model.2 model.3 model.4
##       0     162     155     157
```

```
prix['model.3']
```

```
## model.3
##     155
```

Dans un vecteur, tous les éléments doivent avoir le même mode :

```
x <- c(1,2,'a', 'b'); x
```

```
## [1] "1" "2" "a" "b"
```

```
mode(x)
```

```
## [1] "character"
```

Pour générer le vecteur des n premiers entiers on utilise la syntaxe `1:n`

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
2:6
```

```
## [1] 2 3 4 5 6
```

Pour générer des suites plus générales on utilise la fonction `seq()` :

```
seq(from=2, to=20, by=2) # ou plus simplement seq(2,20,2)
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

On peut créer un vecteur d'éléments répétés avec `rep()` :

```r
rep(1,len=3) # même chôse que rep(1,3)
```

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

```r
rep(NA,4)
```

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

2.3.2 Matrices

Une matrice est un vecteur avec un attribut `dim` de longueur deux. Tous les éléments d'une matrice ont donc le même mode. Pour créer une matrice :

```r
M <- matrix(2:7, nrow=2, ncol=3); M
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    3    5    7
```

```r
matrix(2:7, nrow=2, ncol=3, byrow=TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]    2    3    4
## [2,]    5    6    7
```

Par défaut `matrix()` remplit la nouvelle matrice par colonne. L'indiçage se fait avec les crochets :

```r
M[2,] # 2e ligne
```

```
## [1] 3 5 7
```

```r
M[,3] # 3e colonne
```

```
## [1] 6 7
```

```r
M[2,3]
```

```
## [1] 7
```

```r
M[3]
```

```
## [1] 4
```

```r
M[,-2] # pour extraire toutes les colonnes sauf la 2e
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    3    7
```

Pour fusionner verticalement (horizontalement) deux matrices on utilise `rbind()` (resp. `cbind()`) :

```
cbind(M,-M)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    2    4    6   -2   -4   -6
## [2,]    3    5    7   -3   -5   -7
```

```
rbind(M,2*M)
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    3    5    7
## [3,]    4    8   12
## [4,]    6   10   14
```

2.3.3 Opérations sur vecteurs et matrices numériques

Elément par élément :

```
v <- c(3,4,1,6)
v + 2
```

```
## [1] 5 6 3 8
```

```
v * 2
```

```
## [1]  6  8  2 12
```

```
v * v
```

```
## [1]  9 16  1 36
```

```
v/2
```

```
## [1] 1.5 2.0 0.5 3.0
```

```
v/v
```

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

```
v + v^2
```

```
## [1] 12 20  2 42
```

```
sqrt(M)
```

```
##          [,1]     [,2]     [,3]
## [1,] 1.414214 2.000000 2.449490
## [2,] 1.732051 2.236068 2.645751
```

```
M * M
```

```
##      [,1] [,2] [,3]
## [1,]    4   16   36
## [2,]    9   25   49
```

```
# Essayer la commande suivante:
# M + v
```

Transposée, produit matriciel, inverse :

```
t(M)
```

```
##      [,1] [,2]
## [1,]    2    3
## [2,]    4    5
## [3,]    6    7
```

```
N <- M[,-3]
N %*% diag(1,2)
```

```
##      [,1] [,2]
## [1,]    2    4
## [2,]    3    5
```

```
# diag(1,2) construit la matrice diagonale de dimension 2x2 dont tous les
# éléments de la diagonale sont égaux à 1
solve(N)
```

```
##      [,1] [,2]
## [1,] -2.5    2
## [2,]  1.5   -1
```

```
solve(N) %*% N
```

```
##      [,1]         [,2]
## [1,]    1 1.776357e-15
## [2,]    0 1.000000e+00
```

Le transposé d'un vecteur est une matrice-ligne :

```
V <- t(v)
dim(V)
```

```
## [1] 1 4
```

```
t(V)
```

```
##      [,1]
## [1,]    3
## [2,]    4
## [3,]    1
## [4,]    6
```

Faire attention aux exemples suivants :

```
v %*% t(v)  # v est consideré comme un vecteur-colonne!
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    9   12    3   18
## [2,]   12   16    4   24
## [3,]    3    4    1    6
## [4,]   18   24    6   36
```

```
t(v) %*% v  # idem
```

```
##      [,1]
## [1,]   62
```

```
diag(1,4) %*% v # idem
```

```
##      [,1]
## [1,]    3
## [2,]    4
## [3,]    1
## [4,]    6
```

```
v %*% v     # v est consideré un vecteur-ligne et un vecteur-colonne
```

```
##      [,1]
## [1,]   62
```

2.4 Facteurs

Un facteur est un vecteur utilisé pour contenir une variable qualitative, c'est à dire une variable à valeurs discrètes. Ses valeurs, ou catégories ou encore modalités, sont appelées les `levels` en `R`.

