

Langage R

Nicolas Baradel
`nicolas.baradel@polytechnique.edu`



Introduction

Premiers pas

Les conditionnelles

Les boucles

Les fonctions

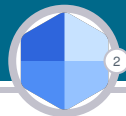
Les variables

Les objets

Fonctions usuelles

Programmation efficace

Simulation de variables aléatoires



- ▶ **R** est un langage **interprété** ;
- ▶ **R** se programme de manière **vectorielle** ;
- ▶ **R** orienté pour les applications en **statistiques** et en **probabilités**.



- ▶ **R** se **télécharge** à <https://cran.r-project.org/> ;
- ▶ La version 64-bit ne fonctionne que sur les systèmes 64-bit, la version 32-bit fonctionne sur les deux systèmes ;
- ▶ Dans la version 32-bit, **R** ne peut allouer qu'environ 3 Go de mémoire vive ;
- ▶ Tous les systèmes d'exploitations modernes sont en 64-bit : téléchargez la version 64-bit.



- ▶ **R** dispose de **Rgui** qui est une console R légèrement plus avancée ;
- ▶ **Rgui** dispose d'un système de gestion de **scripts R** ;
- ▶ **Rgui** est minimal, notamment la version Windows ;
- ▶ Il existe **RCode** : un système moderne de gestion de projets **R** ;
- ▶ **RCode** se télécharge à <https://pgm-solutions.com/rcode/>.

Introduction

Comment utiliser R



The screenshot displays the R GUI interface. The main window is titled "RGui (64-bit)" and has a menu bar with "Fichier", "Édition", "Packages", "Fenêtres", and "Aide". Below the menu bar are several icons. The "R Console" window is open, showing the following text:

```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.

> "Bonjour tout le monde"
[1] "Bonjour tout le monde"
> |
```

The "Sans titre - Éditeur R" window is also open, displaying the text: "Bonjour tout le monde".



- ▶ Il existe **RCode** : un système moderne de gestion de projets **R** ;
- ▶ **RCode** se télécharge à <https://pgm-solutions.com/rcode/>

Introduction

Comment utiliser R



The screenshot displays the RStudio environment. The top-left pane (EXPLORATEUR) shows a file explorer with folders 'données', 'SPX.csv', 'fonctions.R', and 'main.R'. The top-right pane (main.R) contains the following R code:

```
1 cat(coucou <- "Bonjour tout le monde\n")
2 |
```

The bottom-left pane (Console) shows the output of the R command:

```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.








Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> cat(coucou <- "Bonjour tout le monde\n")
Bonjour tout le monde
>
```

The bottom-right pane (Environnement) shows the current environment with the variable 'coucou' assigned the value 'Bonjour tout le monde'.



RCode se décompose en les vues :

- ▶  File Explorer : un dossier contenant votre projet ;
- ▶  main.R : pour éditer le code ;
- ▶  >_ Console : pour interagir avec **R** ;
- ▶  Packages : pour charger et installer des packages ;
- ▶  ? Browser : utilisé pour l'aide de **R** et afficher les sorties HTML ;
- ▶  History : affiche l'historique des commandes envoyées à **R** ;
- ▶  Environment : affiche en temps réel les variables instanciées dans **R** et permet leur visualisation / édition.



- ▶ 1
[1] 1
- ▶ "R"
[1] "R"
- ▶ 5+7
[1] 12
- ▶ $1+(1+1/2*(1+1/3*(1+1/4*(1+1/5))))$
[1] 2.716667
- ▶ TRUE
[1] TRUE



- ▶ **R** a un typage **faible** ;
- ▶ On ne précise pas le type, mais toutes les variables ont un type **implicite** ;
- ▶ L'**affectation** se fait avec `=` ou `<-`
- ▶

```
x = 3
x
[1] 3
```
- ▶

```
y <- 4
y
[1] 4
```
- ▶

```
(z <- x^y)
[1] 81
```
- ▶ On préfère `<-` qui est l'opérateur d'affectation standard de **R**.



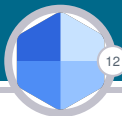
- ▶ La fonction `typeof` permet d'obtenir le type d'une variable sous la forme d'une chaîne de caractères.
- ▶

```
x <- "R"  
typeof(x)  
[1] "character"
```
- ▶

```
typeof(TRUE)  
[1] "logical"
```
- ▶

```
typeof(pi)  
[1] "double"
```
- ▶

```
typeof(5)  
[1] "double"
```



- ▶ Les booléens sont les variables de type `logical` ;
- ▶ Il y a `TRUE` et `FALSE` ;
- ▶ L'opérateur `&&` correspond au ET logique ;
- ▶ `TRUE && FALSE`
`[1] FALSE`
- ▶ L'opérateur `||` correspond au OU logique ;
- ▶ `TRUE || FALSE`
`[1] TRUE`
- ▶ L'opérateur `!` correspond au NON logique ;
- ▶ `!TRUE`
`[1] FALSE`



- ▶ Une condition se fait avec **if** de la manière suivante :

```
if(condition)
{
  instructions
}
```
- ▶ `condition` est un **booléen** qui vaut `TRUE` ou `FALSE`;
- ▶ Le bloc `instructions` est exécuté si et seulement si `condition` vaut `TRUE`;



