

Initiation à julia

Mickaël Canouil
mickael.canouil@cnrs.fr

Génomique Intégrative et Modélisation des Maladies Métaboliques (CNRS UMR 8199)
Institut de Biologie de Lille

5 février 2016




Sommaire

- 1 Introduction
- 2 Les bases de Julia
- 3 Mise en pratique
- 4 Calcul parallèle
- 5 Les paquets
- 6 Appel de fonctions
- 7 Aller plus loin avec Julia
- 8 Références

Sommaire

1 Introduction

Présentation

 **julia** est un langage (2013) sous licence libre et open source **MIT**, destiné au calcul scientifique.



- Langage compilé à la volée (JIT compiler).
- Calcul parallèle et distribué.
- Performance proche des langages compilés comme le C.
- Support des caractères Unicode, incluant l'UTF-8.
- Appel de fonctions C et Fortran sans API extérieure (C++ via paquet).
- Appel de fonctions Python via le paquet **PyCall**.
- Macros et Metaprogrammation.

Julia et R : Similarités


- "Julia is a **high-level**, high-performance **dynamic programming** language for technical computing."
 - High-level** Fonctionne sur des éléments atomiques, des vecteurs, matrices, listes, etc.
 - Dynamic programming** Présence de type/classe. Les fonctions peuvent être définies de façon interactive.
- "**defining functions and overloading** them for different combinations of argument types"
- "syntax that is **familiar** to users of other technical computing environments"

 **julia** offre les mêmes possibilités que  **R** (benchmark à suivre), mais surtout des performances proches du C/C++

Julia et R : Syntaxe

		
Affectation	=	<-
Vecteur	[1, 2, 3]	c(1, 2, 3)
Transposition	M'	t(M)
Parenthèses	for i in [1, 2, 3] if i==1	for (i in c(1, 2, 3)) if (i==1)
Dimension	size(M, 1) size(M, 2)	nrow(M) ncol(M)
Concatenation	hcat vcat	c rbind cbind
Type NULL	X	✓
Commentaire	#	#

Installer Julia


 est disponible en téléchargement sur :

<http://julialang.org/downloads/>

Au 23 juin 2015, la version stable est la version 0.3.9

Windows Self-Extracting Archive (.exe)	32/64-bit
Mac OS X Package (.dmg)	10.7+ 64-bit
Ubuntu packages	32/64-bit
Fedora/RHEL/CentOS packages (.rpm)	32/64-bit
Source	Tarball/GitHub

Interagir avec Julia : le mode non-interactif

 propose plusieurs options de démarrage aussi bien pour le mode interactif que non-interactif.

```
julia [options] [program] [args...]
-v, --version           Display version information
-h, --help              Print this message
-q, --quiet             Quiet startup without banner
-H, --home <dir>       Set location of julia executable


-e, --eval <expr>       Evaluate <expr>
-E, --print <expr>      Evaluate and show <expr>
-P, --post-boot <expr> Evaluate <expr>, but don't disable interactive mode
-L, --load <file>       Load <file> immediately on all processors
-J, --sysimage <file>   Start up with the given system image file
-C --cpu-target <target> Limit usage of cpu features up to <target>

-p <n>                  Run n local processes
--machinefile <file>    Run processes on hosts listed in <file>

-i                     Force isinteractive() to be true
--no-history-file       Don't load or save history
-f, --no-startup        Don't load ~/.juliarc.jl
-F                      Load ~/.juliarc.jl, then handle remaining inputs
--color={yes|no}        Enable or disable color text

--code-coverage={none|user|all}, --code-coverage
                        Count executions of source lines (omitting setting is equivalent to
                        'user')
--track-allocation={none|user|all}
                        Count bytes allocated by each source line
--check-bounds={yes|no} Emit bounds checks always or never (ignoring declarations)
--int-literals={32|64}  Select integer literal size independent of platform
```


Interagir avec Julia : le mode non-interactif

Les commandes ou scripts (*.jl) peuvent également être lancés avec  en mode non-interactif.

```
$ julia -e "for x in ARGS; println(x); end" Hello Bye
Hello
Bye

$ echo 'for x in ARGS; println(x); end' > script.jl # sans guillemets sous Windows
$ julia script.jl Hello Bye
Hello
Bye
```

Interagir avec Julia : le mode interactif

En tapant `julia` dans un terminal sous Linux, une bannière s'affiche présentant des informations quant à la version installée.

```
$ julia
```

```
 _           _ _  (-)       | A fresh approach to technical computing  
(-)         | (-) (-)      | Documentation: http://docs.julialang.org  
_ _ _ _ _  | | _ _ _ _    | Type "help()" to list help topics  
| | | | |  | | / _ _ _   |  
| | | | |  | | (-) _ _   | Version 0.3.9 (2015-05-30 11:24 UTC)  
_| / \ \ ' _ | | _ _ \ _'| Official http://julialang.org/ release  
| _ /        | i686-w64-mingw32
```

```
julia> println("This is a Julia command!")  
This is a Julia command!
```

```
julia> quit()
```

Pour quitter une session interactive **julia**, il suffit d'utiliser le raccourci clavier **Ctrl+D**, de taper l'une des commandes : `quit()` ou `exit([code])`.

Interagir avec Julia : le mode interactif

- `help(quit) ?quit`
afficher l'aide de la fonction `quit`.
- `apropos("quit")`
cherche la documentation pour la fonction `quit`.
- `whos()`
liste les variables globales et leur type.
- `cd("/home/") & pwd()`
change et renvoie le répertoire courant.
- `include("script.jl") & require("script.jl")`
exécute le fichier systématiquement ou s'il ne l'a pas encore été.
- `clipboard(X)`
copie X dans le presse-papier.
- `workspace()`
réinitialise l'espace de travail.

Interagir avec Julia : le mode interactif

Les caractères `?` et `;` permettent d'ouvrir l'aide et l'invite de commande (bash linux). **julia** permet l'autocomplétion.

```
help?> quit
INFO Loading help data...
Base.quit()

Quit the program indicating that the processes completed
succesfully. This function calls "exit(0)" (see "exit()").


shell> echo 'x = 1; for x in ["A" 2]; println(x); end' > script.jl
```

```
julia> pwd()
"/home/mcanouil"

julia> include("script.jl")
A
2

julia> whos()
Base                Module
Core                Module
Main                Module
ans                 Nothing
x                   Int64
```

Le style, c'est important !

 étant sensible à la casse et afin de limiter les problèmes de syntaxe avec les noms de variables, types et fonctions, une convention de nommage est recommandée :

- Noms de variables en minuscule (ex : `mavariable`).
- Le underscore `_` peut être utilisé (non recommandé) dans les noms de variables (ex : `ma_variable_est_complique_et_longue`).
- Les mots composant le nom d'un type, commencent par une majuscule (ex : `MonTypePerso`).
- Le nom des fonctions et des macros est en minuscule (ex : `somme`).
- Les fonctions modifiant leurs arguments doivent finir par un **!** (ex : `plusun!`).

