Tutoriel Julia

Dimitri Watel

Attention! Ne pas imprimer, c'est long.

1 Installer Julia

1.1 Sous Windows

Pour installer julia sur windows, le plus simple est de télécharger l'exécutable sur la page `https://julialang.org/downloads/` ou `https://julialang.org/downloads/oldreleases.html`.

Utilisez ensuite l'exécutable et suivez les instructions.

Vous pouvez également récupérer et compiler les sources. Tout est expliqué dans le README de la page Github `https://github.com/JuliaLang/julia`.

1.2 Sous Linux ou OS X

Faire comme Windows. Pour linux, vous pouvez récupérer les fichiers binaires nécessaires sur la page `https://julialang.org/downloads/` ou `https://julialang.org/downloads/oldreleases.html`, puis décompresser l'archive dans le dossier de votre choix.

Pour MacOS, vous pouvez récuperer sur cette page un fichier `.dmg` puis vous vous débrouillez parce que le rédacteur de ce tutoriel ne sait pas comment ça marche.

Avec les sources. (Attention à ne pas faire fondre votre machine avec cette méthode.)

Vous pouvez également télécharger les sources sur la page Github `https://github.com/JuliaLang/julia`. Vous pouvez cloner le dépôt avec

```
git clone git://github.com/JuliaLang/julia.git
```

puis récupérer la version 1.0.3 avec

```
git checkout v1.0.3
```

Ensuite, installez les logiciels requis, indiqués sur la page `https://github.com/JuliaLang/julia#required-build-tools-and-external-libraries`.

Enfin, il vous faudra compiler avec la commande `make` (ou `make -j N` si vous avez N coeurs sur votre machine). Tout ceci est précisé sur le README de la page github. Ce faisant, vous installez julia en local sur votre machine. Vous pouvez alors installer (en local toujours) tous les packages julia que vous souhaitez. Si, plus tard, vous voulez désinstaller julia et tous les packages, il vous suffira de supprimer le dossier contenant julia.

Si l'erreur `recipe for target 'julia-deps' failed` apparaît, c'est probablement qu'il vous manque des prérequis. Pensez, dans ce cas, à lire également la ligne `*** Clean the OpenBLAS build with 'make -C deps clean-openblas'. Rebuild with 'make OPENBLAS_USE_THREAD=0 if OpenBLAS had trouble linking libpthread.so, and with 'make OPENBLAS_TARGET_ARCH=NEHALEM' if there were errors building SandyBridge support. Both these options can also be used simultaneously. ***` indiquée au dessus de l'erreur.

Vous pouvez finir avec un `make testall -j N` pour tester que tout va bien.

Avec un gestionnaire de paquet. Autre méthode, votre gestionnaire de paquet peut contenir une version de Julia. Veillez bien à récupérer la version 1.0.3.

1.3 Démarrer Julia en mode console

Pour démarrer Julia, dans le dossier d'installation, vous trouverez un programme `julia` ou `julia.exe` à exécuter. Julia s'ouvre alors *en mode console*. Vous pouvez maintenant coder en julia. Tous les exemples de ce tutoriel sont donnés comme s'ils avaient été fait dans ce mode.

1.4 Exécuter un programme Julia

Vous pouvez toutefois coder dans un programme. Les programmes Julia ont pour extension `.jl` et peuvent être exécutés avec la commande `julia script.jl` où `script.jl` est le programme que vous souhaitez exécuter.

Vous pouvez également exécuter un programme directement depuis le mode console ainsi :

```
> include("script.jl")
```

2 Coder avec Julia

Dans ce tutoriel, nous allons rapidement aborder les aspects classiques de la programmation: Types, Opérations, Variables, Boucles, Conditions, Fonctions, ...

2.1 Types.

Les types en Julia sont, en gros, dynamiques. C'est à dire qu'il n'est pas généralement nécessaire de préciser le type de la variable dans le programme. Pour connaître le type d'une variable ou d'une constante, il suffit d'utiliser la commande `typeof`

```
> typeof(100)
Int64
```

Les types primitifs de base sont `Int64`, `Float64`, `Bool` et `Char`. Il existe d'autres types, mais c'est principalement ceux là qui seront utilisés par Julia quand vous créerez vos variables.