```
ville <- c('paris', 'lyon', 'lyon', 'paris', 'nantes')
fact.ville <- as.factor(ville); fact.ville
```

```
## [1] paris  lyon   lyon   paris  nantes
## Levels: lyon nantes paris
```

```
class(fact.ville)
```

```
## [1] "factor"
```

```
levels(fact.ville)
```

```
## [1] "lyon"   "nantes" "paris"
```

Un facteur a le mode `numeric` : en effet ses éléments sont stockés comme les entiers énumérant les différentes modalités :

```
mode(fact.ville)
```

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

```
as.numeric(fact.ville)
```

```
## [1] 3 1 1 3 2
```

2.5 Listes

Les listes sont des vecteurs spéciaux qui peuvent stocker des éléments de n'importe quelle mode (y comprises d'autres listes).

Comme tout autre vecteur, une liste est indicée par l'opérateur `[ ]`. Cependant, cela retourne une liste contenant l'élément souhaité :

```
ma.liste[1]
```

```
## $Noms
## [1] "Daniel"  "Jehanne" "Romain"
```

```
mode(ma.liste[1])
```

```
## [1] "list"
```

Pour obtenir directement l'élément, on utilise donc l'opérateur `[[ ]]` ou l'opérateur `$` suivi par le nom de l'élément (si disponible) :

```
ma.liste[[1]]
```

```
## [1] "Daniel"  "Jehanne" "Romain"
```

```
ma.liste$age
```

```
## NULL
```

Les éléments d'une liste peuvent avoir des longueur différentes :

```
ma.liste$ville <- ville
ma.liste
```

```
## $Noms
## [1] "Daniel"  "Jehanne" "Romain"
##
## $Age
## [1] 33 28 33
##
## $ville
## [1] "paris"  "lyon"   "lyon"   "paris"  "nantes"
```

2.6 Data frames

Un des conteneurs de données le plus utilisé est le data frame, une liste spéciale de classe `data.frame` dont tous les éléments ont la même longueur. Pour cette raison, un data frame est représenté sous forme d'un tableau à deux dimensions dont les colonnes sont ses éléments. Typiquement, dans un data frame les colonnes représentent les **variables** et les ligne les **observations**. Contrairement aux matrices, les éléments d'un data frame peuvent avoir des modes différents.

```r
id <- c('id.453', 'id.452', 'id.455', 'id.459', 'id.458', 'id.456', 'id.450', 'id.451')
age <- c(19, 45, 67, 53, 17, 30, 27, 35)
fumeur <- c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, TRUE, TRUE)
sex <- c('f', 'f', 'h', 'h', 'f', 'h', 'f', 'f')
ma.bd <- data.frame(Id=id, Age=age, Fumeur=fumeur, Sex=sex); ma.bd
```

```
##       Id Age Fumeur Sex
## 1 id.453  19   TRUE   f
## 2 id.452  45  FALSE   f
## 3 id.455  67   TRUE   h
## 4 id.459  53   TRUE   h
## 5 id.458  17  FALSE   f
## 6 id.456  30   TRUE   h
## 7 id.450  27   TRUE   f
## 8 id.451  35   TRUE   f
```

```r
dim(ma.bd); nrow(ma.bd); ncol(ma.bd)
```

```
## [1] 8 4
```

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

```
## [1] 4
```

```r
names(ma.bd)
```

```
## [1] "Id"     "Age"    "Fumeur" "Sex"
```

Un data frame étant une liste, on pourra extraire une colonne à l'aide de l'opérateur `$` précédé par le nom du data frame et suivi par le nom de la colonne (ou variable), ou utiliser l'opérateur `[ ]`

```r
ma.bd$Sex # une colonne de caractères est transformée automatiquement en facteur
```