Sommaire

2 Les bases de Julia

Les opérateurs

Opérateur	Description
<code>+</code> <code>-</code>	Addition, soustraction
<code>*</code> <code>/</code>	multiplication, division
<code>%</code>	modulo
<code>&&</code> <code> </code>	ET, OU
<code>==</code> <code>!=</code>	EGAL, DIFFERENT
<code><</code> <code>></code>	INFERIEUR, SUPERIEUR
<code>.Op</code> or <code>broadcast (Op, x, y)</code>	Distribution de <code>Op</code> (vectoriel)

Les opérateurs

```
julia> x = [1 2 3]
1x3 Array{Int64,2}:
 1  2  3

julia> 2x + 1 # le signe de multiplication
                optionnelle (cf. perl)
1x3 Array{Int64,2}:
 3  5  7

julia> y = [10, 5, 21]
3-element Array{Int64,1}:
 1
 2
 3

julia> x + y
ERROR: dimensions must match
       in promote_shape at operators.jl:191
       in + at array.jl:723

julia> x + y'
1x3 Array{Int64,2}:
 2  4  6
```

```
julia> x .+ y
3x3 Array{Int64,2}:
 2  3  4
 3  4  5
 4  5  6

julia> broadcast(+, x, y)
3x3 Array{Int64,2}:
 2  3  4
 3  4  5
 4  5  6

julia> true || false
true

julia> true && true
true

julia> 1 < 2 < 3
true

julia> [1 2 3] .< [4 1 6]
1x3 BitArray{2}:
 true false true
```


Les numériques

 fonctionne sur un système d'évaluation d'expressions :

```
julia> 2 + 2
4

julia> 1/3
0.3333333333333333

julia> 1//3 # Nombre rationnel
1//3


julia> 1/3 == float(1//3)
true

julia> eps(Float64)
2.220446049250313e-16
```

Les numériques spéciaux : **Inf**, NaN

- `isequal(x, y)` `x` et `y` sont identiques
- `isfinite(x)` `x` est un nombre fini
- `isinf(x)` `x` est infini
- `isnan(x)` `x` n'est pas un nombre

Les variables

Les variables peuvent être nommées de façon très (trop ?) flexible dans .

```
julia> valeurPI = "Que j'aime a faire connaitre un nombre utile aux sages"
"Que j'aime a faire connaitre un nombre utile aux sages"

julia> valeurpi = 3.1415926535
3.1415926535

julia>  $\pi$ 
 $\pi$  = 3.1415926535897...

julia> pi
 $\pi$  = 3.1415926535897...

julia> ans
 $\pi$  = 3.1415926535897...
```

Attention à ne pas utiliser des noms de fonctions existantes comme nom de variables. (Le message d'avertissement ne s'affiche pas sur tous les systèmes.)

```
julia> pi = 1
Warning imported binding for pi overwritten in module Main
1
```

Les chaînes de caractères


julia permet également d'interpoler et d'intégrer des calculs dans une chaîne de caractères.

```
julia> x = 8; machaine = "Le sens de la vie? $(2(x-1)^2 - 3(x+9) - 5) "  
"Le sens de la vie? 42"  
  
julia> string(machaine, " (H2G2) ")  
"Le sens de la vie? 42 (H2G2) "  
  
julia> "machaine" * " (H2G2) "  
"Le sens de la vie? 42 (H2G2) "
```

Quelques fonctions utiles pour manipuler les chaînes de caractères :

- `search(str, substr)` Renvoie l'indice (position) de `substr`.
- `length(str)` Renvoie le nombre de caractères.
- `endof(str)` Renvoie le dernier indice (byte) utilisé dans `str`.
- `start(str)` Renvoie le premier indice valide.

Les tuples

 dispose de la structure "**tuple**". Chaque valeur est séparée par une virgule `,` et délimitée par des parenthèses `(` et `)`.

```
julia> ()  
()  
  
julia> x = (1, 2)  
(1,2)  
  
julia> x[3]  
ERROR: BoundsError()
```

Il est possible de réattribuer les valeurs d'un **tuple** à plusieurs variables.

```
julia> a, b = x  
(1,2)  
  
julia> a  
1  
  
julia> b  
2
```

Construire des tableaux

La création d'un tableau peut s'exécuter de plusieurs façons.

```
julia> [1, 2, 3, 4, 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> [i for i = 1:5] # ou [1:5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

```
julia> [1 2 3 4 5]
1x5 Array{Int64,2}:
 1 2 3 4 5

julia> [i+j for i = 1:5, j = 1:5]
5x5 Array{Int64,2}:
 2 3 4 5 6
 3 4 5 6 7
 4 5 6 7 8
 5 6 7 8 9
 6 7 8 9 10
```

Concaténer des tableaux avec : **cat**, **hcat**, **vcat** et **[...]**.

```
julia> [[1:5] [6:10]]
5x2 Array{Int64,2}:
 1 6
 2 7
 3 8
 4 9
 5 10
```

```
julia> cat(2, [1:5], [6:10])
5x2 Array{Int64,2}:
 1 6
 2 7
 3 8
 4 9
 5 10
```

Accéder au contenu des tableaux

L'accès au contenu s'effectue avec des indices numériques et des symboles `:` et `end`.

```
julia> myarray = [i+j for i = 1:5, j = 1:5];
```

```
julia> myarray[1:5, 1:5]
5x5 Array{Int64,2}:
 2  3  4  5  6
 3  4  5  6  7
 4  5  6  7  8
 5  6  7  8  9
 6  7  8  9 10
```

```
julia> myarray[1:5, 1:end]
5x5 Array{Int64,2}:
 2  3  4  5  6
 3  4  5  6  7
 4  5  6  7  8
 5  6  7  8  9
 6  7  8  9 10
```


```
julia> myarray[1:2, :]
2x5 Array{Int64,2}:
 2  3  4  5  6
 3  4  5  6  7
```

```
julia> myarray[4:end, :]
2x5 Array{Int64,2}:
 5  6  7  8  9
 6  7  8  9 10
```

Les fonctions de base des tableaux

- `eltype(A)`
le type d'élément contenu dans A.
- `length(A)`
le nombre d'éléments de A.
- `ndims(A)`
le nombre de dimensions de A.
- `countnz(A)`
le nombre de valeurs différentes de zéro dans A.
- `size(A)`
un tuple contenant les dimensions de A.
- `size(A, n)`
la taille de A dans la dimension n.

Les constructeurs de tableaux

 fournit de nombreuses fonctions pour construire et initialiser des tableaux :

Fonction	Description
<code>Array(type, dims...)</code>	un tableau non initialisé (<code>type=Any</code>)
<code>cell(dims...)</code>	une cellule de tableau (hétérogène) non initialisé
<code>zeros(type, dims...)</code>	tableau de zéros du type spécifié
<code>ones(type, dims...)</code>	tableau de uns du type spécifié
<code>trues(dims...)</code>	tableau de booléens <code>true</code>
<code>false(dims...)</code>	tableau de booléens <code>false</code>
<code>eye(m, n)</code> ou <code>eye(n)</code>	matrice identité de dimensions (m,n) ou (n,n)
<code>fill!(A, x)</code>	remplit le tableau A avec x

Déclaration d'un type à Julia

Le mot clé `type` permet la déclaration d'un type, tandis que l'opérateur `::`, permet l'affectation de type aux expressions et variables.

