

Introduction aux outils utilisés en Travaux Pratiques

Anthony Przybylski

Université de Nantes, L3 Informatique

Plan

1 Julia

2 JuMP

Plan

1 Julia

2 JuMP

Installation

Pour les TP, nous aurons besoin des installations suivantes :

- Julia : <http://julialang.org/downloads>
- GLPK : <http://www.gnu.org/software/glpk>

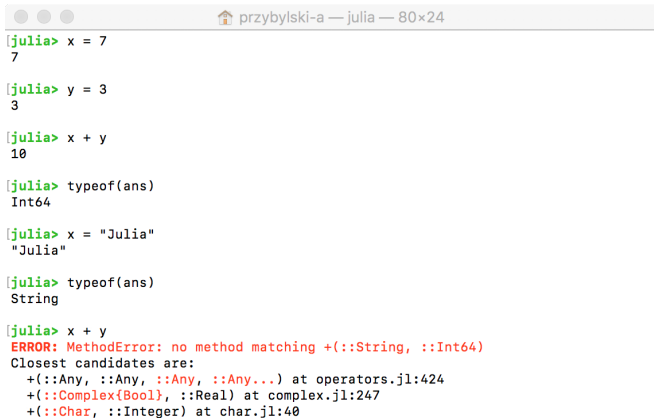
Ces installations nécessitent au préalable un compilateur C

Julia : principales caractéristiques

- Langage de programmation de haut-niveau à haute performance pour le calcul numérique, disponible depuis 2012 sous licence MIT
- Objectifs : langage avec les avantages de C (rapidité d'exécution), R (traitements statistiques), Python (simplicité et dynamisme), Matlab (algèbre linéaire)...
- Conséquences : Nombreuses possibilités, nombreux paradigmes de programmation disponibles, syntaxe très (trop?) riche
- Typage des variables : fort, dynamique
- Interface de commande en ligne (REPL) pour les interactions avec le langage, compilateur JIT basé sur LLVM

REPL

- Utilisable directement pour écrire des expressions simples, et tester des fonctions (similaire à la boucle interactive de OCaml)



```
[julia> x = 7
7

[julia> y = 3
3

[julia> x + y
10

[julia> typeof(ans)
Int64

[julia> x = "Julia"
"Julia"

[julia> typeof(ans)
String

[julia> x + y
ERROR: MethodError: no method matching +(::String, ::Int64)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:424
  +(::Complex{Bool}, ::Real) at complex.jl:247
  +(::Char, ::Integer) at char.jl:40
```

Quelques commandes/raccourcis essentiels dans le REPL

- En tapant `;` en début de ligne, l'invite de commande `julia>` devient `shell>`, et permet de lancer les commandes UNIX classiques
- En tapant `?` en début de ligne, l'invite de commande `julia>` devient `help?`, et permet de chercher de l'aide sur les fonctions en utilisant des mots-clés (exemple : `sort`)
- L'auto-complétion (ou des suggestions d'auto-complétions) pour des noms de variables, des champs de variables d'un type composé, des noms de fonctions accessibles avec la touche de tabulation
- Liste de fonctions utilisant un type particulier accessible avec la fonction `methodswith(type)`

Quelques types de base en Julia

- Entier : `a = 3` (par défaut de type `Int64` mais `Int8`, `Int16`, `Int32`, `Int128` existent aussi)
- Flottant : `a = 3.14` (par défaut de type `Float64` mais `Float16` et `Float32` existent aussi)
- Booléen : `a = true` (de type `Bool`)
- Caractère : `a = 'a'` (de type `Char`)
- Chaîne de caractères : `a = "Julia"` (de type `String`)

Les types sont hiérarchisés

Exemple : les types `Integer` sont considérés comme des sous-types des types `Real` qui sont eux-mêmes des sous-types du type abstrait `Number`

Tableaux uni-dimensionnels (vecteur) (1/2)