- ▶

```
x <- 7 #Ceci est un commentaire
if(x %% 2 == 1) #Si x modulo 2 est égal à 1
{
  cat("x est impair\n") #cat permet d'écrire dans la
  console, \n permet de faire un retour à la ligne
}
x est impair
```
- ▶ Il est possible d'ajouter des instructions si la condition n'est pas réalisée avec le mot-clé `else`.
- ▶ Toutefois, si on ajoute `else` ici à la suite, on obtiendra une erreur car **R** aura déjà exécuté le `if` et aura terminé cette instruction.
- ▶ Soit on encadre la condition par des accolades, soit on écrit le mot-clé `else` sur la même ligne que l'accolade fermante du `if`.



```
▶ x <- x + 1 #x <- 8
  if(x %% 2 == 1) #FALSE
  {
    cat("x est impair\n")
  } else
  {
    cat("x est pair\n")
  }
  x est pair
```




- ▶ Une boucle **for** est une instruction qui est réalisée sur un ensemble $i \in I$. Le plus courant est d'effectuer une instruction sur $i \in \{1, 2, \dots, n\}$ pour $n \in \mathbb{N}^*$. À cette fin, l'opérateur `:` complète par les entiers entre le membre de gauche et le membre de droite :
- ▶ `1:5`
`[1] 1 2 3 4 5`
- ▶ Il s'agit d'un **vecteur** que nous verrons plus loin, ici I est $\{1, 2, \dots, 5\}$ représenté par `1:5`. La boucle `for` a la sémantique :
- ▶

```
for(i in I)
{
  instructions
}
```



```
▶ I <- 1:5
▶ for(i in I)
  {
    cat(paste0("Itération i = ", i, "\n")) #paste0
    permet de concaténer des chaînes de caractères
  }
Itération i = 1
Itération i = 2
Itération i = 3
Itération i = 4
Itération i = 5
```



- ▶ La boucle **while** permet de répéter une instruction tant qu'une condition est vérifiée. Sa syntaxe est :
- ▶

```
while(condition)
{
  instructions
}
```
- ▶ Une boucle **for** peut toujours s'écrire comme une boucle **while** :
- ▶

```
i <- 1
while(i <= n)
{
  instructions
  i <- i+1
}
```
- ▶ Attention aux **boucles infinies** avec **while**.

Les boucles

while



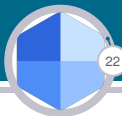
```
▶ n <- 123456789
▶ i <- 2
{
  while(n != 1) #Tant que n est différent de 1
  {
    if(n %% i == 0)
    {
      cat(paste0(i, " "))
      n <- n %/% i #Division euclidienne (entière)
    }
    else
      i <- i+1
  }
  cat("\n")
}
3 3 3607 3803
```



- ▶ Une fonction prend une ou plusieurs valeurs et retourne une variable ;
- ▶ Une fonction peut ne rien retourner, elle peut aussi ne pas prendre d'argument.
- ▶ `nomfonction <- function(arg1, ..., argn)`
 {
 instructions
 }
- ▶ L'appel se fait :
- ▶ `nomfonction(x1, ..., xn)`
- ▶ Si la fonction renvoie une valeur, elle se termine par `return` ; dès qu'un `return` est rencontré, l'exécution de la fonction s'arrête.
- ▶ Par exemple, `return("R")` permet de retourner la chaîne de caractère "R".



- ▶ **R** contient beaucoup de fonctions ;
- ▶ `sqrt(2)` #fonction racine carrée
[1] 1.414214
- ▶ `sum(1:100)` #sum prend un vecteur en argument (que nous verrons au chapitre suivant) et renvoie la somme des éléments
[1] 5050
- ▶ Mettons le code qui décomposait en nombres premiers en fonction.



```
► premiers <- function(n)
{
  i <- 2
  while(n != 1)
  {
    if(n %% i == 0)
    {
      cat(paste0(i, " "))
      n <- n %/% i
    }
    else
      i <- i+1
  }
  cat("\n")
}
```

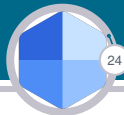
```
► premiers(123456789)
3 3 3607 3803
```



- ▶ Écrire une fonction `factorielle` qui calcule le nombre factoriel d'un entier naturel ; la fonction de **R** qui fait ça est `factorial` ;
- ▶ Écrire une fonction `gammaEuler` qui approxime à l'ordre $n \in \mathbb{N}^*$ la constante γ d'Euler définie par la limite :

$$\gamma = \lim_{n \rightarrow +\infty} \left(\sum_{k=1}^n \frac{1}{k} - \log(n) \right).$$

On pourra vérifier avec `-digamma(1)` qui vaut γ .



- Écrire une fonction qui calcule le n -ème élément de la suite de Fibonacci de valeur initiale u_0 et u_1 en mettant les valeurs par défaut : $u_0 = 0$, $u_1 = 1$. La suite de Fibonacci est définie par :

$$U_0 = u_0,$$

$$U_1 = u_1,$$

$$U_{n+2} = U_{n+1} + U_n, \quad n \geq 0.$$

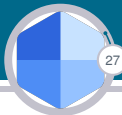


```
► facto <- function(n)
  {
    if(n == 0)
      return(1)

    fac <- 1
    for(i in 1:n)
      fac <- fac * i
    return(fac)
  }
```



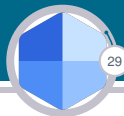
```
▶ gammaEuler <- function(n)
{
  gamma <- - log(n)
  for(k in 1:n)
    gamma <- gamma + 1/k
  return(gamma)
}
```



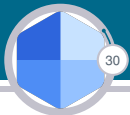
```
► fibonacci <- function(n, u0=0, u1=1)
{
  if(n == 0)
    return(u0)
  if(n == 1)
    return(u1)
  u <- 0
  um1 <- u1
  um2 <- u0
  for(i in 1:n)
  {
    u <- um1 + um2
    um2 <- um1
    um1 <- u
  }
  return(u)
}
```



