CNAM
RCP218

IA, Optimisation et Contrôle

TP : Introduction à `Julia`

1 Installation et mise en place

1. Pour installer `Julia`
   Windows-Mac-Linux https://julialang.org/downloads/
   Installation en suivant les instruction ici https://julialang.org/downloads/platform/
2. Récupérer l'archive du TP sur `lecnam.net`
3. Ouvrir une console et lancer `Julia`. Dans cette console, déplacez-vous dans le répertoire `RCP2018/TP_Julia` en utilisant la commande `cd` : `# Exemple sous linux`
   `julia> cd("/home/user/RCP2018/TP_Julia")`
   `# Exemple sous windows`
   `julia> cd("C:/Users/user/RCP218/TP_Julia")`
4. Charger les package nécessaire au TP :
   ```
   julia> import Pkg
   julia> Pkg.add("JuMP")
   julia> Pkg.build("JuMP")
   julia> using JuMP
   julia> Pkg.add("GLPK")
   julia> Pkg.build("GLPK")
   julia> using GLPK
   ```

`Julia` peut être utilisé de deux façons :
- **En mode console :** les instructions julia sont écrites une par une une et en appuyant sur `Entrée` elle est exécutée.
- **Exécution d'un fichier d'instructions julia** (il est suffixé par `.jl`)
  `julia> include("fichier.jl")`

2 Les bases de `Julia`

2.1 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)`

Remarque : Préciser le type d'une variable est facultatif (mais peut permettre d'éviter des erreurs). On peut donc également écrire `a = 3`

2.2 Les fonctions

La syntaxe pour définir une fonction comportant un attribut est :

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

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

Exercice 1 — *Calcul du carré d'une variable*

Soit la fonction `carre` calculant le carré d'un entier `n` :

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

Écrire ce code dans le fichier `carre.jl` et l'exécuter grâce aux commandes suivantes :

```
julia> include("carre.jl")
julia> carre = square(10)
```

Exercice 2 — *Plusieurs retours de fonction*

Une fonction peut retourner **plusieurs valeurs** et prendre plusieurs arguments. Soit la fonction `incDec` retournant la somme et la différence de deux entiers :

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

Écrire ce code dans le fichier `incDec.jl` et l'exécuter grâce aux commandes suivantes :

```
julia> include("incDec.jl")
julia> huit, douze = incDec(10, 2)
```

2.3 Structures de contrôle

Conditionnelle : if

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

Itération : while

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

Itération : 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"
```

Exercice 3 — *Suite de Fibonacci*

La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent.

$$F_0 = 0, \quad F_1 = 1, \text{ et } F_n = F_{n-1} + F_{n-2} \text{ pour } n \geqslant 2.$$

La fonction suivante évalue la valeur de cette suite pour un `n` donné :

```
function fibonacci(n::Int64)
  a = 0
  b = 1
  if n==0
    res=0
  else
    if n==1
      res=1
    else
      i=2
      while i <= n
        res = a+b
        a = b
        b = res
        i= i+1
      end
    end
  end
  println(res)
end
```

Écrire ce code dans le fichier `fibonacci.jl` et l'exécuter grâce aux commandes suivantes :

```
julia> include("fibonacci.jl")
julia> fibonacci(5)
```

Exercice 4 — *Affichage de valeurs fractionnaire*

Le code suivant utilise une boucle `for` pour afficher la valeur de $\frac{1}{2}$, $\frac{1}{3}$, ..., $\frac{1}{10}$:

2.4 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)`
   Matrice à deux colonnes et 0 lignes
   `matrix = Array{Int64}(undef, 0, 2)`
   Remarque : `ArrayInt64` est équivalent à `ArrayInt64, 1`.
2. **Déclaration de tableaux contenant 0 ou 1** Vecteur de taille 3 contenant des 0
   `v0 = zeros(3)`
   Matrice de taille 2x4 contenant des 1
   `m1 = ones(2, 4)`
3. **Déclaration explicite d'un tableau** Vecteur de taille 4
   `v = [1, 2, 3, 4]`
   Matrice de taille 2×2
   `m = [1 2; 3 4]`
   Attention : `[1 2 3 4]` définit une matrice de taille 1×4 ce qui est différent d'un vecteur de taille 4×1.

4. **Ajout d'éléments à un tableau** Déclaration d'un tableau
   `v = ArrayInt64(undef, 0)`
   Ajout de la valeur `1` au tableau `v` à une dimension
   `append!(v, 1)`
   Remarque : '!' indique que la méthode modifiera le 1er argument
   Déclaration d'une matrice
   `m = ArrayInt64(undef, 0, 2)` Ajout de la ligne `[1 2]` à la matrice `m`
   `m = vcat(m, [1 2])`
5. **Taille d'un tableau** Déclaration d'un tableau de 2 dimensions
   `m = [1 2 3; 4 5 6]`
   Nombre de lignes du tableau (= 2)
   `l = size(m, 1)`
   Nombre de colonnes d'un tableau (= 3)
   `c = size(m, 2)`
6. **Appliquer une fonction ou une opération à tous les éléments d'un tableau**
   On utilise l'opérateur point `. a = [1.1, 2.2, 3.3, 4.4]`
   Ajouter 1 à tous les éléments du tableau `a`
   `b = a .+ 1 # b sera égal à [2.1, 3.2, 4.3, 5.4]`
   Arrondir tous les éléments du tableau `a`
   `c = round.(Int, a) # c sera égal à [1, 2, 3, 4]`
7. **Itérer sur les éléments d'un tableau** `animaux = ["chat", "chien", "mouette", "ornithorynque"]`
   `for animal in animaux`
   `println(animal) end`
8. **Filtrage des éléments d'un tableau** Déclaration du tableau `t`
   `t = [1, 2, 3, 4, 5]`
   Filtrer toutes les valeurs `t` en ne gardant que les éléments > 2
   `t345 = filter(x->x > 2, t)`
   `animaux = ["chat", "chien", "girafe"]`
   Garde les éléments du tableau contenant `"a"`
   `animauxContenantA = filter(x->occursin("a", x), animaux)`

Exercice 5 Tester la fonction `min` (fichier `minTab.jl`) qui prend en argument un tableau d'entier et retourne la plus petite valeur qu'il contient ainsi que son indice.