- Uniquement des tableaux dynamiques, à l'utilisation similaire à la classe Vector du C++

```
przybylski-a — julia — 79×35

[julia> T = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

[julia> T[3] = 4
4

[julia> T
3-element Array{Int64,1}:
 1
 2
 4

[julia> T[4] = 8
ERROR: BoundsError: attempt to access 3-element Array{Int64,1} at index [4]
Stacktrace:
 [1] setindex!{::Array{Int64,1}, ::Int64, ::Int64} at ./array.jl:583

[julia> push!(T,4)
4-element Array{Int64,1}:
 1
 2
 4
 4

[julia> T
```

Tableaux uni-dimensionnels (vecteur) (2/2)

- Inconvénient : réallocation périodique, voire recopies du tableau
- Solution préférable : Allouer initialement le tableau
`T = Vector{Int}(5)`
 T est alors un tableau non-initialisé de 5 cases de type Int64 (Attention : la fonction `push!` ajouterait un 6e element et pas un premier!)
- Autre solution pour un tableau (initialisé ou non)
`sizehint!(T,5)`
Précise au compilateur qu'on aura besoin d'au moins 5 cases (et idéalement pas plus)
- Remarque : pas de libération manuelle de la mémoire (garbage collector)

Tableaux multi-dimensionnels et Tableaux de tableaux (1/2)

- De nombreux langages (dont C) ne gèrent pas de tableaux multi-dimensionnels mais des tableaux de tableaux
- Deux types distincts en Julia (important pour l'usage de fonctions associées)

```
przybylski-a — julia — 80×24

julia> A = [1 2 3;
            4 5 6;
            7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> B = [[1,2,3],
            [4,5,6],
            [7,8,9]]
3-element Array{Array{Int64,1},1}:
 [1, 2, 3]
 [4, 5, 6]
 [7, 8, 9]

julia> det(A)
0.0

julia> det(B)
ERROR: MethodError: no method matching det(::Array{Array{Int64,1},1})
```

Tableaux multi-dimensionnels et Tableaux de tableaux

(2/2)

- Syntaxe :
 - `A[i,j]` pour accéder la case (i,j) de la matrice A
 - `B[i][j]` pour accéder à la case i du tableau B (qui est lui-même un tableau) puis à la case j du tableau $B[i]$
- Accès aux lignes/colonnes d'une matrice
 - `A[i,:]` pour obtenir la ligne i de A
 - `A[:,j]` pour obtenir la colonne j de A

Conditionnelles

Algorithmique :

```
si <condition> alors  
  <instruction(s)>  
fin si
```

```
si <condition> alors  
  <instruction(s) 1>  
sinon  
  <instruction(s) 2>  
fin si
```

Exemple en Julia :

```
if x < y  
  println(x," est inférieur à ",y)  
end
```

```
if x < y  
  println(x," est inférieur à ",y)  
else  
  println(x," est supérieur ou  
égal à ",y)  
end
```

Range de valeurs

- `a:b` avec $a < b$ définit la suite de valeurs allant de a à b par pas de 1 (type `UnitRange{T}` où T est le type de a et b)
- `a:i:b` avec $a \neq b$ et $i \neq 0$ définit la suite de valeurs allant de a à b par pas de i (type `StepRange{T1,T2}` où $T1$ est le type de a et b , et $T2$ est le type de i)
- Utilisation possible pour définir des sous-tableaux :
`T = [1,2,4,8,16]`
`T2 = T[2:4] # T2 = [2,4,8]`

Répétitives (1/2)

Algorithmique :

```
pour <itérateur> faire  
  <instruction(s)>  
fin pour
```

```
tant que <condition> faire  
  <instruction(s)>  
fin tant que
```

Exemple en Julia :

```
for i in 1:8  
  println(i)  
end  
  
i = 1  
while i <= 3  
  println(i)  
  i += 1  
end
```

Remarques :

- <itérateur> dans la boucle pour peut être compris au sens large, `for` est utilisable pour parcourir les valeurs de nombreuses collections (exemple : tableau)
- `i++` n'existe pas en Julia!!!

