RO203 - Jeux, graphes et Recherche Opérationnelle

Zacharie Ales, Marc Etheve, Eric Soutil
2020 - 2021

Introduction à Julia

Exercise 1 Mise en place

1. Récupérer l'archive du TP d'initiation julia sur ecampus.
2. Décompresser l'archive téléchargée.
3. *(à faire uniquement si vous êtes sur un ordinateur de l'ENSTA)*
   Utiliser la commande suivante dans un terminal afin de faciliter l'utilisation des logiciels de RO à l'ENSTA :

```
usediam ro
```

4. Ouvrir une console Julia.
5. Dans cette console, déplacez-vous dans le répertoire `RO203/TP_Julia` en utilisant la commande `cd` :

```
# Exemple sous linux
julia> cd("/home/user/RO203/TP_Julia")

# Exemple sous windows
julia> cd("C:/Users/user/RO203/TP_Julia")
```

Exercise 2 Les bases

1. **Les types en Julia**
   Les types que vous utiliserez sont :
   - `Bool` : booléen ;
   - `Int64` : entier ;
   - `Float64` : réel ;
   - `String` : chaîne de caractères ;
   - `VariableRef` : variable d'un programme linéaire.

   La syntaxe pour définir une variable en Julia est

```
nomVariable = Type(valeur)
```

   Exemple :

```
a = Int64(3)
```

   Préciser le type d'une variable est facultatif (mais peut permettre d'éviter des erreurs). On peut donc également écrire :

```
a = 3
```

2. **Les fonctions**
La syntaxe pour définir une fonction comportant un attribut est :

```
function nomDeLaFonction(argument1::Type)
   # Contenu de la fonction
end
```

Exemple de fonction calculant le carré d'un entier `n` :

```
function square(n::Int64)
  return n*n
end
```

Remarque : préciser le type des arguments d'une fonction est facultatif mais **très fortement** recommandé.

3. Une fonction peut retourner plusieurs valeurs et prendre plusieurs arguments :

```
# Fonction retournant a-b et a+b
function incDec(a::Int64, b::Int64)
    return a-b, a+b
end

huit, douze = incDec(10, 2)
```

4. **Structures de contrôle** (tests et boucles)
   a) Exemple de `if`

```
if a == 0
  println("a est égal à 0")
else
  println("a est différent de 0")
end
```

   b) Exemple de `while`

```
i = 1
while i == 1
  i += 3
  println("i vaut ", i)
end
```

Utiliser une boucle `while` pour calculer le premier élément de la suite de Fibonacci ($u_1 = u_2 = 1$, $u_n = u_{n-1} + u_{n-2}$) qui dépasse un million (solution : $u\_31 = 1346269$).

Remarque : Pour éviter des problèmes liés à la visibilité des variables, votre code devra être inclu dans une fonction. Exemple :

```
                        ┌ Définition de la fonction
                        ↓
julia> function fibonacci()
         # Votre code ici
       end
            ┌ Appel de la fonction
            ↓
julia> fibonacci()
```

Remarque 2 : Pour éviter de réécrire votre fonction à partir de 0 si jamais vous vous trompez, vous pouvez utiliser les flèches haut et bas de votre clavier.

c) Exemples de `for` :

```
for i in 1:10
  println(i)
end

# i augmente de 2 à chaque passage
for i in 1:2:10
  print(i, ", ")
end # Affiche "1, 3, 5, 7, 9"
```

Utiliser une boucle `for` pour afficher la valeur de $\frac{1}{2}$, $\frac{1}{3}$, ..., $\frac{1}{10}$.

5. **Aléatoire et arrondis**

a) Pour arrondir un réel, on utilise les fonctions `round`, `ceil` et `floor` :

```
a = round(3.2) # Retourne 3.0
a = round(Int64, 3.2) # Retourne 3

b = ceil(3.2) # Retourne 4.0
b = ceil(Int, 3.2) # Retourne 4

c = floor(3.8) # Retourne 3.0
c = floor(Int, 3.8) # Retourne 3
```

b) Pour générer aléatoirement un réel entre 0.0 et 1.0, on utilise la commande

```
rand()
```

Générer aléatoirement :

— 0 ou 1 ;

— un entier entre 1 et 10.