Vous serez également amenés à utiliser d'autres types plus complexes comme `String` ou `Array`.

2.2 Variables.

Une variable est, comme dans (sans doute presque) tous les langages, un nom associé à une valeur. Vous pouvez affecter une variable avec `=`.

```
> x = 1
1
> x
```

```
1
```

2.3 Opérations.

Les opérations classiques (`+, *, -, /`) sont codées en julia. Quelques détails:

- `x / y` est la division classique, non euclidienne
- `x ÷ y` effectue la division euclidienne et si vous vous demandez comment on écrit le symbole ÷ avec un clavier, le rédacteur de ce tutoriel aussi... Cependant, `div(x, y)` vous donne le même résultat.
- `x \ y` effectue `x / y` (c'est logique)
- `x % y` effectue le modulo.
- `x ^ y` est l'opérateur puissance.

En ce qui concerne les booléens, vous pouvez utiliser les opérateurs `||`, `&&` et `!` pour les classiques OU, ET et NOT.

Enfin, les comparaisons classiques sont également présentes : `==, !=, >, <, >=, <=` ainsi que quelques fonctions utiles `isfinite, isinf, isnan`. Il est possible de chaîner les comparisons comme dans `1 <= x < 10` ou encore plus surprenant `1 <= x != y > 3`. Les comparaisons chaînées sont cassées et lues séparément: `1 <= x; x != y ; y > 3`.

2.4 Structure.

Vous pouvez aussi créer des structures complexes comme en C (mais vous n'êtes pas contraint de préciser le type):

```
> struct Point
x
y
end
> p = Point(1, 2)
Point(1, 2)
> p.x
1
```

```
> p.y
2
```

Les valeurs de x et y ne sont aucunement fixées comme des entiers.

```
> p = Point("abc", 2)
Point("abc", 2)
> p.x
"abc"
```

Par contre une structure n'est pas modifiable:

```
> p = Point("abc", 2)
Point("abc", 2)
> p.x = 10
ERROR: type Point is immutable
```

Pour pouvoir la modifier, il faut utiliser `mutable struct`

```
> mutable struct MPoint
x
y
end
> p = MPoint(1, 1)
Point(1, 1)
> p.x = 10
10
> p.x
10
```

2.5 Sortie et entrée standard.

Pour pouvoir écrire dans la sortie standard, il suffit d'invoquer la fonction `print` ou `println`. La seconde rajoute un saut de ligne à la fin.

Remarque: si vous travaillez en mode console, le résultat est toujours affiché, même si vous n'avez pas invoqué la fonction `print`.

```
> x = 1
1
> y = 2
2
> print(x, ' ', y)
```

```
1 2
> print(x, y)
12
> println(x, ' ', y)
1 2

> println(x); println(y)
1
2
```

Pour lire une entrée, vous pouvez utiliser la fonction `readline` puis, si besoin, `parse` pour transformer la chaîne de caractère en nombre .

```
> x = readline(stdin)
10 <---- taper 10 dans la console
"10"
> x
"10"
> y = parse(Int64, x)
10
```

2.6 Conversions

On vient de voir la fonction `parse` pour transformer une chaîne de caractère en entier. Cela fonctionne également pour les `Float64` et les `Bool`.