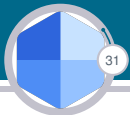
- ▶ Le type d'une variable se récupère avec la fonction `typeof`.
- ▶ `typeof(2)`
`[1] "double"`
- ▶ Le type par défaut des variables **numériques** est le **double**.
- ▶ L'entier existe, il suffit de mettre le caractère **L** à la fin du nombre.
- ▶ `(x <- 1L)`
`[1] 1`
- ▶ `typeof(x)`
`[1] "integer"`
- ▶ De manière générale, ces deux types sont `numeric` et la fonction `is.numeric` renvoie `TRUE` pour ceux deux types.
- ▶ Il est possible de faire un programme **R** entier sans se soucier de cela, si besoin, **R** fera automatiquement les conversions.
- ▶ `is.numeric(1) && is.numeric(1L)`
`[1] TRUE`



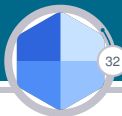
- ▶ Le type **booléen** contient deux alternatives : TRUE et FALSE.
- ▶ Il est nommé `logical` dans **R**.
- ▶ Il existe aussi la valeur **NA** qui représente une information indisponible.



- ▶ Le type **character** représente les chaînes de caractères.
- ▶ L'affectation se fait de la manière suivante :
- ▶ `s <- "rpgm"`
- ▶ Pour concaténer une chaîne de caractère, nous utilisons la fonction `paste0` et pour afficher une chaîne de caractère, la fonction `cat`.
- ▶ `cat(paste0(s, "\n"))`
`rpgm`
- ▶ La fonction **substr** permet d'extraire une sous-chaîne de caractères en indiquant le premier et le dernier caractère.
- ▶ `substr(s, 2, 4)`
`[1] "pgm"`



- ▶ Les fonctions `tolower` et `toupper` permettent de mettre respectivement en minuscule et en majuscule une chaîne de caractères.
- ▶ `toupper(s)`
`[1] "RPGM"`
- ▶ Il existe bien d'autres fonctions de manipulation des chaînes de caractères.



- ▶ Le type `Date` est un type bien utile pour manipuler des données dont l'une des variable est une date précise.
- ▶ Pour créer une date depuis une chaîne de caractère, on utilise la fonction `as.Date` de la manière suivante :
- ▶

```
(R <- as.Date("29/02/2000", "%d/%m/%Y"))
```

```
[1] "2000-02-29"
```
- ▶ Dans la chaîne de caractères, `%d` indique la position du jour, `%m` celle du mois et `%Y` celle de l'année.
- ▶ La fonction `format` convertit une date en chaîne de caractères.
- ▶

```
format(R, "%d/%m/%Y")
```

```
[1] "29/02/2000"
```
- ▶ La date du jour est donnée par :
- ▶

```
Sys.Date()
```

```
[1] "2018-06-12"
```



- ▶ Le **vecteur** est l'objet fondamental de **R**.
- ▶ Il s'agit d'un regroupement de valeurs de même type.
- ▶ Pour créer un vecteur de type `double` et de taille 3, on écrit :
- ▶ `(x <- numeric(3))`
- ▶ `[1] 0 0 0`
- ▶ On peut accéder à un élément en utilisant `[]` :
- ▶ `x[1]`
`[1] 0`
- ▶ L'indexation démarre à **1** et se fait de 1 à *n*.



- ▶ Un vecteur que nous avons déjà croisé :
- ▶ `1:5`
`[1] 1 2 3 4 5`
- ▶ La fonction `length` permet de renvoyer la longueur d'un vecteur.
- ▶ `length(x)`
`[1] 3`
- ▶ Il est possible de **concaténer** des éléments pour former un vecteur
- ▶ `(x <- c(1, 3, 7))`
`[1] 1 3 7`
- ▶ Et même de concaténer des vecteurs
- ▶ `c(x, x)`
`[1] 1 3 7 1 3 7`



- ▶ En fait, une variable de taille 1 est représentée comme un vecteur dans **R**.
- ▶ `x <- pi`
- ▶ `x[1]`
`[1] 3.141593`
- ▶ `length(x)`
`[1] 1`
- ▶ La fonction **rep** permet de créer un vecteur de taille n dont chaque composante est identique.
- ▶ `rep(5, 3)`
`[1] 5 5 5`



- ▶ La fonction `seq` (pour sequence) permet de créer une suite de nombre régulière.
- ▶ `seq(1, 2, 0.2)`
`[1] 1.0 1.2 1.4 1.6 1.8 2.0`
- ▶ `seq(0, 1, length=6)`
`[1] 0.0 0.2 0.4 0.6 0.8 1.0`



- ▶ Il est possible de donner un vecteur de booléens qui donne les indices à garder.

```
x <- c(1, 3, 5)
x[c(FALSE, TRUE, TRUE)]
[1] 3 5
```
- ▶ Il est possible de donner directement la valeur des indices choisis.

```
x[c(1, 3, 3, 2)]
[1] 1 5 5 3
```
- ▶ Une autre possibilité est d'appeler tous les éléments sauf un. La syntaxe est `x[-a]` où `a` est un indice entier.

```
x[-2]
[1] 1 5
```



La règle suivante est **fondamentale** sur **R**.

- ▶ Tous les opérateurs (arithmétiques, logiques, etc.) appliqués à deux vecteurs de même taille renvoient un vecteur de même taille où l'opérateur a été appliqué **élément par élément**.

- ▶ `y <- 0:2`

```
x+y
```

```
[1] 1 4 7
```

```
x*y
```

```
[1] 0 3 10
```

```
x^y
```

```
[1] 1 3 25
```

```
x == y
```

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



Autre règle **fondamentale** : le **recyclage**.