```
julia> type MonType
    x::Int
    y::Float64
end


julia> p = MonType(2, 3)
MonType(2,3.0)

julia> typeof(ans)
MonType (constructor with 1 method)

julia> names(p)
2-element Array{Symbol,1}:
 :x
 :y

julia> p.x
2
```

Les dictionnaires

 permet également de définir des dictionnaires :

```
julia> Dict{Float64, Int64}() # Dictionnaire vide
Dict{Float64,Int64} with 0 entries

julia> (Float64=>Int64)[1.1=>1, 2=>2] # Dictionnaire contenant deux entrees
Dict{Float64,Int64} with 2 entries:
 1.1 => 1
 2.0 => 2

julia> dico = {"key"=>1, (2,3)=>true}
Dict{Any,Any} with 2 entries:
 "key" => 1
 (2,3) => true

julia> dico[(2, 3)] # Acces a la valeur du dictionnaire pour la cle (2, 3)
true
```

➤ `haskey(dico, "key")` Teste si dico contient la clé "key".

➤ `keys(dico) & values(dico)`
Renvoie les clés et valeurs de dico.

➤ `delete!(dico, (2, 3))` Efface la clé (2, 3) et sa valeur du dictionnaire.

Les blocs d'expressions

Les commandes **begin** et **end** permettent de définir des blocs d'expressions à évaluer, qui pourront être ensuite affectés à une variable par exemple.

```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
3

julia> z = begin x = 1; y = 2; x + y end
3
```

Les blocs peuvent également être construits en utilisant :

- (Ouvre un bloc d'expressions
- ; Délimite les sous expressions
-) Ferme un bloc d'expressions

```
julia> z = (x = 1;
           y = 2;
           x + y)
3

julia> z = (x = 1; y = 2; x + y)
3
```

Attention à ne pas inverser le , et le ; !

Les tests de conditions

Les tests de conditions s'effectuent avec les commandes :

if, **else**, **end** et **elseif**.

```
if x <= y
    "x <= y"
else
    "x > y"
end
```


```
if x < y
    "x < y"
elseif x > y
    "x > y"
else
    "x = y"
end
```

Et en une seule ligne ... (? et)

```
x <= y ? "x <= y" : "x > y"
```

```
x < y ? "x < y" : x > y ? "x > y" : "x = y"
```

Les boucles

La syntaxe du **for** est sensiblement la même que dans , mais sans les parenthèses.

```
for i = 1:5
    println(i)
end
```

```
for i in [1:5]
    println(i)
end
```

```
for i in [1, 2, 3, 4, 5]
    println(i)
end
```

Et tant qu'on y est, le **while** :

```
i = 1;
while i <= 5
    println(i)
    i += 1
end
```

Les boucles

La commande **break** permet d'interrompre l'exécution d'un **for** ou d'un **while**, à une itération donnée.

Inversement, la commande **continue** force le passage à l'itération suivante.

```
julia> for i = 1:3
           i == 2 ? break : println(i)
       end
1
```

```
julia> for i in 1:3
           i == 2 ? continue : println(i)
       end
1
3
```

Plusieurs boucles **for** peuvent être combinées en une seule boucle :

```
julia> for i = ["A", "B"]
           for j = [1:2]
               println((i,j))
           end
       end
("A",1)
("A",2)
("B",1)
("B",2)
```

```
julia> for i in ["A", "B"], j in [1:2]
           println((i,j))
       end
("A",1)
("A",2)
("B",1)
("B",2)
```

Définir une fonction

Contrairement aux blocs d'expressions, les fonctions peuvent être réutilisées sans qu'il ne soit nécessaire de les redéfinir.

Deux façons de définir de nouvelles fonctions :


➤ Multi-lignes

```
function f(x, y)
    x + y
end
```

➤ Mono-ligne

```
f(x, y) = x + y
```

Renvoyer le résultat d'une fonction

Comme dans , le résultat de la dernière expression d'une fonction est retourné, à moins d'un appel à `return`.

```
julia> function f(x,y)
           x * y
           return x + y
       end
f (generic function with 1 method)

julia> f(2,3)
5
```

```
julia> function f(x,y)
           return x * y
           x + y
       end
f (generic function with 1 method)

julia> f(2,3)
6
```

De plus, les résultats retournés par la fonction, peuvent être affectés directement à plusieurs variables.

```
julia> myfunction(x,y) = (x,y);

julia> a,b = myfunction(2,3)
(5, 6)

julia> a
5

julia> b
6
```


Fonction et type

Dans une fonction, l'opérateur `:` permet de contraindre les paramètres au type spécifié.

```
julia> f(x::Int64) = print("x est un entier: ", x)
f (generic function with 1 method)

julia> f(7)
x est un entier: 7

julia> f("chaine")
ERROR: no method f{ASCIIString}
```

Gérer les ambiguïtés

La définition de méthodes spécifiques à certains types peut engendrer des ambiguïtés, signalées par **julia** sous la forme d'un message d'avertissement précisant où se situent celles-ci.

```
julia> f(x::Float64, y) = 2x + y
f (generic function with 1 method)

julia> f(x, y::Float64) = x + 2y
Warning: New definition
  f(Any,Float64) at none1
is ambiguous with
  f(Float64,Any) at none1.
To fix, define
  f(Float64,Float64)
before the new definition.
f (generic function with 2 methods)

julia> methods(f)
# 2 methods for generic function "f":
f(x::Float64,y) at none:1
f(x,y::Float64) at none:1

julia> f(x::Float64, y::Float64) = 2x + 2y
f (generic function with 3 methods)
```

Sommaire

3 Mise en pratique

Exemple : Générer des données

Projet : Générer des données de méthylation

- 30 individus (en vrai quelques centaines)
- 200 sites CpG (en vrai entre 400 000 et plusieurs millions)
- La méthylation est définie entre 0 et 1

```
julia> simule((2, 3), 10)
12x5 Array{Any,2}:
  "Individu1"  "Individu2"  "Individu3"  "Individu4"  "Individu5"
  "cas"        "cas"        "controle"   "controle"   "controle"
1.0           0.863474    0.256478     0.185314     0.000256711
1.0           1.0         0.720053     0.0156981    0.641579
1.0           0.350158    0.880605     0.114643     0.727741
0.466751      0.388357     0.433628     0.50445      0.0804214
1.0           0.830145    0.453638     0.527395     0.0874052
0.46995       1.0         0.235512     0.241468     0.128219
1.0           0.740111    0.126524     0.993061     0.149074
0.766059      0.467199     0.986665     0.928067     0.719145
0.240649      0.463098     0.840223     0.0552221    0.397078
0.518303      1.0         0.837655     0.248769     0.679063
```

Fonctions utiles : `rand`, `min`, `hcat`, `vcat`, `transpose`, `'` et `help`.

Exemple : Générer des données

Fonctions à écrire

`genere()` Ecrire une fonction `genere()` qui prend deux arguments entiers (`nind` et `ncpg`) et renvoie un tableau (`nind` \times `ncpg`) de nombres aléatoires compris entre 0 et 1 (`rand`).

Exemple : Générer des données

Fonctions à écrire

`genere()` Ecrire une fonction `genere()` qui prend deux arguments entiers (`nind` et `ncpg`) et renvoie un tableau (`nind` x `ncpg`) de nombres aléatoires compris entre 0 et 1 (`rand`).

```
help?> rand
INFO Loading help data...
Base.rand() -> Float64

Generate a "Float64" random number uniformly in [0,1)
...
```

Exemple : Générer des données

Fonctions à écrire

`genere()` Ecrire une fonction `genere()` qui prend deux arguments entiers (`nind` et `ncpg`) et renvoie un tableau (`nind` x `ncpg`) de nombres aléatoires compris entre 0 et 1 (`rand`).