Pour convertir un entier en flottant, vous pouvez utiliser le constructeur `Float64`:

```
> Float64(10)
10.0
```

Pour faire l'inverse, vous pouvez utiliser les fonctions `floor`, `ceil`, `trunc` et `round`. Mais ils renvoient par défaut des flottants, vous pouvez rajouter Int64 dans les arguments pour forcer le type. Attention également aux nombres négatifs.

x	`floor`(x)	`ceil`(x)	`trunc`(x)	`round`(x)
10.6	10.0	11.0	10.0	11.0
-10.6	-11.0	-10.0	-10.0	-11.0

x	`floor`(Int64, x)	`ceil`(Int64, x)	`trunc`(Int64, x)	`round`(Int64, x)
10.6	10	11	10	11
-10.6	-11	-10	-10	-11

Enfin, pour les booléens, vous pouvez convertir avec le constructeur `Bool`.

```
> Bool(1)
true
> Bool(0)
false
> Bool(1.0)
true
> Bool(2)
ERROR: InexactError: Bool(Bool, 2)
```

2.7 Conditions : if

Vous pouvez écrire une condition comme suit

```
> x = 11
11
> if x > 10
print(2)
end
2
```

Vous pouvez rajouter des `elseif` ou `else` pour les cas où la condition est fausse

```
> x = 1
1
> if x > 10
print(2)
elseif x > 5
print(1)
else
print(0)
end
0
```

Vous pouvez aussi utiliser l'opérateur ternaire `- ? - : -` (attention aux espaces).

```
> x = 1
> y = x > 10 ? 1 : 5
5
```

2.8 Boucles : while et for

On écrit une boucle `while` comme suit:

```
> x = 0
> while x <= 3
print(x)
global x += 1
end
012
```

Le mot clef `global` est nécessaire ici car, sans lui, on ne peut pas modifier une variable globale dans un bloc (ici le bloc entre `while` et `end`). Ce ne sera plus nécessaire dans les fonctions car les variables seront locales.

On écrit une boucle `for` comme suit:

```
> for x in 1:5
print(x)
end
12345
> for x in [1,2,3,4,5]
print(x)
end
12345
```

On peut aussi écrire `=` au lieu de `in`. On peut enfin utilier $\in$ mais comment écrire ce symbole avec un clavier facilement?

On peut enfin utiliser `break` et `continue` comme dans de nombreux langages : le premier pour quitter la boucle et le second pour terminer l'itération en cours et passer à l'itération suivante.

2.9 Fonctions

Une fonction s'écrit comme suit:

```
> function f(x)
y = 2 * x
return x + y
end
```

Dans les cas simples comme ci-dessus, où on renvoie `3x`, on peut simplifier la notation:

```
> f(x) = 3 * x
```

ou encore avoir des fonctions anonymes (pour les intégrer facilement comme arguments d'une autre fonction par exemple):

```
> x -> 3 * x
```

Une fonction peut être récursive. Une fonction peut avoir plusieurs arguments. Une fonction peut renvoyer plusieurs éléments. Elle renvoie alors un *tuple* (voir la section ci-après).

2.10 Tableaux et autres stuctures de données

On définit et manipule un tableau comme suit

```
> ar = [1, 3, 2]
3-element Array{Int64,1}:
1
3
2
> length(ar)
3
> 4 in ar
false
> ar[2] = 4
4
> ar
3-element Array{Int64,1}:
1
4
2
> sort!(ar)
3-element Array{Int64,1}:
```

```
1
2
4
```

Comme vous pouvez le constater, l'indice des variables commence à 1 et non 0.

Attention, le type des éléments d'un tableau est défini à sa construction.

```
> ar = [1, 2, 3]
3-element Array{Int64,1}:
1
2
3
> ar[2] = "abc"
ERROR: MethodError: Cannot ‘convert‘ an object of type
String to an object of type Int64
> ar = Array{Any}([4, 5, 6])
3-element Array{Any,1}:
1
2
3
> ar[2] = "abc"
"abc"
> ar
3-element Array{Any,1}:
1
"abc"
3
```