- ▶ Tous les opérateurs définis précédemment (arithmétiques, logiques, etc.) appliqués à un vecteur et une variable de taille un renvoient un vecteur où l'opérateur a été appliqué à la variable et à tous les éléments du vecteur initial **un à un**.

- ▶

```
2*x+1
[1] 3 7 11
x^2
[1] 1 9 25
factorial(x) %% 2
[1] 1 0 0
x <= 2
[1] TRUE FALSE FALSE
```



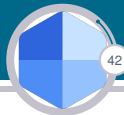

- ▶ Ecrire une fonction f qui prend n en argument et qui renvoie un vecteur composé des $n + 1$ premiers carrés de \mathbb{N} .
- ▶ Ajoutez un test qui permet de s'assurer que n est un nombre positif (s'il n'est pas entier, on pourra arrondir à l'inférieur). Pour cela, on pourra utiliser la fonction `stop` qui permet d'arrêter le programme en générant un message d'erreur (en argument) et `as.integer` qui renvoie un type entier d'un nombre numérique, arrondi à l'inférieur s'il n'était pas entier.



- Ecrire une fonction `deriv1` qui prend un vecteur x , un pas h , et qui renvoie un vecteur de taille `length(x) - 1` l'approximation du nombre dérivé :

$$\partial_{[h]}x_i := \frac{x_{i+1} - x_i}{h}.$$

On fera **sans boucle**.



```
► f <- function(n) ##cas simple: return((0:n)^2)
  {
    if(is.numeric(n))
      {
        if(n >= 0)
          return( (0:as.integer(n))^2 )
        stop(paste0("Le nombre ", n, " est négatif"))
      }
    stop(paste0("La valeur ", n, " n'est pas
numérique"))
  }
```



```
▶ deriv1 <- function(x, h)
  return((x[-1] - x[-length(x)])/h)
```



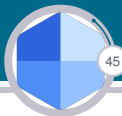
- ▶ Une **matrice** est un vecteur avec une représentation à **deux indices**.
- ▶ Une matrice se crée avec la fonction `matrix` dans laquelle on donne un vecteur, le nombre de lignes, le nombre de colonnes.

▶ `(X <- matrix(1:9, 3, 3))`

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

- ▶ On peut accéder à un élément (i, j) en utilisant `[,]` :

▶ `X[2, 3]`
`[1] 8`



- ▶ On peut aussi l'appeler en utilisant un indice de type *vecteur*, i.e. on utilisant :

$$i' := j + (i - 1) * nrow$$

- ▶ `x[8]`
`[1] 8`
- ▶ Il est possible de remplir une matrice par *ligne* avec l'argument `byrow = TRUE`.
- ▶ La fonction `as.matrix` permet de convertir un vecteur de taille n en une matrice à n lignes et une colonne.
- ▶ La fonction `t` permet d'obtenir la matrice **transposée**.
- ▶ Le **produit matriciel** se fait avec `%*%` et non pas avec `*`, ce dernier est le produit des deux matrices **élément par élément**



- ▶ Il est possible de construire une matrice **diagonale** avec **diag** :

- ▶ `diag(1:3)`

```
      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

- ▶ La fonction **cbind** permet de combiner des vecteurs en une matrice en les plaçant par colonne :

- ▶ `cbind(rep(1,3), rep(2,3), rep(3,3))`

```
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
```



- ▶ La fonction `rbind` fait la même chose mais en plaçant les vecteurs en ligne
- ▶ Il est possible de nommer les lignes et les colonnes en utilisant l'argument `dimnames` :
- ▶ `X <- matrix(1:9, 3, 3, dimnames = list(c("R1", "R2", "R3"), c("C1", "C2", "C3")))`

▶ X

	C1	C2	C3
R1	1	4	7
R2	2	5	8
R3	3	6	9



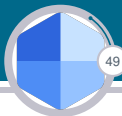
- ▶ On peut extraire une ligne ou une colonne sous la forme d'un vecteur en ne spécifiant que l'indice la ligne ou de la colonne

- ▶ `X[1,]`
C1 C2 C3
1 4 7

- ▶ `X[, 1]`
R1 R2 R3
1 2 3

- ▶ On peut aussi utiliser le nom de la ligne ou de la colonne

- ▶ `X["R1",]`
C1 C2 C3
1 4 7



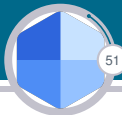
- ▶ Il est possible de forcer la conservation du type `matrix` avec l'argument `drop = FALSE`
- ▶ `c(is.matrix(X[1,]), is.matrix(X[1, ,drop=FALSE]))`
`[1] FALSE TRUE`
- ▶ `dim(X[1, ,drop=FALSE])` #`dim` renvoie le nombre de lignes et de colonnes
`[1] 1 3`
- ▶ La fonction `length` renverra le nombre total d'éléments de la matrice
- ▶ `length(X)`
`[1] 9`
- ▶ On peut obtenir le nombre de lignes grâce à la fonction `nrow` et le nombre de colonnes grâce à la fonction `ncol`
- ▶ `c(nrow(X), ncol(X))`
`[1] 3 3`



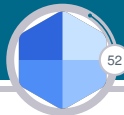
- ▶ Le type `data.frame` est le type naturelle des **matrices de données**
- ▶ La matrice a un type **unique** pour l'ensemble de ses éléments
- ▶ La `data.frame` a un type **unique par colonne** où chaque colonne représente une **caractéristique** et chaque ligne une **donnée**
- ▶ Exemple de `data.frame` :
- ▶