```
help?> rand
INFO Loading help data...
Base.rand() -> Float64

Generate a "Float64" random number uniformly in [0,1)
...
```

```
julia> genere(5, 7)
5x7 Array{Float64,2}:
 0.588669  0.570379  0.916509  0.0297843  0.358949  0.845641  0.767147
 0.292024  0.672616  0.374952  0.751985  0.538212  0.117445  0.800937
 0.27846   0.425266  0.78022  0.605692  0.256313  0.438527  0.821982
 0.691418  0.254037  0.932809  0.0853761  0.332398  0.457604  0.992759
 0.246875  0.214584  0.374868  0.974698  0.55114  0.369378  0.7708
```

Exemple : Générer des données

Fonctions à écrire

`genere()` Ecrire une fonction `genere()` qui prend deux arguments entiers (`nind` et `ncpg`) et renvoie un tableau (`nind` \times `ncpg`) de nombres aléatoires compris entre 0 et 1 (`rand`).

```
help?> rand
INFO Loading help data...
Base.rand() -> Float64

Generate a "Float64" random number uniformly in [0,1)
...
```

```
julia> genere(5, 7)
5x7 Array{Float64,2}:
0.588669  0.570379  0.916509  0.0297843  0.358949  0.845641  0.767147
0.292024  0.672616  0.374952  0.751985  0.538212  0.117445  0.800937
0.27846   0.425266  0.78022   0.605692  0.256313  0.438527  0.821982
0.691418  0.254037  0.932809  0.0853761  0.332398  0.457604  0.992759
0.246875  0.214584  0.374868  0.974698  0.55114   0.369378  0.7708
```

```
function genere(nind::Int, ncpg::Int)
    rand(nind, ncpg)
end
```

```
genere(nind::Int, ncpg::Int) = rand(nind,
    ncpg)
```


Exemple : Générer des données

Fonctions à écrire

`effet()` Ecrire une fonction qui prend une matrice **M** (`Array{Float64, 2}`) et un flottant **beta** en arguments.

Cette fonction renvoie la somme de **M+beta** bornée par **1** (valeur maximum).

Exemple : Générer des données

Fonctions à écrire

`effet()` Ecrire une fonction qui prend une matrice **M** (`Array{Float64, 2}`) et un flottant **beta** en arguments.

Cette fonction renvoie la somme de **M+beta** bornée par **1** (valeur maximum).

```
help?> min  
Base.min(x, y, ...)
```

```
Return the minimum of the arguments. Operates elementwise over arrays.
```

Exemple : Générer des données

Fonctions à écrire

effet() Ecrire une fonction qui prend une matrice **M** (**Array{Float64, 2}**) et un flottant **beta** en arguments.

Cette fonction renvoie la somme de **M+beta** bornée par **1** (valeur maximum).

```
help?> min  
Base.min(x, y, ...)
```

Return the minimum of the arguments. Operates elementwise over arrays.

```
julia> effet([0.0 for i in 1:5, j in 1:7],  
0.5)  
5x7 Array{Float64,2}:  
0.5 0.5 0.5 0.5 0.5 0.5 0.5  
0.5 0.5 0.5 0.5 0.5 0.5 0.5  
0.5 0.5 0.5 0.5 0.5 0.5 0.5  
0.5 0.5 0.5 0.5 0.5 0.5 0.5  
0.5 0.5 0.5 0.5 0.5 0.5 0.5
```

```
julia> effet([0 for i in 1:5, j in 1:7], 2)  
5x7 Array{Int64,2}:  
1 1 1 1 1 1 1  
1 1 1 1 1 1 1  
1 1 1 1 1 1 1  
1 1 1 1 1 1 1  
1 1 1 1 1 1 1
```

Exemple : Générer des données

Fonctions à écrire

`effet()` Ecrire une fonction qui prend une matrice **M** (`Array{Float64, 2}`) et un flottant **beta** en arguments.

Cette fonction renvoie la somme de **M+beta** bornée par **1** (valeur maximum).

```
help?> min
Base.min(x, y, ...)
```

Return the minimum of the arguments. Operates elementwise over arrays.

```
julia> effet([0.0 for i in 1:5, j in 1:7],
0.5)
5x7 Array{Float64,2}:
0.5 0.5 0.5 0.5 0.5 0.5 0.5
0.5 0.5 0.5 0.5 0.5 0.5 0.5
0.5 0.5 0.5 0.5 0.5 0.5 0.5
0.5 0.5 0.5 0.5 0.5 0.5 0.5
0.5 0.5 0.5 0.5 0.5 0.5 0.5
```

```
function effet(M::Array{Float64, 2},
    beta::Float64)
    min(M+beta, 1)
end
```

```
julia> effet([0 for i in 1:5, j in 1:7], 2)
5x7 Array{Int64,2}:
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
```

```
effet(M::Array{Float64, 2}, beta::Float64) =
    min(M+beta, 1)
```

Exemple : Générer des données

Fonctions à écrire

`simule()` Ecrire une fonction **simule** qui renvoie un tableau de données.

```
function simule(nind::(Int,Int), ncpg::Int)
    ...
end
```

```
julia> simule((2, 3), 10)
12x5 Array{Any, 2}:
  "Individu1"  "Individu2"  "Individu3"  "Individu4"  "Individu5"
  "cas"        "cas"        "controle"   "controle"   "controle"
1.0           0.863474    0.256478     0.185314     0.000256711
1.0           1.0        0.720053     0.0156981    0.641579
1.0           0.350158    0.880605     0.114643     0.727741
0.466751      0.388357    0.433628     0.50445      0.0804214
1.0           0.830145    0.453638     0.527395     0.0874052
0.46995       1.0        0.235512     0.241468     0.128219
1.0           0.740111    0.126524     0.993061     0.149074
0.766059      0.467199    0.986665     0.928067     0.719145
0.240649      0.463098    0.840223     0.0552221    0.397078
0.518303      1.0        0.837655     0.248769     0.679063
```

Fonctions utiles : **genere**, **effet**, **rand**, **min**, **hcat**, **vc****at**, **transpose**, **'** et **help**.

Exemple : Générer des données

Fonctions à écrire

`simule()` Génère **nind** cas et contrôles (**tuple**) pour **nepg** sites avec **genere ()**

Exemple : Générer des données

Fonctions à écrire

`simule()` Génère **nind** cas et contrôles (**tuple**) pour **ncpg** sites avec **genere()**

```
function simule(nind::(Int,Int), ncpg::Int)
```

```
    dns = genere(sum(nind), ncpg)
```

Exemple : Générer des données

Fonctions à écrire

`simule()` Génère **nind** cas et contrôles (**tuple**) pour **ncpg** sites avec **genere()**

```
function simule(nind::(Int,Int), ncpg::Int)
```

```
    dns = genere(sum(nind), ncpg)
```

Ajoute un **effet()** pour les cas
(sous ensemble de la matrice **dns**)

Exemple : Générer des données

Fonctions à écrire

`simule()` Génère **nind** cas et contrôles (**tuple**) pour **ncpg** sites avec **genere()**

```
function simule(nind::(Int,Int), ncpg::Int)
```

```
    dns = genere(sum(nind), ncpg)
```

Ajoute un **effet()** pour les cas
(sous ensemble de la matrice **dns**)

```
    ncas, ncontrole = nind  
    dns[1:ncas, :] = effet(dns[1:ncas, :], 0.2)
```

Fonctions utiles : **genere**, **effet**, **rand**, **min**, **hcat**, **vcate**, **transpose**, **'** et **help**.

Exemple : Générer des données

Fonctions à écrire

`simule()` Renvoie un tableau avec deux lignes d'entêtes et un échantillon par colonne :

Exemple : Générer des données

Fonctions à écrire

`simule()` Renvoie un tableau avec deux lignes d'entêtes et un échantillon par colonne :

- Nom (ligne 1) : "Individu1", "Individu2", ...