Vous pouvez créer des tableaux de tableaux ou des tableaux multidimensionnels (ce n'est pas exactement la même chose). Le second correspond plutôt à une matrice (on peut les multiplier, les additionner, ..., ce qui n'est pas le cas du premier).

```
> ar = [[1, 2], [2, 3]]
2-element Array{Array{Int64,1},1}:
[1, 2]
[2, 4]
> ar2 = [[1 2]; [2 3]]
2x2 Array{Int64,2}:
```

```
1 2
2 3
> ar[1][2]
2
> ar2[1][2]
2
> ar2[1, 2]
2
> ar[1, 2]
2
ERROR: BoundsError: attempt to access 2-element
Array{Array{Int64,1},1} at index [1, 2]
> ar2 * ar2
2x2 Array{Int64,2}:
5 8
8 13
> ar * ar2
ERROR: DimensionMismatch("matrix A has dimensions (2,1),
matrix B has dimensions (2,2)")
```

Il est possible, comme en Python d'utiliser les *Lists comprehensions*.

```
> ar = [2 * i * j for i in 1:3 for j in 1:2 if i != j]
4-element Array{Int64,1}:
4
4
6
12
```

Pour modifier une liste, vous pouvez utiliser les fonctions suivantes:

- `push!(ar, x1, x2, ...)` pour ajouter un ou plusieurs éléments à `ar`
- `pop!(ar)` pour supprimer le dernier élément de `ar`
- `append!(ar, ar2)` pour ajouter tous les éléments de `ar2` à `ar`
- `pushfirst!(ar, x1, x2)` pour ajouter des éléments au début de `ar`
- `popfirst!(ar)` pour retirer le premier élément de `ar`

- `prepend!(ar, ar2)` pour ajouter tous les éléments de `ar2` au début de `ar`
- `insert!(ar, i, x)` pour insérer un élément à `ar` l'indice donné
- `deleteat!(ar, i)` pour supprimer l'élément d'indice `i` de `ar`

Pour itérer sur une liste, vous pouvez utiliser une boucle `for`.

```
> ar = [4, 5, 6]
> for x in ar
print(x)
end
456
> for (i, x) in enumerate(ar)
println(i, ' ', x)
end
1 4
2 5
3 6
```

Vous pouvez également créer des tuples (qu'on peut voir comme des tableaux non modifiables) comme en Python.

```
> t = (1, 2, 3)
(4, 2, 3)
> t[1]
4
> t[1] = 5
ERROR: MethodError: no method matching
setindex!(::Tuple{Int64,Int64,Int64}, ::Int64, ::Int64)
```

Vous pouvez créer des tables d'association, ou dictionnaire:

```
> d = Dict()
Dict{Any,Any} with 0 entries
> d["abc"] = 1
1
> d["abc"]
1
> d[2] = 4.0
4.0
```

```
> d[1]
ERROR: KeyError: key 1 not found
> "abc" in keys(d)
true
> 1 in values(d)
true
> for (x, y) in d
println(x, ' ', y)
end
abc 1
2 4.0
> length(d)
2
```

Vous pouvez créer des ensembles:

```
> s = Set([1, 2, 3])
Set([2, 3, 1])
> push!(s, 4)
Set([4, 2, 3, 1])
> push!(s, 3)
Set([4, 2, 3, 1])
```

Vous trouverez la liste complète des structures ici : `https://docs.julialang.org/en/v1/base/collections/`.

2.11 Passage par référence dans les fonctions

Les arguments sont passés par valeur et non par référence, comme en C, en Java ou en Python. Ca signifie, en gros, qu'il est impossible de modifier la valeur d'une variable dans une fonction.

```
> function incr(x)
x += 1
return
end
> v = 1
1
> incr(v)
> v
```

```
1
```

On peut toutefois modifier des contenants, comme par exemple un tableau. Par exemple

```
> function swap!(x)
x[1] = x[1] + x[2]
x[2] = x[1] - x[2]
x[1] = x[1] - x[2]
end
> ar = [1, 2]
2-element Array{Int64,1}:
1
2
> swap!(ar)
> ar
2-element Array{Int64,1}:
2
1
```

Remarque: le symbole `!` est juste indicatif, il n'est pas obligatoire. Par convention, comme c'est le cas en Ruby par exemple, une fonction qui aura un effet de bord (notamment la modification d'un objet en place) contient un point d'exclamation. Ainsi les fonctions `sort` et `sort!` font la même chose mais la première trie une copie d'un tableau et la renvoie, et la seconde trie le tableau directement.