Répétitives (2/2)

Algorithmique :

répéter <instructions>

jusqu'à ce que <condition>

Exemple en Julia :

```
i = 1
```

```
while true
```

```
    println(i)
```

```
    i += 1
```

```
    if i > 3 break end
```

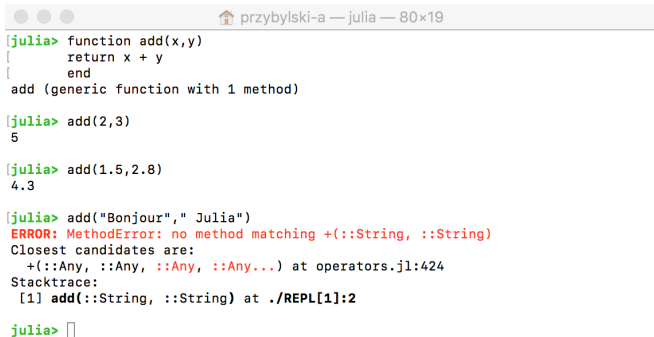
```
end
```

Remarques :

- Pas de `repeat ... until` ni de `do ... while` en Julia!!!
- Obligation d'avoir recours à un `break`!

Fonctions (1/6)

- Sans précision sur les paramètres, une fonction est générique



```
przybylski-a — julia — 80x19
[julia> function add(x,y)
    return x + y
end
add (generic function with 1 method)

[julia> add(2,3)
5

[julia> add(1.5,2.8)
4.3

[julia> add("Bonjour"," Julia")
ERROR: MethodError: no method matching +(::String, ::String)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:424
Stacktrace:
 [1] add(::String, ::String) at ./REPL[1]:2

julia> 
```

- Pour des raisons d'efficacité (gestion de la mémoire), il est souhaitable que le compilateur connaisse les types manipulés
- Possibilité de surcharger les fonctions

Fonctions (2/6)

```
przybylski-a — julia — 80x32

[julia> function add(x,y)
[     println("Cas général")
[     return x + y
[     end
add (generic function with 1 method)

[julia> function add(x::Int,y::Int)
[     println("Cas entier")
[     return x + y
[     end
add (generic function with 2 methods)

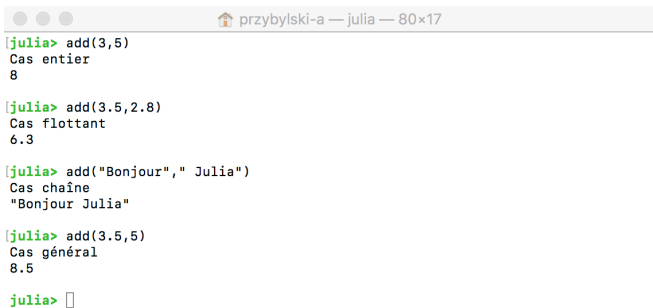
[julia> function add(x::Float64,y::Float64)
[     println("Cas flottant")
[     return x + y
[     end
add (generic function with 3 methods)

[julia> function add(x::String,y::String)
[     println("Cas chaîne")
[     return x * y
[     end
add (generic function with 4 methods)

[julia> methods(add)
# 4 methods for generic function "add":
add(x::String, y::String) in Main at REPL[4]:2
add(x::Float64, y::Float64) in Main at REPL[3]:2
add(x::Int64, y::Int64) in Main at REPL[2]:2
add(x, y) in Main at REPL[1]:2

julia> 
```

Fonctions (3/6)



```
[julia> add(3,5)                                     ]  
Cas entier  
8  
  
[julia> add(3.5,2.8)                               ]  
Cas flottant  
6.3  
  
[julia> add("Bonjour"," Julia")                     ]  
Cas chaîne  
"Bonjour Julia"  
  
[julia> add(3.5,5)                                   ]  
Cas général  
8.5  
  
julia> ]
```

- Un mécanisme appelé “multiple-dispatch” est chargé de choisir la variante la plus appropriée, avec une utilisation éventuelle de la hiérarchie de types
- Le fonctionnement est similaire au polymorphisme de la programmation orientée objet

Fonctions (4/6)