```
julia> nind = (2,3);  
  
julia> individus = ["Individu$i" for i in 1:sum(nind)]  
5-element Array{Union{UTF8String,ASCIIString},1}:  
"Individu1"  
"Individu2"  
"Individu3"  
"Individu4"  
"Individu5"
```

Exemple : Générer des données

Fonctions à écrire

`simule()` Renvoie un tableau avec deux lignes d'entêtes et un échantillon par colonne :

- Nom (ligne 1) : "Individu1", "Individu2", ...

```
julia> nind = (2,3);

julia> individus = ["Individu$i" for i in 1:sum(nind)]
5-element Array{Union{UTF8String,ASCIIString},1}:
"Individu1"
"Individu2"
"Individu3"
"Individu4"
"Individu5"
```

- Statut (ligne 2) : "cas" ou "contrôle"

```
julia> statuts = [{"cas" for i in 1:ncas},
                  {"contrôle" for i in 1:ncontrôle}]
5-element Array{ASCIIString,1}:
"cas"
"cas"
"contrôle"
"contrôle"
"contrôle"
```

Exemple : Générer des données

Résultats

```
function simule(nind::(Int,Int), ncp::Int)
    dns = genere(sum(nind), ncp)
    ncas, ncontrole = nind
    dns[1:ncas, :] = effet(dns[1:ncas, :], 0.2)

    individus = ["Individus$i" for i in 1:sum(nind)]
    statuts = [{"cas" for i in 1:ncas}, {"controle" for i in 1:ncontrole}]

    return transpose(hcat(individus, statuts, dns))
end
```

Exemple : Générer des données

Résultats



```
function simule(nind::(Int,Int), ncpg::Int)
    dns = genere(sum(nind), ncpg)
    ncas, ncontrole = nind
    dns[1:ncas, :] = effet(dns[1:ncas, :], 0.2)

    individus = ["Individu$i" for i in 1:sum(nind)]
    statuts = [{"cas" for i in 1:ncas}, {"controle" for i in 1:ncontrole}]

    return transpose(hcat(individus, statuts, dns))
end
```

```
julia> simule((2, 3), 10)
12x5 Array{Any,2}:
  "Individu1"  "Individu2"  "Individu3"  "Individu4"  "Individu5"
  "cas"        "cas"        "controle"   "controle"   "controle"
1.0           0.863474    0.256478     0.185314     0.000256711
1.0           1.0        0.720053     0.0156981    0.641579
1.0           0.350158    0.880605     0.114643     0.727741
0.466751      0.388357     0.433628     0.50445      0.0804214
1.0           0.830145    0.453638     0.527395     0.0874052
0.46995       1.0        0.235512     0.241468     0.128219
1.0           0.740111    0.126524     0.993061     0.149074
0.766059      0.467199     0.986665     0.928067     0.719145
0.240649      0.463098     0.840223     0.0552221    0.397078
0.518303      1.0        0.837655     0.248769     0.679063
```

Lire et écrire des données

		Description
<code>readdlm</code>	<code>read.table</code>	Lire un fichier txt selon un délimiteur
<code>writedlm</code>	<code>write.table</code>	Ecrire un fichier txt selon un délimiteur
<code>readcsv</code>	<code>read.csv</code>	Lire un fichier csv
<code>writecsv</code>	<code>write.csv</code>	Ecrire un fichier csv

Lire et écrire des données

Ecrire des données

```
dta = simule((14, 16), 200);  
writedlm("methylation.txt", dta, '*')
```

Attention à la gestion des caractères ' et des chaînes de caractères ".

Lire un fichier

```
julia> readlm("methylation.txt", "*")  
ERROR `readlm` has no method matching readlm{ASCIIString, ASCIIString}
```

```
readlm("methylation.txt", '*')
```


Exemple : Décrire des données

Projet : Décrire des données de méthylation

- Donner la moyenne, l'écart-type, le maximum, le minimum et la médiane
 - par individu
 - par statut : cas/contrôle

```
julia> resume(dta)
33x6 Array{Any,2}:
" "          "moy"      "min"      "max"      "med"      "std"
"cas"        0.683906  0.200082   1.0        0.709797  0.262965
"contrôle"   0.502301  0.000103171 0.999583  0.508735  0.285948
"Individu1"  0.687317  0.206128   1.0        0.733917  0.257906
"Individu2"  0.66751  0.201064   1.0        0.696693  0.269757
"Individu3"  0.693711  0.200507   1.0        0.719138  0.266323
"Individu4"  0.721926  0.201448   1.0        0.785435  0.259657
"Individu5"  0.640261  0.200885   1.0        0.640462  0.267654
"Individu6"  0.662807  0.200082   1.0        0.674797  0.267656
...
```

Exemple : Décrire des données

Fonctions à écrire

`summarystat()` Définir une fonction qui renvoie un tableau (moyenne, écart-type, maximum, minimum et la médiane).

- Deux arguments `x` (`Array{Float64, 2}`) et `dim` (`Int`), où `dim` peut prendre les valeurs :
 - `0` pour le tableau complet,
 - `1` pour les colonnes,
 - `2` pour les lignes.

Fonctions utiles : `mean`, `minimum`, `maximum`, `median`, `std`, `transpose`, `cat`, `hcat`, `vcut` et `help`.

Exemple : Décrire des données

Le code

summarystat()

```
function summarystat(x::Array{Float64, 2}, dim::Int)
    if dim==0
        res = hcat(
            mean(x), minimum(x), maximum(x),
            median(x), std(x)
        )
    else
        res = cat(dim,
            mean(x, dim), minimum(x, dim), maximum(x, dim),
            median(x, dim), std(x, dim)
        )
    end
    dim==1 ? transpose(res) : res
end
```

```
julia> summarystat(rand(25,4), 0)
1x5 Array{Float64,2}:
 0.460565  0.00419562  0.975303  0.444553  0.283187
```

```
julia> summarystat(rand(25,4), 1)
4x5 Array{Float64,2}:
 0.495807  0.0631534  0.964803  0.468451  0.288229
 0.511329  0.0181536  0.942708  0.482024  0.261304
 0.522599  0.0784808  0.998121  0.464263  0.326022
 0.431901  0.000887866  0.955217  0.411995  0.295526
```

Messages et erreurs

Parfois, il peut être utile d'afficher des informations relatives à des erreurs, avertissements ou de simples informations.

 propose trois commandes : `info`, `warn` et `error`.

```
julia> info("Bonjour!")  
INFO Bonjour!  
  
julia> warn("Bonjour!")  
WARNING Bonjour!  
  
julia> error("Bonjour!")  
ERROR Bonjour!  
in error at error.jl:21
```

Messages et erreurs

Il est également possible de tester et gérer les erreurs avec les commandes `try` et `catch`.

```
julia> x = -1
-1

julia> sqrt(-1)
ERROR: DomainError
      sqrt will only return a complex result if called with a complex argument.
      try sqrt(complex(x)) in sqrt at math.jl:280

julia> try
          sqrt(x)
        catch
          sqrt(complex(x,0))
        end
0.0 + 1.0im
```

Exemple : Décrire des données

Fonctions à écrire

Définir une nouvelle fonction `isascii()` qui renvoie `false` au lieu d'une erreur.