```
(X <- data.frame(Prenom = c("Joseph", "Augustin"), age = c(15, 19)))
```

	Prenom	age
1	Joseph	15
2	Augustin	19



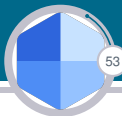
- ▶ Les appels peuvent se faire comme ceux d'une matrice
- ▶ `X[, "Prenom"]`
`[1] Joseph Augustin`
Levels: Augustin Joseph
- ▶ `X$age`
`[1] 15 19`
- ▶ On remarque que la première ne semble pas représenter une chaîne de caractères. La fonction `str` permet de voir le type par colonne.
- ▶ `str(X)`
`'data.frame': 2 obs. of 2 variables:`
`$ Prenom: Factor w/ 2 levels "Augustin","Joseph": 2 1`
`$ age : num 15 19`



- ▶ Par défaut, dans **R** 3.y.z et antérieurs les chaînes de caractères sont considérées comme des **factor** qui représentent des **modalités**, pour les régressions.
- ▶ Il est possible de forcer à conserver les chaînes de caractères avec `stringsAsFactors = FALSE`
- ▶

```
(X <- data.frame(Prenom = c("Joseph", "Augustin"), age = c(15, 19)), stringsAsFactors = FALSE)
```

```
  Prenom age
1  Joseph  15
2 Augustin 19
```



- ▶ Mais cette fois, le type est bien character
- ▶ `str(X)`
`'data.frame': 2 obs. of 2 variables:`
`$ Prenom: chr "Joseph" "Augustin"`
`$ age : num 15 19`
- ▶ Depuis **R** 4.0.0 (avril 2020), les chaînes de caractères sont bien importées en chaîne de caractères.



- ▶ La liste est un regroupement d'**objets arbitraires**
- ▶ L'exemple suivant illustre la création d'une liste
- ▶

```
(l0 <- list(a = TRUE, b = 1:3))
```

```
$a  
[1] TRUE  
$b  
[1] 1 2 3
```
- ▶ L'appel peut se faire par le nom avec \$ ou [[]]
- ▶

```
l0$a
```

```
[1] TRUE
```
- ▶

```
l0[["a"]]
```

```
[1] TRUE
```



- ▶ L'appel peut aussi se faire via l'indice numérique
- ▶ `10[[1]]`
`[1] TRUE`
- ▶ L'ajout de nouveaux éléments à la liste se fait facilement :
- ▶ `10$c <- "s"`
- ▶ `10$c`
`[1] "s"`



- ▶ Si la fonction `dim` de **R** est appliquée à un vecteur, elle renvoie `NULL`. Si elle est appliquée à une matrice de taille $n \times m$, elle renvoie n et m . Écrire une fonction `DIM` qui renvoie la taille du vecteur entré en argument (si l'argument est un vecteur) et la valeur habituelle de `dim` sinon.
- ▶ Écrire une fonction qui teste le type de l'argument. Si celui-ci est un vecteur, la fonction renvoie le dernier élément. Si celui-ci est une matrice, la fonction renvoie le dernier élément de la première ligne.



- ▶ `sign` : fonction signe, renvoie 1 si l'argument est positif, 0 s'il est nul, -1 s'il est négatif ;
- ▶ `abs` : fonction valeur absolue ;
- ▶ `sqrt` : fonction racine carrée ;
- ▶ `exp` : fonction exponentielle ;
- ▶ `log` : fonction logarithme, le second argument (base) est la base du logarithme ;
- ▶ `log2`, `log10` : logarithme de base 2 et 10 ;
- ▶ `expm1` : fonction $x \mapsto \exp(x) - 1$;
- ▶ `log1p` : fonction $x \mapsto \log(1 + x)$;



- ▶ \cos : fonction cosinus ;
- ▶ \sin : fonction sinus ;
- ▶ \tan : fonction tangente ;
- ▶ \arccos : fonction arccosinus ;
- ▶ \arcsin : fonction arcsinus ;
- ▶ \arctan : fonction arctangente ;
- ▶ \cosh : fonction cosinus hyperbolique, $x \mapsto \frac{e^x + e^{-x}}{2}$;
- ▶ \sinh : fonction sinus hyperbolique, $x \mapsto \frac{e^x - e^{-x}}{2}$;
- ▶ \tanh : fonction tangente hyperbolique, $x \mapsto \frac{\sinh(x)}{\cosh(x)}$;



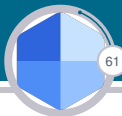
- ▶ `sort` : trie un vecteur ;
- ▶ `order` : renvoie les indices d'un vecteur de tel sorte que `x[order(x)]` renvoie `sort(x)` ;
- ▶ `which.min` : renvoie l'indice du minimum d'un vecteur ;
- ▶ `which.max` : renvoie l'indice du maximum d'un vecteur ;
- ▶ `sum` : renvoie la somme d'un vecteur ;
- ▶ `mean` : renvoie la moyenne d'un vecteur ;
- ▶ `var` : renvoie la variance d'un vecteur ;



- Soit X une variable aléatoire dont la densité est définie par la fonction f :

$$\forall x \in \mathbb{R}, f(x) = \frac{1}{\sqrt{2\pi}(1+x^2)} \exp\left(-\frac{\tan^2(x)}{2}\right) \mathbf{1}_{]-\frac{\pi}{2}, \frac{\pi}{2}[}(x)$$

où $\mathbf{1}$ est la fonction indicatrice (elle vaut 1 si x est dans l'intervalle, 0 sinon). Écrire la fonction f sous **R**.



- ▶ On suppose avoir la matrice suivante :
- ▶ `X <- cbind(c(1,2,1,3,2), c(121, 256, 842, 510, 82),
c(1, 2, 3, 4, 5), c(5, 11, 2, 7, 3))`
- ▶ Trier la matrice par ordre croissant selon la première colonne et, en cas d'égalité, selon la deuxième colonne (toujours par ordre croissant).



- ▶