- Une fonction peut retourner plusieurs valeurs, ou plutôt un tuple de valeurs

```
przybylski-a — julia — 80×21
[julia> function euclide(a::Int,b::Int)
[      return div(a,b), a%b
[      end
[      euclide (generic function with 1 method)

[julia> euclide(10,3)
(3, 1)

[julia> typeof(ans)
Tuple{Int64,Int64}

[julia> quotient, reste = euclide(10,3)
(3, 1)

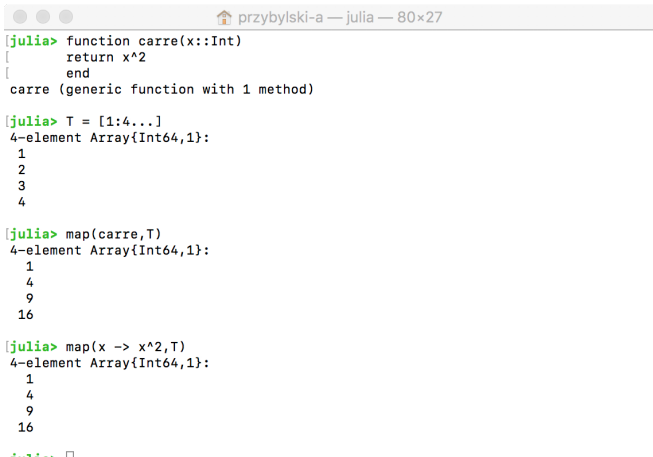
[julia> quotient
3

[julia> reste
1

[julia> ]
```

Fonctions (5/6)

- Une variable peut contenir une valeur de type fonction, une fonction peut prendre en paramètre une autre fonction, et même retourner une fonction



```
[julia> function carre(x::Int)
[       return x^2
[       end
carre (generic function with 1 method)

[julia> T = [1:4...]
4-element Array{Int64,1}:
 1
 2
 3
 4

[julia> map(carre,T)
4-element Array{Int64,1}:
 1
 4
 9
16

[julia> map(x -> x^2,T)
4-element Array{Int64,1}:
 1
 4
 9
16
```

Fonctions (6/6)

Quelques remarques

- Possibilité d'avoir des paramètres optionnels, avec éventuellement des valeurs par défaut
- Les valeurs d'un type primitif passées par paramètre ne sont pas modifiées par une fonction, cela n'est pas le cas pour un tableau
- Dans les fonctions prédéfinies :
 - Si le nom de la fonction finit par `!`, cela indique que le premier paramètre de la fonction est modifié
 - Sinon aucun paramètre n'est modifié

Utilisation d'un éditeur de texte

- Travailler dans le REPL est utile pour effectuer des tests, mais rien n'est sauvegardé
- L'utilisation combinée d'un éditeur de texte s'impose (extension `.jl` pour le nom du fichier)
- Pour charger le contenu d'un fichier `nomfichier.jl`, il suffit de taper
`include("nomfichier.jl")`

Conclusion

- Encore beaucoup à découvrir
- Nombreuses bibliothèques de fonctions disponibles
- Nombreuses structures de données disponibles, pas forcément chargées par défaut (bibliothèque `DataStructures.jl`)

Plan

1 Julia

2 JuMP

Langages de modélisation

- Langages utilisés pour une saisie naturelle des programmes linéaires (voire des programmes mathématiques)
- Écriture très proche de celle utilisée sur papier :
 - Utilisation possible de plusieurs tableaux (éventuellement multi-dimensionnels) de variables
(exemple : x_{ij} et y_j dans une même modélisation)
 - Utilisation d'ensembles pour définir les indices des variables
(exemple : $\{1, \dots, n\}$)
Cela permet également de faire exactement les mêmes regroupements de contraintes que sur papier
(exemple : contrainte avec i fixé, $\forall i \in \{1, \dots, n\}$)
- Idéal pour “juste” résoudre un programme linéaire en variables mixtes
- Utilisable sans connaissance en algorithmique et programmation

Solveur : bibliothèque de fonctions

- Solveur de programmation linéaire en variables mixtes : bibliothèque de fonctions (spécifique à UN solveur)
- Unique tableau uni-dimensionnel pour représenter les variables
⇒ Si le modèle contient plusieurs tableaux de variables (éventuellement multi-dimensionnels), une table de correspondance avec les indices de l'unique tableau du solveur doit être faite
- Aucun regroupement de contraintes n'est possible, la saisie de la matrice (creuse) des contraintes doit être réalisée
⇒ Nouveau travail de correspondance des indices
- Remarque : les bibliothèques de fonctions de certains solveurs commerciaux proposent certaines facilités
- Indispensable quand on veut faire plus qu'une simple résolution d'un Programme Linéaire (ex : traitement à partir de la solution optimale obtenue, succession de résolutions...)