```
julia> isascii('a')
true

julia> isascii(1)
ERROR: 'isascii' has no method matching isascii{Int64}
```

Fonctions utiles : `try`, `catch` et `isascii`.

Exemple : Décrire des données

Fonctions à écrire

Définir une nouvelle fonction `isascii()` qui renvoie `false` au lieu d'une erreur.

```
julia> isascii('a')
true

julia> isascii(1)
ERROR: 'isascii' has no method matching isascii{Int64}
```

```
isascii2(x) = try isascii(x) catch; false end
```

```
julia> isascii2('a')
true

julia> isascii2(1)
false
```

Fonctions utiles : `try`, `catch` et `isascii`.

Exemple : Décrire des données

Fonctions à écrire

`resume()` Définir une fonction permettant de

- 1 lire l'entête,
- 2 calculer les statistiques de base (`summarystat()`) selon le statut (cas / contrôle) et par échantillon.

```
julia> dta = simule((14, 16), 200); resume(dta)
33x6 Array{Any,2}:
" "      "moy"      "min"      "max"      "med"      "std"
"cas"      0.683906  0.200082   1.0        0.709797  0.262965
"controle" 0.502301  0.000103171 0.999583  0.508735  0.285948
"Individu1" 0.687317  0.206128   1.0        0.733917  0.257906
"Individu2" 0.66751  0.201064   1.0        0.696693  0.269757
"Individu3" 0.693711  0.200507   1.0        0.719138  0.266323
"Individu4" 0.721926  0.201448   1.0        0.785435  0.259657
"Individu5" 0.640261  0.200885   1.0        0.640462  0.267654
"Individu6" 0.662807  0.200082   1.0        0.674797  0.267656
...

```

Fonctions utiles : `findin`, `broadcast`, `map`, `isascii`, `float`, `hcat`, `vcate` et `help`.

Exemple : Décrire des données

Fonctions à écrire

Identifier les lignes d'entête (ligne 1 et 2) d'un **Array** avec la fonction **isascii2()**.

```
julia> M = simule((14, 16), 3);  
  
julia> isascii2(M[:, 1])  
false
```

Fonctions utiles : **findin**, **broadcast**, **isascii2** et **help**.

Exemple : Décrire des données

Fonctions à écrire

Identifier les lignes d'entête (ligne 1 et 2) d'un **Array** avec la fonction **isascii2()**.

```
julia> M = simule(14, 16, 3);  
  
julia> isascii2(M[:, 1])  
false
```

```
julia> testascii = broadcast(isascii2, M[:, 1])  
5-element Array{Any,1}:  
 true  
 true  
false  
false  
false
```

Fonctions utiles : **findin**, **broadcast**, **isascii2** et **help**.

Exemple : Décrire des données

Fonctions à écrire

Identifier les lignes d'entête (ligne 1 et 2) d'un **Array** avec la fonction **isascii2()**.

```
julia> M = simule(14, 16, 3);  
  
julia> isascii2(M[:, 1])  
false
```

```
julia> testascii = broadcast(isascii2, M[:, 1])  
5-element Array{Any,1}:  
 true  
 true  
false  
false  
false
```

```
julia> findin(testascii, true)  
2-element Array{Int64,1}:  
 1  
 2
```

Fonctions utiles : **findin**, **broadcast**, **isascii2** et **help**.

Exemple : Décrire des données

Fonctions à écrire

Extraire l'entête et les données.

```
testascii = broadcast(isascii2, M[:, 1])  
entete = M[findin(testascii, true), :]  
data = float(M[findin(testascii, false), :])
```

Fonctions utiles : `findin`, `broadcast`, `isascii2` et `help`.

Exemple : Décrire des données

Fonctions à écrire

Extraire l'entête et les données.

```
testascii = broadcast(isascii2, M[:, 1])  
entete = M[findin(testascii, true), :]  
data = float(M[findin(testascii, false), :])
```

Extraire les **noms** (ligne 1) et les **statuts** (ligne 2) de l'entête.

Fonctions utiles : **findin**, **broadcast**, **isascii2** et **help**.

Exemple : Décrire des données

Fonctions à écrire

Extraire l'entête et les données.

```
testascii = broadcast(isascii2, M[:, 1])  
entete = M[findin(testascii, true), :]  
data = float(M[findin(testascii, false), :])
```

Extraire les **noms** (ligne 1) et les **statuts** (ligne 2) de l'entête.

```
noms = entete[1, :]  
statuts = entete[2, :]
```

Fonctions utiles : **findin**, **broadcast**, **isascii2** et **help**.

Exemple : Décrire des données

Fonctions à écrire

Extraire l'entête et les données.

```
testascii = broadcast(isascii2, M[:, 1])  
entete = M[findin(testascii, true), :]  
data = float(M[findin(testascii, false), :])
```

Extraire les **noms** (ligne 1) et les **statuts** (ligne 2) de l'entête.

```
noms = entete[1, :]  
statuts = entete[2, :]
```

Identifier les individus **cas** et les individus **controle**.

Fonctions utiles : **findin**, **broadcast**, **isascii2** et **help**.

Exemple : Décrire des données

Fonctions à écrire

Extraire l'entête et les données.

```
testascii = broadcast(isascii2, M[:, 1])  
entete = M[findin(testascii, true), :]  
data = float(M[findin(testascii, false), :])
```

Extraire les **noms** (ligne 1) et les **statuts** (ligne 2) de l'entête.

```
noms = entete[1, :]  
statuts = entete[2, :]
```

Identifier les individus **cas** et les individus **controle**.

```
indexcas = findin(statuts=="cas", true)  
indexcontrole = findin(statuts=="controle", true)
```

Fonctions utiles : **findin**, **broadcast**, **isascii2** et **help**.

Exemple : Décrire des données

Fonctions à écrire

Utiliser la fonction `summarystat()` pour obtenir les informations relatives aux cas, contrôles et l'ensemble des individus.

Fonctions utiles : `cat`, `vcats`, `hcat` et `summarystat`.

Exemple : Décrire des données

Fonctions à écrire

Utiliser la fonction `summarystat()` pour obtenir les informations relatives aux cas, contrôles et l'ensemble des individus.

```
tmp = vcat(  
  ["moy" "min" "max" "med" "std"], # hcat("moy", "min", "max", "med", "std")  
  summarystat(data[:, indexcas], 0),  
  summarystat(data[:, indexcontrole], 0),  
  summarystat(data, 1)  
)  
  
resultat = hcat(  
  transpose(["" "cas" "controle" noms]), # transpose(hcat("", "cas", "controle", noms))  
  tmp  
)
```

Fonctions utiles : `cat`, `vcart`, `hcat` et `summarystat`.

Exemple : Décrire des données

Le code

```
function resume (M::Array)
    testascii = broadcast(isascii2, M[:, 1])
    entete = M[findin(testascii, true), :]
    data = float(M[findin(testascii, false), :])

    noms = entete[1, :]
    statuts = entete[2, :]
    indexcas = findin(statuts=="cas", true)
    indexcontrole = findin(statuts=="controle", true)

    tmp = vcat(
        ["moy" "min" "max" "med" "std"], # vcat("moy", "min", "max", "med", "std")
        summarystat(data[:, indexcas], 0),
        summarystat(data[:, indexcontrole], 0),
        summarystat(data, 1)
    )



    resultat = hcat(
        transpose([" " "cas" "controle" noms]), # hcat("", "cas", "controle", noms)
        tmp
    )
    return resultat
end
```

```
julia> resume(dta)
33x6 Array{Any,2}:
" "          "moy"          "min"          "max"          "med"          "std"
"cas"        0.686297      0.200113      1.0           0.711756      0.263421
"controle"   0.505942      0.00178939    0.998642      0.508479      0.289279
"Individul"  0.663394      0.20072       1.0           0.676587      0.265681
"Individu2"  0.68806       0.2047        1.0           0.708784      0.260872
...
```

Sommaire

4 Calcul parallèle

Lancer Julia en parallèle

Par défaut,  est lancé sur un seul cœur (CPU core unit) ou processeur, il faut donc spécifier au démarrage le nombre de cœurs que  va pouvoir exploiter avec l'option `-p n`.