Une autre manière de simuler le passage par référence est d'utiliser la possibilité de renvoyer plusieurs sorties:

```
> function strangerthings(a, b, c)
a = b * c
c = b * 2 * a
b = a * c + b
return a + b + c, a, b, c
end
> x, y, z = 1, 2, 3
> s, x, y, z = strangerthings(x, y, z)
```

Etant donné que le nombre de sorties n'est pas limité, il est toujours possible de renvoyer une ou plusieurs entrées en sortie pour la réaffecter à l'issue de la fonction. Par contre, on ne peut pas vraiment parler d'effet de

bord: l'affectation se fait en dehors de la fonction. Donc pas de `!` dans le nom de la fonction.

2.12 Strings

Comme dans beaucoup de langage, les chaînes se manipulent plus ou moins comme des tableaux.

```
> s = "abcde"
"abcde"
> s[2]
'b'
> 'd' in s
true
> 'f' in s
false
> s2 = "def"
"def"
> s * s2
"abcdedef"
> length(s)
5
> occursin("abc", s) # verifie si "abc" apparait dans s
true
> join([s, s2], ',')
"abcde,def"
> split(s, "c")
2-element Array{SubString{String},1}:
"ab"
"de"
```

Les chaînes ne peuvent être modifiées.

2.13 Aléatoire

Pour générer une valeur aléatoire, vous pouvez utiliser la fonction `rand`. Cette fonction accepte de nombreux paramètres en entrée:

```
> rand()
0.09398343011021293
```

```
> rand(1:4)
3
> rand(Int64)
3239622741057195049
> rand(Bool)
false
> rand([1, 5, 98, "abc"])
"abc"
> rand("abcde")
'd'
> rand(3)
3-element Array\{Float64,1\}:
0.27805324164891676
0.9586112470403574
0.9737864145278117
```

2.14 Lecture et écriture dans un fichier.

Sans trop rentrer dans les détails, vous pouvez lire et écrire dans un fichier ainsi:

```
> open("fichier.txt") do file
for l in eachline(file)
println(l)
end
end
... <---- affichage des lignes de "fichier.txt"
> open("fichier.txt", "w") do file
write(file, "Je remplace tout le fichier par cette ligne.\n")
end
> open("fichier.txt", "a") do file
write(file, "J'ajoute cette ligne a la fin du fichier. \n")
end
```

3 Programmation linéaire avec JuMP

JuMP est un langage de modélisation de programmation linéaire (mais aussi non linéaire) codé et compatible avec Julia. Il vous permet simplement d'implanter vos programmes et de les résoudre avec de nombreux solveurs. Ce n'est pas un solveur, juste un intermédiaire entre vous et les solveurs.

Des exemples se trouvent sur cette page: `https://github.com/JuliaOpt/JuMP.jl/tree/master/examples`. Toute la documentation de JuMP est fournie sur la page `www.juliaopt.org/JuMP.jl/0.18/`.

Ce tutoriel est essentiellement une redite partielle de ce qui est dit dans cette documentation. Libre à vous de la parcourir et de vous l'approprier.

3.1 Installer JuMP et des solveurs

Vous pouvez facilement installer des paquets pour Julia. Pour installer JuMP ainsi que des solveurs de programmation linéaire, il vous suffit de taper les commandes suivantes en mode console.

Pour installer JuMP et les solveur GLPK et Cbc, il vous suffit de taper les commandes suivantes.

```
> using Pkg
> Pkg.add("JuMP")
> Pkg.add("GLPK")
> Pkg.add("Cbc")
```

Il n'est pas nécessaire d'installer plusieurs solveurs. JuMP est compatibles avec de nombreux solveurs. Choisissez celui qui vous convient le mieux. Les deux ci-dessus conviendront très bien pour le projet et ne sont pas difficiles à installer. D'autres solveurs comme CPLEX par exemple nécessitent une installation préalable et indépendante de Julia (au delà de la portée de ce tutoriel).