Julia for Mathematical Programming

- Objectif de JuMP : proposer la facilité d'usage d'un langage de modélisation, dans une bibliothèque de fonctions directement utilisable en Julia
- Réussite : beaucoup plus direct d'utilisation que la bibliothèque de fonctions d'un solveur
- Échec : nécessite de comprendre l'usage de structures de données en Julia pour être utilisé...
- ...mais des évolutions importantes sont attendues

Installation de JuMP (et GLPK)

Dans le REPL, tapez :

- `Pkg.add(" JuMP")` pour installer JuMP
- `Pkg.add(" GLPK")` pour installer GLPK
(solveur libre pour la programmation linéaire en variables mixtes)
- `Pkg.add(" GLPKMathProgInterface")` pour pouvoir utiliser GLPK avec JuMP

Exemple

$$\begin{array}{llllll} \max z & = & 15x_1 + 60x_2 + 4x_3 + 20x_4 & & & \\ \text{s.c.} & & 20x_1 + 20x_2 + 10x_3 + 40x_4 & \leq & 21 & \\ & & 10x_1 + 30x_2 + 20x_3 & \leq & 6 & \\ & & 20x_1 + 40x_2 + 30x_3 + 10x_4 & \leq & 14 & \\ & & x_1, x_2, x_3, x_4 & \geq & 0 & \end{array}$$

à résoudre en utilisant JuMP

Exemple : Modèle explicite (1/3)

- Saisie du modèle dans un fichier dont l'extension est `.jl` (`medoc1.jl` sur `madoc`)
- On commence par spécifier les packages à utiliser
`using JuMP, GLPKMathProgInterface`
- Déclaration d'un modèle initialement vide, en spécifiant le solveur utilisé pour la résolution
`m = Model(solver = GLPKSolverLP())`
- Déclaration des variables à ajouter au modèle
`@variable(m,x1 >= 0)`
`@variable(m,x2 >= 0)`
`@variable(m,x3 >= 0)`
`@variable(m,x4 >= 0)`

Exemple : Modèle explicite (2/3)

- Déclaration de la fonction objectif (avec le sens d'optimisation)

`@objective(m, Max, 15x1 + 60x2 + 4x3 + 20x4)`

- Déclaration des contraintes en leur donnant un nom (facultatif)

`@constraint(m, Toxine1, 20x1 + 20x2 + 10x3 + 40x4 <= 21)`

`@constraint(m, Toxine2, 10x1 + 30x2 + 20x3 <= 6)`

`@constraint(m, Toxine3, 20x1 + 40x2 + 30x3 + 10x4 <= 14)`

- Résolution

`status = solve(m)`

Exemple : Modèle explicite (3/3)

- Statut de la résolution
 - `:Optimal` Problème résolu à l'optimalité
 - `:Unbounded` Problème non-borné
 - `:Infeasible` Problème impossible
 - `:Error` Sortie avec une erreur
 - ...
- En cas de problème résolu à l'optimalité
 - `getobjectivevalue(m)` retourne la valeur optimale
 - `getvalue(x1)` retourne la valeur de `x1`
(même chose pour les autres variables)

Remarques

- Les variables sont par défaut continues et libres
- Possibilité de spécifier des bornes inférieures et/ou supérieures pour une variable x
 - `@variable(m,x)`
 - `@variable(m, x >= lb)`
 - `@variable(m, x <= ub)`
 - `@variable(m, lb <= x <= ub)`
- Un troisième paramètre peut spécifier le type des variables
 - `Int` pour une variable entière
 - `Bin` pour une variable binaire
- Le symbole `==` est utilisé pour déclarer des contraintes d'égalité
- Le choix du solveur (pour GLPK) dépend du type des variables
Dès qu'une variable n'est pas continue,
`GLPKSolverLP()` doit être remplacé par `GLPKSolverMIP()`