```
$ julia -p 2

julia> r = remotecall(2,rand,3,3) # Execute la construction d'une matrice (3,3) par le cpu/core 2
RemoteRef{2,1,26}

julia> fetch(r) # Recuperation du resultat de r dans l'environnement du cpu/core 2
3x3 Array{Float64,2}:
 0.635725  0.785651  0.790699
 0.653473  0.659035  0.872854
 0.333211  0.334218  0.623318

julia> s = @spawnat 2 2*fetch(r) # Execution de l'operation 2*fetch(r) sur le cpu/core 2
RemoteRef{2,1,28}

julia> fetch(s) # Recuperation du resultat de s dans l'environnement du cpu/core 2
3x3 Array{Float64,2}:
 1.27145  1.5713  1.5814
 1.30695  1.31807  1.74571
 0.666422 0.668436 1.24664
```

Les boucles

La macro `@parallel` [`reducteur`] `for` permet l'exécution de boucle en parallèle.

Cette macro nécessite l'utilisation d'une fonction de réduction pour concaténer les résultats.

```
julia> ntirages = @parallel (+) for i in [1:1000000]
    int(randbool())
end
500430
```

Pour conserver l'ordre des résultats, il faut utiliser la macro `@sync`.

Les boucles

`pmap` est à utiliser lorsque les calculs à effectuer sont importants en ressources et dans le cas où chaque calcul est indépendant.

`pmap` se charge d'attribuer les tâches aux différents cœurs.

```
julia> pmap(x -> x.>0.5, [rand(1) for i in 1:40000])
40000-element Array{Any,1}:
 Bool[false]
 Bool[true]
 Bool[true]
 ...
 Bool[false]
 Bool[true]
 Bool[true]
```

Exemple : Des calculs en parallèle

Fonctions à écrire

`resume()` Réécrire la fonction `resume` en utilisant `pmap` ou `@parallel for`.

Paralléliser l'exécution des commandes suivantes de la fonction `resume` :

```
tmp = vcat(  
  ["moy" "min" "max" "med" "std"],  
  summarystat(N[:, indexcas], 0),  
  summarystat(N[:, indexcontrole], 0),  
  summarystat(N, 1)  
)
```


Exemple : Des calculs en parallèle

Le code

```
function resume (M::Array)
    noms = M[1, :]

    statuts = M[2, :]
    indexcas = findin(statuts=="cas", true)
    indexcontrole = findin(statuts=="controle", true)

    testascii = broadcast(isascii2, M[:, 1])
    cherchenum = findin(testascii, false)
    N = float(M[cherchenum, :])

    tmp = vcat(
        ["moy" "min" "max" "med" "std"],
        summarystat(N[, indexcas], 0),
        summarystat(N[, indexcontrole], 0),
        summarystat(N, 1)
    )

    resultat = hcat(
        transpose(["" "cas" "controle" noms]),
        tmp
    )
    return resultat
end
```

Exemple : Des calculs en parallèle

Le code

```
function resume (M::Array, method::Int)
    noms = M[1,:]

    statuts = M[2,:]
    indexcas = findin(statuts=="cas", true)
    indexcontrole = findin(statuts=="controle", true)

    testascii = broadcast(isascii2, M[1, 1])
    cherchenum = findin(testascii, false)
    N = float(M[cherchenum,:])


    x1 = {N[:, indexcas], N[:, indexcontrole], N}
    x2 = [0 0 1]

    if method==1
        println("@parallel method")
        tmp = @sync @parallel vcat for i in [1:3]
            summarystat(x1[i], x2[i])
        end
    else
        println("pmap method")
        tmp = pmap(summarystat, x1, x2)
        tmp = reduce(vcat, tmp)
    end
    tmp = vcat(["moy" "min" "max" "med" "std"], tmp)

    resultat = hcat(
        transpose([" " "cas" "controle" noms]),
        tmp
    )
    return resultat
end
```

Exemple : Des calculs en parallèle

Processus et fonction

L'utilisation du mode parallèle de  nécessite d'exporter les fonctions dans les différents processus, pour ce faire il y a la macro `@everywhere`.

```
@everywhere function summarystat(x::Array, dim::Int)
    if dim==0
        res = hcat(
            mean(x), minimum(x), maximum(x),
            median(x), std(x)
        )
    else
        res = cat(dim,
            mean(x, dim), minimum(x, dim), maximum(x, dim),
            median(x, dim), std(x, dim)
        )
    end
    dim==1 ? transpose(res) : res
end
```

```
julia> res1 = resume(dta, 1);
@parallel method


julia> res2 = resume(dta, 2);
pmap method

julia> res1 == res2
true
```

Sommaire

5 Les paquets

Installer un paquet

L'installation d'un paquet répertorié par  s'effectue à l'aide de la fonction `Pkg.add()`.

<http://pkg.julialang.org/>

```
julia> Pkg.add("GLM")
INFO Cloning cache of Distributions from git://github.com/JuliaStats/Distributions.jl.git
INFO Cloning cache of GLM from git://github.com/JuliaStats/GLM.jl.git
INFO Cloning cache of NumericFuns from git://github.com/lindahua/NumericFuns.jl.git
INFO Cloning cache of PDMats from git://github.com/JuliaStats/PDMats.jl.git
INFO Installing Distributions v0.6.6
INFO Installing GLM v0.4.4
INFO Installing NumericFuns v0.2.3
INFO Installing PDMats v0.3.1
INFO Package database updated
INFO METADATA is out-of-date - you may not have the latest version of GLM
INFO Use `Pkg.update()` to get the latest versions of your packages
```

Le chargement d'un paquet installé se fait avec le mot clé `using`.

```
using GLM
```

Information et statut

L'état des paquets installés (nom et version) est donné par `Pkg.status()`.

```
Pkg.status()
```


La commande `Pkg.installed()` permet d'obtenir la même information sous forme d'un dictionnaire, pouvant être utilisé dans un script.

```
julia> Pkg.installed()  
Dict{ASCIIString, VersionNumber} ()
```

Pour obtenir les dernières versions de tous les paquets installés :

```
Pkg.update()
```

Désinstaller un paquet

La fonction `Pkg.clone()` permet d'installer un paquet non répertorié par .

```
Pkg.clone("git://example.com/path/to/Package.jl.git")
```

La désinstallation d'un paquet (répertorié ou non) s'effectue avec la fonction `Pkg.rm()`

```
Pkg.rm("GLM")
```

Exemple : Charger et utiliser un paquet

Chargement d'un paquet et édition du jeu de données simulées.