```
f <- function(x)
{
  if(x > -pi/2 && x < pi/2)
    return( exp(-0.5*tan(x)^2)/(sqrt(2*pi)*(1+x^2)) )
  return(0)
}
```
- ▶

```
X <- cbind(c(1, 2, 1, 3, 1), c(121, 256, 842, 510,
82), 1:5, c(5, 11, 2, 7, 3))
X <- X[order(X[, 1], X[, 2]), ]
```



- ▶ On dira qu'une fonction f est **vectorielle** si f prend un ou des vecteurs en argument (et éventuellement d'autres arguments de taille 1) et renvoie un vecteur où une fonction a été appliquée élément par élément. C'est le cas par exemple de la fonction `sqrt` ou `exp`.
- ▶ Soit la boucle de la forme :
- ▶ `for(i in I)`
- ▶ `x[i] <- f(i,x[i],z[i])`
- ▶ Si f est **vectorielle** alors cette boucle est **toujours évitable**, la solution est :
- ▶ `x <- f(1:length(x),x,z)`



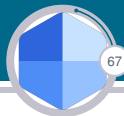
- ▶ Par exemple, si nous souhaitons affecter à x le carré de i , la solution est
- ▶ `x <- (1:length(x))^2`
- ▶ Ou alors, associer à x_i l'exponentielle d'un élément z_i d'un vecteur z auquel on ajoute la constante 2, la solution est
- ▶ `x <- exp(z) + 2`
- ▶ Il se peut que nous souhaitions modifier x uniquement sur une partie de ses indices. Par exemple, quelque chose de la forme
- ▶ `for(i in I)`
- ▶ `if(h(i,x[i],z[i]))`
- ▶ `x[i] <- f(i,x[i],z[i])`
- ▶ où h est une fonction **vectorielle** qui renvoie TRUE ou FALSE, f n'est appliquée que sur un **sous-ensemble** de I où h est vérifiée



- ▶ Si f et h sont **vectorielles** alors cette boucle est **toujours évitable**, la solution est :
- ▶ `ind <- h(1:length(x),x,z)`
- ▶ `x[ind] <- f((1:length(x))[ind], x[ind], z[ind])`



- ▶ Nous pouvons souhaiter modifier x uniquement sur les 30 premiers indices et leur affecter z , dans ce cas
- ▶ `ind <- 1:length(x) <= 30`
- ▶ `x[ind] <- z[ind]`
- ▶ Dans ce cas très simple nous aurions pu écrire
- ▶ `x[1:30] <- z[1:30]`
- ▶ Un autre exemple : nous souhaitons affecter la valeur 0 à tous les indices où x vaut NA. Dans ce cas
- ▶ `ind <- is.na(x)`
- ▶ `x[ind] <- 0`
- ▶ Ici, `ind` n'est utilisé qu'une seule fois, nous aurions pu écrire directement
- ▶ `x[is.na(x)] <- 0`



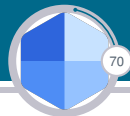
- ▶ Il ne faut pas être effrayé par le fait d'écrire x dans x . Ce qui est à l'intérieur n'est que le calcul d'un vecteur de `logical` en fonction de x (s'il vaut `NA` ou non).
- ▶ Ensuite, nous effectuons une opération dans les indices de x vérifiant cette condition. Par exemple
- ▶ `(x <- c(7, 2, NA, 3, -1, NA))`
- ▶ `[1] 7 2 NA 3 -1 NA`
- ▶ `x[is.na(x)] <- 0`
- ▶ `x`
- ▶ `[1] 7 2 0 3 -1 0`



- ▶ Un autre exemple : là où la somme de x et y (deux vecteurs de même taille) est supérieure à z , affecter à x le modulo de z par 2, sinon celui de y par 2
- ▶ `ind <- x + y > z`
- ▶ `x[ind] <- z[ind]%%2`
- ▶ `x[!ind] <- y[!ind]%%2`
- ▶ Rappel : le point d'exclamation est le NON logique, il inverse les TRUE et FALSE
- ▶ Nous affectons, en fonction de $x + y > z$, à chaque élément, soit $z%%2$, soit $y%%2$



- ▶ Un dernier exemple intervient dans la classification d'une variable.
- ▶ Par exemple, si x est inférieur à un seuil a fixé, nous le mettons dans la classe 0, s'il est supérieur à b , nous le mettons dans la classe 2, et enfin s'il est entre les deux, dans la classe 1.
- ▶ `y <- rep(1, length(x))`
- ▶ `y[x < a] <- 0`
- ▶ `y[x > b] <- 2`



- ▶ Pour les conditions, les opérateurs `&&` et `||` ne sont pas vectoriels. Les versions vectorielles correspondantes sont `&` et `|`.
- ▶ Pour tester si une condition est vérifiée sur **tous** les éléments d'un vecteur, on utilisera la fonction `all`.
- ▶ Pour tester si une condition est vérifiée sur **au moins** un élément d'un vecteur, on utilisera la fonction `any`.



- ▶ Retour sur un exercice passé : Écrire une fonction `gammaEuler` qui approxime à l'ordre $n \in \mathbb{N}^*$ la constante γ d'Euler définie par la limite :

$$\gamma = \lim_{n \rightarrow +\infty} \left(\sum_{k=1}^n \frac{1}{k} - \log(n) \right).$$

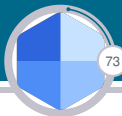
On pourra vérifier avec `-digamma(1)` qui vaut γ . On veillera à **ne pas utiliser de boucle**.



- ▶ Retour sur un exercice passé : Soit X une variable aléatoire dont la densité est définie par la fonction f :

$$\forall x \in \mathbb{R}, f(x) = \frac{1}{\sqrt{2\pi}(1+x^2)} \exp\left(-\frac{\tan^2(x)}{2}\right) \mathbf{1}_{]-\frac{\pi}{2}, \frac{\pi}{2}[}(x)$$

où $\mathbf{1}$ est la fonction indicatrice (elle vaut 1 si x est dans l'intervalle, 0 sinon). Écrire cette fois-ci la fonction f de manière vectorielle sous **R** afin qu'elle puisse prendre un vecteur x et renvoie un vecteur de même taille $f(x)$ où f est appliquée élément par élément. On veillera à **ne pas utiliser de boucle**.



- ▶ Écrire une fonction qui prend une matrice $X = (x_{i,j})$ de taille $n \times m$ en argument et qui renvoie un vecteur de taille m où l'élément j est la moyenne sur i de $(\cos(x_{i,j})^i)_{1 \leq i \leq n}$ (attention, bien voir que le cosinus est élevé à la puissance i).
- ▶ Écrire une fonction qui prend une matrice carrée $X = (x_{i,j})$ de taille $n \times n$ en argument et renvoie la trace de la matrice X . La trace de la matrice X est la somme des éléments diagonaux définie par

$$\text{Trace}(X) = \sum_{k=1}^n x_{k,k}.$$

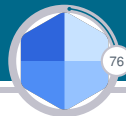