Vers un modèle implicite

- Si on a plusieurs instances numériques (souvent!), il ne faut pas écrire un modèle explicite pour résoudre chaque instance!
⇒ Nécessité de séparer le modèle (sous une forme générique) des données
- On parle de modèle implicite

Exemple : vers un modèle implicite (1/2)

- Exemple complet dans le fichier `medoc2.jl` disponible sur madoc
- En repartant d'un modèle vide, déclaration d'un tableau de variables
`@variable(m,x[1:4] >= 0)`
- Les tableaux de variables reposent sur une syntaxe différente des tableaux en Julia
- On ne spécifie pas la taille mais un ensemble d'indices
 - Les indices peuvent être négatifs, exemple : `-3:3`
 - Les indices peuvent ne pas être entiers, exemple : `["Nantes", "Vertou", "Rezé"]`

Exemple : vers un modèle implicite (2/2)

- Peu de changements dans la déclaration de la fonction objectif...

`@objective(m, Max, 15x[1] + 60x[2] + 4x[3] + 20x[4])`

- ... et des contraintes

`@constraint(m, Toxine1, 20x[1] + 20x[2] + 10x[3] + 40x[4] <= 21)`

`@constraint(m, Toxine2, 10x[1] + 30x[2] + 20x[3] <= 6)`

`@constraint(m, Toxine3, 20x[1] + 40x[2] + 30x[3] + 10x[4] <= 14)`

- Pas de changement dans la résolution, ni dans l'affichage de la valeur optimale
- `getvalue(x)` retourne maintenant un tableau de valeurs flottantes correspondant aux valeurs des variables dans la solution optimale obtenue

Exemple : modèle implicite (1/4)

- Exemple complet dans le fichier `medoc3.jl` disponible sur madoc
- Le modèle est cette fois déclaré dans une fonction dont les paramètres spécifient
 - le solveur utilisé `solverSelected`
 - le vecteur de coûts `c::Vector{Int}`
 - la matrice des contraintes `A::Array{Int,2}`
 - le vecteur des membres de droite des contraintes `b::Vector{Int}`
- Le modèle construit sera ainsi indépendant des données et de leur taille (nombre de médicaments et de toxines)

Exemple : modèle implicite (2/4)

- Déclaration d'un modèle initialement vide
`m = Model(solver = solverSelected)`
- Déduction de la taille du problème (nombre de variables et nombre de contraintes) des données
`nbcontr, nbvar = size(A)`
- Déclaration d'un tableau de variables
`@variable(m, x[1:nbvar] >= 0)`
- Déclaration de la fonction objectif
`@objective(m, Max, sum(c[j]x[j] for j in 1:nbvar))`

Exemple : modèle implicite (3/4)

- Déclaration des contraintes
`@constraint(m, Toxine[i=1:nbcontr], sum(A[i,j]x[j] for j in 1:nbvar) <= b[i])`
 - Une contrainte est générée pour chaque i dans $1:nbcontr$
 - i est fixé dans chaque contrainte
- Le modèle complété est simplement retourné en fin de fonction

Exemple : modèle implicite (4/4)

- Les données sont construites à l'extérieur de la fonction, et passées en paramètre
- Pas de changement dans la résolution, ni dans la lecture de la valeur optimale
- Accès aux valeurs des variables du vecteur x dans la solution optimale obtenue
`getvalue(m[:x])`

Modèle implicite : remarques

- Séparation complète entre le modèle et les données
- Possibilité de déclarer des ensembles de contraintes (de manière similaire à l'écriture "à la main")

Distanciel

Résoudre le problème modélisé dans l'exercice 1.1 des TD, en utilisant un modèle implicite

$$\begin{aligned}\max z &= 12x_1 + 20x_2 \\ 0,2x_1 + 0,4x_2 &\leq 400 \\ 0,2x_1 + 0,6x_2 &\leq 800 \\ x_1, x_2 &\in \mathbb{N}\end{aligned}$$