Remarque : Comme ces instructions s'effectuent en une seule ligne, pas besoin de les définir dans des fonctions.

Exercise 3 Les tableaux

Le type d'un tableau d'entier en une dimension est `Array{Int64, 1}`.
Le type d'un tableau d'entier en deux dimension est `Array{Int64, 2}`.

1. **Déclaration de tableaux vides**

```
    ┌ Vecteur de taille 0
    ↓
vector = Array{Int64}(undef, 0)
matrix = Array{Int64}(undef, 0, 2)
    ↑
    └ Matrice à deux colonnes et 0 lignes
```

Remarque : `Array{Int64}` est équivalent à `Array{Int64, 1}`.

2. **Déclaration de tableaux contenant 0 ou 1**

```
  ┌─ Vecteur de taille 3 contenant des 0
  ↓
v0 = Array{Int64}(zeros(3))
m1 = Array{Int64, 2}(ones(2, 4))
  ↑
  └─ Matrice de taille 2x4 contenant des 1
```

3. **Déclaration explicite d'un tableau**

```
  ┌─ Vecteur de taille 4
  ↓
v = [1, 2, 3, 4]
m = [1 2; 3 4]
  ↑
  └─ Matrice de taille 2x2
```

Attention : `[1 2 3 4]` définit une matrice de taille 1×4 ce qui, en julia, est différent d'un vecteur de taille 4.

4. **Ajout d'éléments à un tableau**

```
v = Array{Int64}(undef, 0)
append!(v, 1) ← Ajoute 1 à un tableau à une dimension
  ↑
  └─ Remarque: par convention '!' indique que la méthode modifiera le 1er argument

m = Array{Int64}(undef, 0, 2)
m = vcat(m, [1 2])
       ↑
       └─ Ajouter la ligne [1 2] à la matrice m
```

5. **Taille d'un tableau**

```
m = [1 2 3; 4 5 6]

l = size(m, 1) # Nombre de lignes du tableau (= 2)
c = size(m, 2) # Nombre de colonnes d'un tableau (= 3)
```

6. **Appliquer une fonction ou un opération à tous les éléments d'un tableau**
On utilise l'opérateur point.

```
a = [1.1, 2.2, 3.3, 4.4]

# Ajoute 1 à tous les éléments de a
b = a .+ 1 # b sera égal à [2.1, 3.2, 4.3, 5.4]

# Arrondir tous les éléments d'un tableau
c = round.(Int, a) # c sera égal à [1, 2, 3, 4]
```

7. **Itérer sur les éléments d'un tableau**

```julia
animaux = ["chat", "chien", "mouette", "ornithorynque", "axototl"]

for animal in animaux
  println(animal)
end
```

Définissez et testez une fonction prenant en argument un tableau d'entier et retournant la plus petite valeur qu'il contient ainsi que son indice.

Rappel : Le type d'un tableau d'entier est `Array{Int64, 1}`.

8. **Filtrage des éléments d'un tableau**

```julia
# Filtre toutes les valeurs x du tableau [1, 2, 3, 4, 5]
# en ne gardant que les éléments > 2
t = [1, 2, 3, 4, 5]
t345 = filter(x->x > 2, t)
          └ Retourne [3, 4, 5]

# Garde les éléments du tableau contenant "a"
animaux = ["chat", "chien", "girafe"]
animauxContenantA = filter(x->occursin("a", x), animaux)
Retourne ["chat", "girafe"] ┘          └ Vrai si x contient "a"
```

Définir une fonction prenant en argument un vecteur de chaînes de caractères et retournant un tableau contenant les chaînes ayant un nombre impair de caractères.

Indication 1 : La fonction `rem(a, b)` renvoie le reste dans la division euclidienne de `a` par `b` (en d'autres termes, cela correspond à : a modulo b).

Indication 2 : La fonction `length(s::String)` renvoie la taille d'une chaîne de caractères.