```
▶ gammaEuler <- function(n)
  return( sum(1/(1:n)) - log(n) )

▶ f <- function(x)
  {
    y <- numeric(length(x))
    ind <- x > -pi/2 & x < pi/2
    z <- x[ind]
    y[ind] <- exp( -0.5*tan(z)^2 )/(sqrt(2*pi)*(1+z^2))
    return(y)
  }
```



```
▶ f <- function(X)
  return( colMeans(cos(X)^row(X)) )
```



- ▶ **R** utilise une suite de nombres pseudo-aléatoires générés par une suite arithmétique.
- ▶ L'algorithme par défaut utilisé est le *Mersenne- Twister*. Sa période est de $2^{19937} - 1$ ce qui vaut environ 10^{6000} .
- ▶ Cet algorithme est souvent considéré comme le meilleur compromis entre efficacité à simuler les variables aléatoires et qualité du générateur (indépendance entre les simulations, la loi simulée est bien la loi uniforme).
- ▶ Le générateur s'initialise tout seul
- ▶ Il est possible de choisir sa graine avec `set.seed`
- ▶ Cela permet de reproduire les mêmes simulations à des fins de *debug*.



À la loi d'une variable aléatoire, dont le nom donné par **R** est `nom`, sont associées quatre fonctions :

- ▶ `dnom`
- ▶ `pnom`
- ▶ `qnom`
- ▶ `rnom`



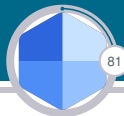
- ▶ La première, d_{nom} , prend en premier argument un nombre décimal puis d'éventuels paramètres pour les arguments suivants. Cette fonction évalue la densité de la loi nom en l'argument passé en paramètre.
- ▶ Pour les variables aléatoires discrètes, cette densité est $\mathbb{P}(\text{nom} = \bullet)$ où \bullet est le point en lequel d_{nom} est évaluée. Dans le cas des variables aléatoires continues, la densité est celle au sens *habituel* du terme.



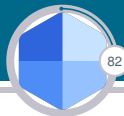
- ▶ La deuxième fonction, p_{nom} , fonctionne comme d_{nom} sauf qu'elle n'évalue pas la densité mais la fonction de répartition
- ▶ La fonction q_{nom} prend en premier argument un nombre α dans l'intervalle $[0,1]$ et lui associe le quantile d'ordre α de la loi nom
- ▶ Enfin, la fonction r_{nom} permet de générer des simulations indépendantes de la loi nom . Son premier argument est le nombre souhaité de simulations



- ▶ La fonction `dnom` possède un argument facultatif `log`. Par défaut, il vaut `FALSE`, mais s'il est mis à `TRUE`, ce n'est pas la densité qui est renvoyée mais le logarithme de la densité
- ▶ Il est préférable d'écrire `dnom(x, log = TRUE)` que `log(dnom(x))`, un exemple est donné dans le cas de la loi normale
- ▶ Nous voyons maintenant les différentes lois sous **R**



- ▶ La loi uniforme $\mathcal{U}([a, b])$ est nommée `unif`.
- ▶ Sa fonction densité évaluée en $x = (x_i)_{1 \leq i \leq n}$ s'évalue avec `dunif(x, a, b)`
- ▶ Sa fonction de répartition s'évalue avec `punif(x, a, b)`
- ▶ Ses quantiles (avec $x_i \in [0, 1]$) s'évalue avec `qunif(x, a, b)`
- ▶ La simulation de $n \in \mathbb{N}$ variables aléatoires i.i.d. se fait avec `runif(n, a, b)`
- ▶ Nous ne donnons qu'après la fonction de simulation des lois, les trois autres se déduisent de la même manière



- ▶ La loi binomiale $\mathcal{B}(m, p) : \text{rbinom}(n, m, p)$
- ▶ La loi de Poisson $\mathcal{P}(a) : \text{rpois}(n, a)$
- ▶ La loi géométrique issue de 0 $\mathcal{G}(p) : \text{rgeom}(n, p)$
- ▶ La loi binomiale négative $\mathcal{BN}(m, p) : \text{rnbinom}(n, m, p)$
- ▶ La loi normale $\mathcal{N}(m, s^2) : \text{rnorm}(n, m, s)$
- ▶ La loi du Khi-2 $\mathcal{X}(d) : \text{rchisq}(n, d)$
- ▶ La loi de student $\mathcal{St}(d) : \text{rt}(n, d)$



- ▶ La loi log-normale $\mathcal{LN}(m, s^2) : \text{rlnorm}(n, m, s)$
- ▶ La loi exponentielle $\mathcal{E}(b) : \text{rexp}(n, b)$
- ▶ La loi gamma $\mathcal{G}(a, b) : \text{rgamma}(n, a, b)$
- ▶ La loi de Weibull $\mathcal{W}(a, b) : \text{rweibull}(n, a, b)$
- ▶ La loi Beta $\mathcal{B}(a, b) : \text{rbeta}(n, a, b)$
- ▶ La loi de Cauchy $\mathcal{C}(a, b) : \text{rcauchy}(n, a, b)$
- ▶ La loi de Fisher $\mathcal{F}(d, k) : \text{rf}(n, d, k)$



- ▶ Ré-échantillonner, c'est créer un nouvel échantillon à partir d'un échantillon initial
- ▶ Soit $(x_i)_{1 \leq i \leq n}$, il s'agit de tirer uniformément dans l'échantillon
- ▶ Chaque x_i a une probabilité de $\frac{1}{n}$, cela peut amener des valeurs **multiples**
- ▶ Par défaut il n'y a pas de remise, il s'agit juste de tirer l'ordre aléatoirement
- ▶