Ces paquets sont installés dans le même dossier que Julia. Désinstaller/-Supprimer Julia supprimera également tous les paquets installés.

3.2 Mon programme est long

Le chargement des bibliothèques de programmation linéaire est long. Ceci arrivant à chaque fois que vous exécutez votre script, il peu mettre entre 3 et

10 secondes à renvoyer son résultat. C'est trop long si on souhaite débugguer confortablement.

Il existe une alternative: exécuter votre script en mode console avec la commande `include` comme expliqué dans la section 1.4. Ainsi la bibliothèque ne se chargera qu'une seule fois, lors de la première exécution. Attention, si vous fermez le mode console, il faudra recharger de nouveau les paquets.

3.3 Créer un modèle

Contrairement au reste du tutoriel, les exemples sont présentés sans le symbole > (donc pas en mode console) pour que vous puissiez les écrire directement dans un script.

Un modèle est une variable de type `Model` auquel on adjoint un solveur.

```
using JuMP
using Cbc
m = Model(solver=CbcSolver())
```

ou

```
using JuMP
using GLPKMathProgInterface
m = Model(solver=GLPKSolverMIP())
```

Vous pouvez ensuite afficher le programme avec la commande

```
print(m)
```

3.4 Ajouter des variables

Chaque variable, comme dans tout programme linéaire possède un nom et un domaine, défini par des bornes et son intégrité.

Pour ajouter 3 variables, x, y et z respectivement réelle entre -1 et $+\infty$, entière entre -3 et +5 et binaire, il suffit de taper les commandes suivantes:

```
@variable(m, -1 <= x)
@variable(m, -3 <= y <= 5, Int)
@variable(m, z, Bin)
```

Il est aussi possible de définir les bornes avec une syntaxe plus lourde (l'intérêt viendra plus loin):

```
@variable(m, y, Int, lowerbound=-3, upperbound=5)
```

x, y et z sont les noms des variables, c'est également le nom qui apparaîtra si vous affichez le modèle (voir plus loin).

Il est donc important de ne pas coder vos variables ainsi:

```
@variable(m, -1 <= x)
@variable(m, -3 <= x <= 5, Int)
@variable(m, x, Bin)
```

Vous auriez alors 3 variables avec le même nom, ce qui n'est pas pratique. De plus vous perdriez la référence des 2 premières variables.

Si vous n'aimez pas le nom de votre variable, vous pouvez le changer:

```
@variable(m, -1 <= x, basename="superman")
```

Vous pouvez toujours utiliser x pour gérer votre variable mais lorsqu'elle sera affichée en console, x sera remplacé par `superman`.

Si vous êtes plutôt du genre à affecter vos variables, JuMP vous autorise également la syntaxe suivante

```
y = @variable(m, lowerbound=-1, category = :Int)
```

Par contre vous ne profitez plus de la syntaxe visuelle `-1 <= y` et aucun nom n'est associé par défaut à y dans le programme lorsqu'il est affiché.

Si vous voulez définir de nombreuses variables d'un coup, par exemple $x_i \in [0, 3]$ pour tout $i \in [\![1; n]\!]$, vous pouvez utiliser des tableaux pour vos variables.

```
n = 10
@variable(m, 0 <= x[1:n] <= 3)
```

`x` est alors un tableau avec n éléments accessibles comme avec n'importe quel tableau en julia. Il est fortement déconseillé de modifier le tableau soit même pour éviter toute inconsistance dans le modèle.

Que faire si vos variables n'ont pas le même ensemble de définition? Par exemple $x_i \in [-i, 2i]$.

```
n = 10
@variable(m, -i <= x[i in 1:n] <= 2i)
```

Si vos bornes commencent à être très complexes et à varier en fonction de i, utiliser la syntaxe plus lourde et faire appel à des fonctions peut rendre le code plus lisible:

```
n = 10
@variable(m, x[i in 1:n], lowerbound=lbx(i), upperbound=ubx(i))
```

Enfin, les entiers ne sont pas les seuls indices possibles et il est possible de metttre plus qu'un indice:

```
s = ["hey", "hoo", "haaa"]
@variable(m, x[i in 1:3, j in s], lowerbound=lbx(i), upperbound=ubx(i, j))
```

3.5 Ajouter des contraintes

On suppose qu'on dispose de 3 types de variables numériques : x réelle, y_i entière pour $i \in [\![1;3]\!]$ et z_j binaire pour $j \in [\![1;6]\!]$.