Exercise 4 Résolution de programmes linéaires en nombres entiers

L'objectif de cet exercice est de résoudre le programme linéaire suivant :

$$(P)\begin{cases} \text{Maximiser} & \sum_{i=1}^{n} x_i - y & & \\ \text{s.c.} & x_1 + y & \geq 4 & \\ & x_i + x_{n-i+1} & \leq n & \forall i \in \{1,2,...,\frac{n}{2}\} \\ & \sum_{i\in\{1,...,n\} \mid i \text{ divisible par } 3} x_i & \leq 1 & \\ & y & \geq 0 & \\ & x_i & \in \mathbb{N} & \forall i \end{cases}$$

1. Comme la définition d'un programme nécessite plus d'une dizaines de lignes, nous allons maintenant utiliser julia en écrivant le code dans un fichier. Créer et ouvrir un fichier `ex4.jl`. Ce fichier pourra être exécuté dans un terminal julia en utilisant la commande :

```julia
include("ex4.jl")
```

Remarque : Pour que cela fonctionne, il faudra bien sûr s'assurer que le terminal julia se trouve dans le même répertoire que le fichier `ex4.jl`.

2. **Les librairies**
La résolution de programmes linéaires en nombres entiers nécessite l'utilisation de deux librairies :

1. la librairie JuMP (permet de modéliser des problèmes d'optimisation)
2. une librairie contenant un solveur (nous utiliserons la librairie propriétaire CPLEX dans ce cours, mais des alternatives existent).

Afin d'inclure ces librairies, ajouter les lignes suivantes au fichier `ex4.jl` :

```
using JuMP
using CPLEX
```

3. **Déclaration d'un modèle**
Un programme est représenté par un modèle contenant les variables, les contraintes et l'objectif du problème. La définition d'un modèle dépend du solveur considéré. Avec CPLEX, la syntaxe est la suivante :

```
m = Model(CPLEX.Optimizer)
```

4. **Exemples de déclaration de variables**

```
# Définition d'une variable binaire
@variable(m, x, Bin)

# Définition d'une variable réelle
@variable(m, y >= 0)

# Définition d'un vecteur de 10 variables entières
@variable(m, z[1:10] >= 0, Int)

# Définition d'une matrice de 5x4 variables
@variable(m, w[1:5,1:4] >= 0)
```

5. **Exemples de définition de contraintes**

```
@constraint(m, x + y >=1)
     ↑ x + y ≥ 1

@constraint(m, [i in 1:10], z[i] >= i)
     ↑ z_i ≥ 1  ∀i ∈ {1,...,10}

@constraint(m, sum(z[i] for i in 1:10) <= 70)
     ↑ Σ_{i=1}^{10} z_i ≤ 70

# Définition d'une contrainte avec condition dans une somme
@constraint(m, sum(z[i] for i in 1:10 if rem(i, 2) == 0) >= 40)
     ↑ Σ_{i∈{1,...,10} | i pair} z_i ≥ 40

# Définition d'une contrainte pour tout i avec une condition sur i
@constraint(m, [i in 1:10; rem(i, 3) == 1], z[i] >= 3)
     ↑ z_i ≥ 3  ∀i ∈ {1,4,7,10}
```

6. **Exemples de définition d'objectifs**

```
@objective(m, Min, x - y)

@objective(m, Max, sum(z[i] for i in 1:10))
```

7. **Résolution et récupération des résultats d'un modèle**

```
# Résolution d'un modèle
optimize!(m)

# Récupération des valeurs d'une variable
vX = JuMP.value(x)
vZ2 = JuMP.value(z[2])

# Récupération du status de la résolution
  status = termination_status(m)
  isOptimal = status == MOI.OPTIMAL
      └─ true si le problème a été résolu optimalement
```

8. Définir une méthode `ex4(n::Int64)` qui modélise le problème (P) dans un fichier `ex4.jl`.
9. Résoudre et afficher les solutions associées à ce problème pour $n = 5$ et $n = 10$.

   Indication : La valeur de l'objectif des solutions optimales de ces deux problèmes est 11 et 50.

10. Vous avez sans doute constaté que le programme met plusieurs secondes à s'exécuter la première fois. Ceci est dû au fait qu'à la première exécution, les librairies JuMP et CPLEX sont chargées.