```
function min(t::Array{Int64, 1})
  min = t[1]
  id = 1
  for i in 2:size(t, 1)
    if t[i] < t[id]
      id = i
      min = t[i]
    end
  end
  return min, id
end
```

3 Résolution de programmes linéaires en nombres entiers

A l'aide du langage de modélisation `JuMP` et du solveur `GLPK`, nous allons maintenant résoudre le programme linéaire suivant :

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

1. Créer et ouvrir un fichier `plne.jl`. Ce fichier pourra être exécuté en tapant la commande : `julia > plne.jl`

2. **Les librairies :** la résolution de programmes linéaires en nombres entiers nécessite l'utilisation de deux librairies :
   (a) la librairie `JuMP` (permet de modéliser des problèmes d'optimisation)
   (b) une librairie contenant un solveur (nous utiliserons la librairie `GLPK` dans ce cours, mais des alternatives existent).

   Afin d'inclure ces librairies, ajouter les lignes suivantes au fichier `plne.jl`
   ```
   using JuMP
   using GLPK
   ```
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 `GLPK`, la syntaxe est la suivante :
   ```
   m = Model(GLPK.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**
   $x + y \geqslant 1$
   ```
   @constraint(m, x + y >=1)
   ```
   $\forall i \in 1, \ldots, 10 \;\; z_i \geqslant 1$
   ```
   @constraint(m, [i in 1:10], z[i] >= i)
   ```
   $\sum_{i=1}^{10} z_i \leqslant 70$
   ```
   @constraint(m, sum(z[i] for i in 1:10) <= 70)
   ```
   Définition d'une contrainte avec condition dans une somme $\sum_{i \in 1,\ldots,10 \mid i \text{ pair}} z_i \geqslant 40$
   ```
   @constraint(m, sum(z[i] for i in 1:10 if rem(i, 2) == 0) >= 40)
   ```
   Définition d'une contrainte pour tout i avec une condition sur i : $z_i \geqslant 3 \; \forall i \in 1, 4, 7, 10$
   ```
   @constraint(m, [i in 1:10; rem(i, 3) == 1], z[i] >= 3)
   ```
6. **Exemples de définition d'objectifs**
   Minimiser $x - y$
   ```
   @objective(m, Min, x - y)
   ```
   Maximiser $\sum_{i=1}^{10} z_i$
   ```
   @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 et de la solution

```
vX = JuMP.value(x)
vZ2 = JuMP.value(z[2])
obj = JuMP.objective_value(model)
```

Exercice 6 — *Un premier PLNE*

1. Dans le fichier `plne.jl`, définir une fonction `plne(n::Int64)` qui modélise le problème (P).
2. 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.

4 Gestion 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
```

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
```

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(myFile, "test")
close(myFile)
```

Exercice 7 — *Lecture dans un fichier*
Définir une fonction `readN1()` qui lit les lignes du fichier "./data/n1.txt" et qui retourne le plus petit entier qu'il contient.
Indication : La syntaxe pour transformer une chaîne de caractères en entier est :
`myInt = parse(Int64, "3")`

Exercice 8 — *Écriture dans un fichier*

Définir une méthode `writeMaxInFile` qui trouve le plus grand entier M contenu dans le fichier `n1.txt` et écrit "maxInFile = M" dans le fichier "resultat.txt".

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

Exercice 9 — *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. Modéliser le problème de résoudre 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.

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