```
## [1] "f" "f" "h" "h" "f" "h" "f" "f"
```

```r
ma.bd[,2]
```

```
## [1] 19 45 67 53 17 30 27 35
```

```r
ma.bd$Age[ma.bd$Fumeur==FALSE]
```

```
## [1] 45 17
```

Les colonnes sont directement accessibles dans l'espace de travail (sans devoir taper le nom du data frame et le `$`) après avoir *attaché* le data frame :

```
attach(ma.bd)
Age
```

```
## [1] 19 45 67 53 17 30 27 35
```

Pour afficher seulement les six premières lignes :

```
head(ma.bd)
```

```
##       Id Age Fumeur Sex
## 1 id.453  19   TRUE   f
## 2 id.452  45  FALSE   f
## 3 id.455  67   TRUE   h
## 4 id.459  53   TRUE   h
## 5 id.458  17  FALSE   f
## 6 id.456  30   TRUE   h
```

De façon similaire, `tail()` permet de créer un data frame avec les six dernières colonnes.

2.7 Fonctions définies par l'utilisateur

Exemple :

```
ma.fonction <- function(x,y=10){ # la valeur par défaut de y est 10
  z=x-y
  return(z)
}
ma.fonction(2)
```

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

```
ma.fonction(2,4)
```

```
## [1] -2
```

```
ma.fonction(y=1, x=4)
```

```
## [1] 3
```

Toute variable définie dans une fonction est *locale* et n'apparaît pas dans l'espace de travail : essayer d'exécuter

```
z
```

2.8 Structures de côntrole

Exemple traitement conditionnel :

```
x <- runif(1) # valeur tirée selon une loi uniforme dans [0,1]
if(x<0.5){
  print('gagné')
}else{
  print('perdu')
}
```

```
## [1] "gagné"
```

Exemple traitement itératif, boucle `for` :

```
y=rep(NA,5)
for(i in 1:5){
  y[i] <- exp(i)
}
y
```

```
## [1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

Il est important de noter que souvent il est possible et (préférable !) **d'éviter les boucles**. En effet, les boucles ne sont pas très efficaces et il faut essayer de les remplacer par des opérations sur vecteurs :

```r
exp(1:5)
```

```
## [1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

Exemple traitement itératif, boucle `while` :

```r
# Pile ou face, on gagné si on a face, on continue à jouer jusqu'à gagner
x <- sample(c('pile','face'), 1, prob = c(0.8,0.2)) # tirage aléatoire d'une pièce de monnaie biaisée
if(x == 'face'){
  print(paste(x, ', gagné', sep=''))
}else{
  while(x != 'face'){ # l'expression suivante est executée tant que la condition entre () est vraie
    print(paste(x, ", perdu", sep=''))
    x <- sample(c('pile','face'),1, prob = c(0.8,0.2))
  }
print(paste(x, ', gagné', sep=''))
}
```

```
## [1] "pile, perdu"
## [1] "pile, perdu"
## [1] "pile, perdu"
## [1] "pile, perdu"
## [1] "face, gagné"
```

2.9 Importation et exportation des données

L'importation des données est une étape fondamentale dans l'analyse. Pour charger dans l'espace de travail (c'est à dire dans la mémoire) les données stockées dans un fichier (de texte, .csv, excel, ...) on peut utiliser la fonction de base `read.table()` avec de nombreux arguments. Les trois arguments plus importants sont :

- `file` : nom (et adresse) du fichier, entre guillemets
- `header` : les éléments de la première ligne sont-ils les noms des colonnes ?
- `sep` : caractère séparant les colonnes

`read.table()` renvoie un data frame.

```
url1 <- 'https://helios2.mi.parisdescartes.fr/~vperduca/cours/programmation/data/Iris.txt'
d1 <- read.table(url1,
                 header=TRUE, # la première ligne contient le nom des variables
                 sep=';') # les variables sont separées par des ;
head(d1)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
url2 <- 'https://helios2.mi.parisdescartes.fr/~vperduca/cours/programmation/data/heart.txt'
d2 <- read.table(url2,
                 header=TRUE,
                 sep='\t') # les variables sont separées par une tabulation
dim(d2); names(d2)
```