Exercise 5 Gestion des fichiers

Dans votre projet vous aurez besoin de :

— lire des fichiers contenant les données de problèmes ; et

— écrire vos résultats dans des fichiers.

1. **Lecture d'un fichier**

```
# Si le fichier "./data/test.txt" existe
if isfile("./data/test.txt")

  # L'ouvrir
  myFile = open("./data/test.txt")

  # Lire toutes les lignes d'un fichier
  data = readlines(myFile) ← Retourne un tableau de String

  # Pour chaque ligne du fichier
  for line in data

    # Afficher la ligne
    println(line)
  end

  # Fermer le fichier
  close(myFile)

end
```

   a) Définir une fonction `readN1()` qui lit les lignes du fichier "./data/n1.txt" et qui retourne le plus petit entier qu'il contient (solution : 1).

   Indication : La syntaxe pour transformer une chaîne de caractères en entier est :

```
myInt = parse(Int64, "3")
```

b) Définir une fonction `readFile(path::String)` qui lit le fichier situé au chemin `path` et retourne le plus grand entier qu'il contient qui soit divisible par 5. Appliquer cette méthode sur le fichier "./data/n2.txt" (solution : 75).

Remarque : Certains fichiers contiennent plusieurs entiers sur une même ligne séparés par des virgules. Pour récupérer tous les entiers d'une ligne dans un tableau, on utilisera la commande `split` :

```
# Découpe la chaîne "1,5,2,4" en ["1", "5", "2", "4"]
t = split("1,5,2,4", ",")
```

2. **Lecture des fichiers contenus dans un répertoire**

```
# Pour chaque fichier contenu dans le répertoire "./data"
for file in readdir("./data")

  # Afficher le nom du fichier
  println(file)
end
```

Définir une fonction `fileNamesWith1` utilisant une boucle for afin d'afficher le nom des fichiers du répertoire "./data" contenant "1" (solution : affiche "n1.txt" et "v1.txt").

3. **Ecriture dans un fichier**

```
# Ouvrir le fichier "output.txt" dans lequel on pourra écrire
myFile = open("output.txt", "w")

# Ecrire "test" dans ce fichier
println(fout, "test")

close(fout)
```

Définir une méthode `findMaxInDirectory` qui trouve le plus grand entier M contenu dans les fichiers du répertoire "./data" et écrit "maxInFile = M" dans le fichier "resultat.txt" (solution : 6786).

Indication : La syntaxe pour concaténer deux chaînes de caractères est :

```
# Concaténation de deux chaînes de caractères
t = "RO" * "203"

# Concaténation d'une chaîne de caractères et d'une variable
a = 203
t = "RO" * string(a)
```

Remarque : N'oubliez de fermer chaque fichier une fois que vous l'avez parcouru.

Exercise 6 Résolution d'une grille de sudoku

L'objectif de cet exercice est de modéliser la résolution de la grille de sudoku ci-dessous par un programme linéaire en nombres entiers :

```
 -----------------------------
| -  7  - | 2  -  3 | -  1  - |
| 3  -  - | -  -  - | -  -  - |
| -  -  - | -  -  - | 2  -  - |
 -----------------------------
| -  -  - | -  -  - | -  -  - |
| -  -  - | -  -  - | -  -  - |
| -  -  - | -  -  - | -  -  2 |
 -----------------------------
| 2  -  - | -  -  - | -  -  - |
| -  -  - | -  -  - | -  -  - |
| -  -  - | -  -  - | -  -  - |
 -----------------------------
```

1. **Modélisation**
   Modéliser (sur papier) la résolution d'une grille de sudoku de taille n sous la forme d'un programme linéaire en nombres entiers contenant les variables binaires suivantes :

$$x_{i,j,k} = \begin{cases} 1 & \text{si la valeur de la case } (i,j) \text{ est égale à } k \\ 0 & \text{sinon} \end{cases} .$$

   Indication : L'objectif importe peu ici car on souhaite simplement obtenir une solution réalisable. Vous pouvez donc, par exemple, choisir de minimiser la valeur d'une case.

   Compléter la méthode `sudoku()` du fichier `sudoku.jl` en y mettant votre modèle et en y ajoutant un affichage du résultat obtenu.