On peut générer ses contraintes ainsi :

```
@constraint(m, x + y[1] >= 0)
@constraint(m, x + y[2] >= 0)
@constraint(m, x + y[3] >= 0)
```

On peut donner des noms à ses contraintes:

```
@constraint(m, cons1, x + y[1] >= 0)
```

ou

```
cons1 = @constraint(m, x + y[1] >= 0)
```

Encore une fois, attention à ne pas donner le même nom à deux contraintes.

On peut faire tout d'un coup:

```
@constraint(m, cons[i in 1:3], x + y[i] >= 0)
```

On peut aussi gérer des sommes simplement:

```
@constraint(m, sumisone, sum(z[i] for i in 1:6) == 1)
```

On peut bien entendu mettre des constantes à gauche et des variables à droite.

```
@constraint(m, sumiswow, sum(z[i] for i in 1:6) - 12 == x)
```

3.6 Fonction objectif

Générer un objectif est plus ou moins la même chose que générer une contrainte. Il faut juste préciser si on souhaite minimiser ou maximiser et ne pas mettre de membre à droite du symbole.

```
@objective(m, Max, sum(y[i] + z[2 * i] for i in 1:3 if i % 2 == 0))
```

3.7 Résoudre le programme

Pour résoudre le programme, vous pouvez appeler la commande

```
solve(m, relaxation=false)
```

Le paramètre `relaxation` est facultatif (et vaut `false` par défaut). S'il est vrai, alors le solveur résout la relaxation continue.

Vous pouvez récupérer le statut de la résolution en sortie:

```
status = solve(m)
```

La variable `status` peut alors avoir les 6 valeurs suivantes: `:Optimal`, `:Unbounded`, `:Infeasible`, `:UserLimit`, `:Error`, `:NotSolved`. (Attention aux deux-points, ils sont importants.)

Vous pouvez ensuite récupérer les valeurs de l'objectif ou des variables:

```
getobjectivevalue(m)
getvalue(z[3])
```

Pour récupérer le temps, il y a deux méthodes. Si votre solveur le permet (et ce n'est le cas ni de Cbc, ni de GLPK), vous pouvez effectuer :

```
getsolvetime(m)
```

Sinon, vous pouvez utiliser la fonction elapsed, quel que soit le solveur mais ce temps inclu l'appel au solveur.

```
t = @elapsed(status = solve(m))
```

3.8 Modifier le programme

Vous pouvez avoir envie de modifier le programme une fois celui-ci résolu, pour rajouter une variable, une contrainte, modifier une contrainte,...

Rajouter une variable ou une contrainte en utilisant les fonctions précédentes est possible, même après avoir appelé la fonction solve.

Pour faire de la génération de colonne, il faut pouvoir insérer les variables dans des contraintes déjà définies ou dans la fonction objectif. C'est possible avec la commande suivante:

```
@variable(m, x, objective = 2, inconstraints =
[cons1, cons2], coefficients = [1, 2])
```

Cette commande ajoute la variable x au modèle, de coût 2 et de coefficient 1 et 2 dans les contraintes cons1 et cons2 respectivement.

Vous ne pouvez pas supprimer de contrainte du modèle. Vous ne pouvez pas non plus modifier le coefficient d'une variable. Pour cela, il faut reconstruire entièrement le modèle. Par contre vous pouvez modifier le membre constant d'une contrainte :

```
JuMP.setRHS(cons1, 4)
```

Enfin, vous pouvez modifier la fonction objectif entièrement, il suffit de rappeler la fonction `@objective`.