```
## [1] 270  13
```

```
##  [1] "age"          "sexe"         "type_douleur" "pression"     "cholester"
##  [6] "sucre"        "electro"      "taux_max"     "angine"       "depression"
## [11] "pic"          "vaisseau"     "coeur"
```

Pour les données stockées dans le format `.Rda` ou `.Rdata`, l'importation se fait avec `load()` avec l'argument `file=` nom fichier. Si on veut charger des données directement à partir d'un url, ne pas oublier d'utiliser la fonction `url()` (cela n'était pas nécessaire dans `read.table()`) :

```
url3 <- 'https://helios2.mi.parisdescartes.fr/~vperduca/cours/programmation/data/Iris.Rda'
load(url(url3))
```

L'exportation des données peut se faire dans un format de type texte (ou .csv, excel...) à l'aide de `write.file()` ou dans le format `.rda` ou `.Rdata` à l'aide de `save()`. Dans les deux cas, les deux arguments principales sont `x=` données à sauvegarder et `file=` le nom du fichier (entre guillemets).

Si les données sont stockées (ou doivent être sauvegardées) localement, il est nécessaire de connaître (et pouvoir modifier) le répertoire de travail :

```r
getwd() # essayer sur sa machine!
```

```
## [1] "/Users/vittorioperduca/Desktop/Programmation"
```

```r
setwd('~/Documents') # pour se déplacer dans le répertoire Documents
```

On rappelle que dans les machines Linux et OS, `~/` est un raccourcie pour `/Users/nom_utilisateur`. Pour les machines Windows, la syntaxe des adresses est légèrement différente. Par exemple on utilise `\` à la place de `/` .

3. Exercices

3.1 Vecteurs, matrices, listes

1. Faire les exercice suivants dans le livre de V. Goulet :
   a. 2.2 (page 47). Question supplémentaire : remplacer le premier élément de `x` par la valeur manquante et retrouver tous les éléments supérieurs à 5. Pour cela essayer aussi à l'aide de la fonction `which()`.
   b. 2.3
   c. 2.1
   d. 3.1 (page 72)
   e. 3.2
   f. 3.8. Pour simuler un vecteur de longueur 10×7 on pourra échantillonner une loi de Poisson d'espérance $\lambda = 8$ à l'aide de la fonction `rpois()`. Indications : On utilisera la fonction `apply(X,MARGIN,FUN)` pour appliquer une fonction `FUN` à toutes les lignes (`MARGIN=1`) ou colonnes (`MARGIN=2`) de la matrice `X`. En particulier, on utilisera les fonctions `sum()`, `mean()` et `max()`. Consultez l'aide de toutes ces fonctions !
2. On considère la matrice
$$B = \begin{pmatrix} 3 & 0 & 0 \\ -4 & -1 & -8 \\ 0 & 0 & 3 \end{pmatrix}$$
A l'aide des fonctions `det()` et `eigen()`, calculer le déterminant de B, les valeurs propres et les vecteurs propres. Calculer l'inverse de B.

3.2 Fonctions et structures de contrôle

1. Faire les exercice suivants dans le livre de V. Goulet :
   a. 4.2 (page 85). Indication : écrire un fonction qui prend en arguments x et w et rend en sortie la somme pondérée.
   b. 5.1 (page 101)
2. Ecrire une fonction `racine()` qui calcule la racine carrée d'un nombre réel positif $a > 0$ en implémentant la méthode de Newton. La fonction prendra en argument `a`, un point d'initialisation positif `x1` et le nombre d'itérations `n`. On rappelle que la méthode de Newton pour trouver les zéros de la fonction $f(x)$ consiste à construire une suite $(x_n)_n$ où x_{n+1} est donnée par l'intersection de la tangente à f en x_n et l'axe des abscisses :
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

3.3 Importation des données et data frame

Enregistrer dans un répertoire spécifique le fichier de texte https://helios2.mi.parisdescartes.fr/~vperduca/cours/programmation/data/hepatitis.txt

1. Importer les données dans le fichier. Attention : les données manquantes ont été codées par un `?`, lire la documentation de `read.table()`.
2. Trouver le nombre d'observations, afficher le noms des variables et les six premières observations. Vérifier que la valeur de `STEROID` de la quatrième observation est manquante à l'aide de la fonction appropriée.
3. Calculer la valeur moyenne de `ALBUMIN` chez les femmes et chez les hommes.
4. Créer une variable `NSYMP` comptant le nombre de fois où une variable est égale à `yes` entre `FATIGUE` et `MALAISE`. Attention au format de ces deux variables !