```
x <- c(1,2,3,5,8,13,21)
```
- ▶

```
sample(x)
```

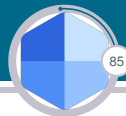
```
[1] 8 2 3 21 1 13 5
```
- ▶

```
sample(x, replace = TRUE)
```

```
[1] 3 5 1 13 3 2 8
```
- ▶

```
sample(x, replace = TRUE, prob = c(0.3, 0.3, 0.1, 0.1, 0.1, 0.1, 0))
```

```
[1] 8 5 1 2 1 2 5
```



- ▶ La fonction qui va nous permettre de simuler des vecteurs gaussiens se situe dans le *package* MASS
- ▶ Ce *package* est déjà présent dans **R**, on le charge avec `library(MASS)`
- ▶ La fonction qui va nous permettre de simuler des vecteurs gaussiens est `mvrnorm`
- ▶ Son premier argument est, comme d'habitude, le nombre $n \in \mathbb{N}$ de simulations
- ▶ Le deuxième argument est un vecteur de taille d qui est le vecteur moyenne dans \mathbb{R}^d
- ▶ Le troisième argument est une matrice symétrique qui est la matrice de variance-covariance dans \mathbb{S}^d



- ▶ Pour simuler n variables aléatoires de loi $\mathcal{N}(m, S)$ avec $m \in \mathbb{R}^d$ et $S \in \mathcal{S}^d$, on utilise :
- ▶

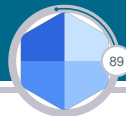
```
library(MASS)  
n <- 300  
m <- c(1, 2)  
S <- matrix(c(1,0.7,0.7,1), 2, 2)  
X <- mvrnorm(n, m, S)
```
- ▶ X est une matrice à 2 colonnes et n lignes
- ▶ On peut obtenir la moyenne de chaque composante avec `colMeans` et regarder la matrice de variance-covariance avec `var`



- ▶ Estimons π par méthode de Monte-Carlo. Pour ce faire, prenons le carré unité $[-1; 1]^2$ et le disque de rayon 1 et de centre 0 de ce carré. L'aire du carré est 4, l'aire du disque est π . Si on tire uniformément dans le carré (ce qui revient à tirer deux lois uniformes dans $[-1, 1]$, une étant l'axe des abscisses, l'autre l'axe des ordonnées), la probabilité d'être dans le disque est $\frac{\pi}{4}$. Écrire une fonction qui prend en argument le nombre de simulations n et qui renvoie une estimation π .



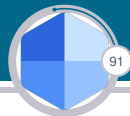
- Vérifions le théorème de la limite centrale. Simulez n échantillons de taille $m = 20$ de loi $\mathcal{G}(\alpha, \beta)$ avec $(\alpha, \beta) = (1, 1)$. Calculez la moyenne de chacun des n échantillons de taille m , centrés réduits avec la vraie moyenne et le vrai écart-type, et comparez les quantiles de la distribution avec celle de la loi normale aux ordre $\alpha = 0.5\%, 1\%, 5\%, 25\%, 50\%, 75\%, 95\%, 99\%, 99.5\%$



```
▶ MCpi <- function(n)
  {
    return(4*mean(runif(n,-1,1)^2 + runif(n,-1,1)^2 <=
1))
  }
```



```
► n <- 10^6
  m <- 25
  X <- matrix(rgamma(n*m, 1, 1), n, m)
  z <- sqrt(m)*(rowMeans(X) - 1)
  p <- c(0.005, 0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9,
        0.95, 0.99, 0.995)
  quantile(z, p)
  qnorm(p)
```



```
► X <-  
  read.table("http://nicolasbaradel.fr/R/donnees/SPX_m.txt",  
            header = TRUE, dec = ",", colClasses = "numeric")
```

Fin du cours !