```
using GLM, DataFrames;

dta = simulate((14, 16), 200);
dta2 = convert(DataFrame, transpose(dta)); # Conversion de type

vrainom = [{"nom" "statut"} {"x$j" for i in 1, j in 1:size(dta2, 2)-2}]; # Noms des colonnes
rename!(dta2, names(dta2), convert(Array{Symbol, 2}, vrainom)); # Renomme les colonnes

dta2[:statut2] = int(dta2[:statut].=="cas"); # Ajoute une colonne statut2 en binaire
```

Utilisation de la fonction `fit(LinearModel, ...)` ou `lm` du paquet GLM.

```
julia> fit(LinearModel, x1~statut2, dta2) # Regression lineaire
DataFrameRegressionModel{LinearModel{DensePredQR{Float64}},Float64}:

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.491542  0.0702945  6.99262   <1e-6
statut2      0.280215  0.102901  2.72316   0.0110
```


Petit benchmark Julia et R

```
dta = simulate((140, 160), 1000);
dta2 = convert(DataFrame, transpose(dta));
vrainom = [{"nom" "statut"} [{"x$j" for i in 1, j in 1:size(dta2, 2)-2}]]];
rename!(dta2, names(dta2), convert(Array{Symbol, 2}, vrainom));
dta2[:,statut2] = int(dta2[:,statut].=="cas");
writetable("dta.txt", dta2);
```



```
> julia -p 2

@everywhere using GLM, DataFrames;
dta = readtable("dta.txt");
nombrevariable =
    sum(map((x)->ismatch(r"x[0-9]*",
    string(x)), names(dta)));
alltime = Float64[1:100];
for j in 1:100
    jtime = @elapsed @parallel heat for i in
        1:nombrevariable
            fit(LinearModel,
                eval(parse("x$i~statut2")), dta)
        end
    alltime[j] = jtime
end
```

```
julia> (mean(alltime), std(alltime))
(0.8263162251699999, 0.1461673308965699)
```



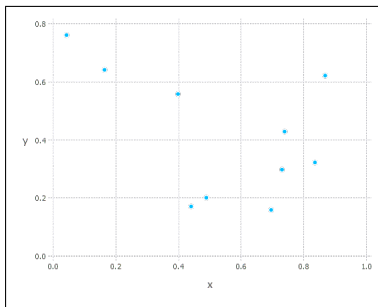
```
library(parallel)
dta = read.delim("dta.txt", sep = ",")
nombrevariable = length(grep("x[0-9]*",
    colnames(dta)))
alltime = 1:100
for (j in 1:100) {
    alltime[j] = system.time({
        mclapply(1:nombrevariable, mc.cores
            = 2, function(i) {
                eval(parse(text =
                    paste0("summary(lm(x", i, "~statut2,
                        data = dta)"))))
            })
    })["elapsed"]
}
```

```
R> c(mean(alltime), sd(alltime))
[1] 0.9881700 0.1910647
```


“Le dessin n'est pas la forme, il est la manière de voir la forme” - Edgar Degas

julia dispose de plusieurs paquets permettant de tracer des graphiques :
Winston, **Gadfly**, **PyPlot**, ...

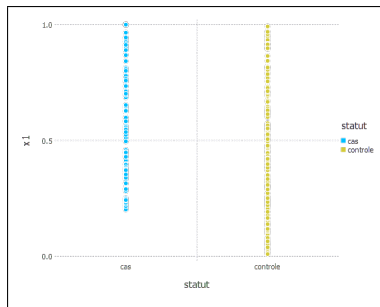
```
Pkg.add("Gadfly"); Pkg.add("Cairo");  
using Gadfly, Cairo;  
  
p = plot(x=rand(10), y=rand(10))  
draw(PNG("plot1.png", 7.5cm, 6cm), p) # Ecriture de l'image
```



“Le dessin n'est pas la forme, il est la manière de voir la forme” - Edgar Degas

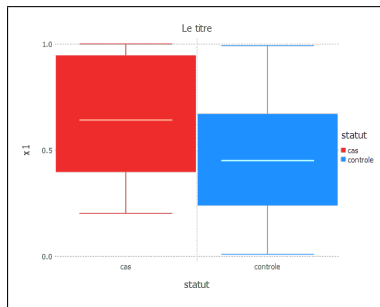
Le paquet **Gadfly** présente une syntaxe très proche de celle utilisée dans le package  : **ggplot2**.

```
p1 = plot(dta2, x = "statut", y = "x1", color = "statut")
```



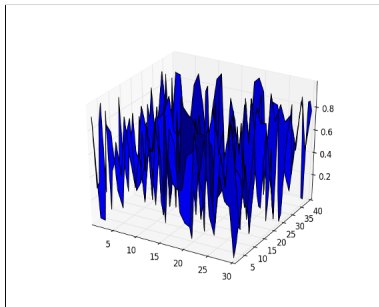
“Le dessin n'est pas la forme, il est la manière de voir la forme” - Edgar Degas

```
p2 = plot(  
  dta2, x = "statut", y = "x1", color = "statut", Geom.boxplot,  
  Scale.color_discrete_manual("firebrick2", "dodgerblue"),  
  Guide.title("Le titre")  
)
```



“Le dessin n'est pas la forme, il est la manière de voir la forme” - Edgar Degas

```
using PyPlot  
surf(rand(30,40))
```



Les modules

julia propose un système de "**module**" permettant d'encapsuler du code pour le (re)charger plus tard. Les paquets sont des modules.

```
julia> module MonModule
    export x
    x = 1
    y = 2 # variable cachee
end

julia> whos(MonModule)
MonModule          Module
x                  Int64

julia> names(MonModule)
2-element Array{Symbol,1}:
 :MonModule
 :x

julia> x
ERROR: x not defined

julia> (MonModule.x, MonModule.y)
(1,2)
```

Les modules

Pour (re)charger un module, il suffit d'utiliser la fonction **using** de la même façon que pour un paquet.

```
julia> using MonModule
```

```
julia> x
```

```
1
```

```
julia> y
```

```
ERROR y not defined
```

Il est également possible d'importer une fonction / variable d'un module, même si celle-ci n'a pas été exportée, via la fonction **import**.

```
julia> import MonModule.y
```

```
julia> y
```

```
2
```

Les macros

Les expressions (type **Expr**), du code non évalué, sont définies avec

```
:( ... )
```

Elles peuvent être évaluées via la fonction **eval()**.

```
julia> :( println("Hello, world!") )
:(println("Hello, world!"))

julia> typeof(ans)
Expr

julia> eval(:( println("Hello, world!") ))
Hello, world!
```

Les macros sont des fonctions commençant par le caractère **@** et utilisant les expressions.

Elles sont définies avec les commandes de début **macro** et de fin **end** :

```
julia> macro premieremacro(mot1, mot2)
    return :( println($mot1, $mot2) )
end
```


Utiliser une macro

Les macros peuvent être utilisées de deux façons :

```
julia> @premieremacro("Hello, ", "world!")  
Hello, world!  
  
julia> @premieremacro "Hello, " "world!"  
Hello, world!
```

Attention, la seconde méthode peut être ambiguë lorsque des tuples sont utilisés.

```
@name expr1 expr2 # deux arguments  
  
@name(expr1, expr2) # deux arguments  
  
@name (expr1, expr2) # un argument, un tuple de deux elements
```

Sommaire

6 Appel de fonctions

Exécuter des commandes externes

julia permet l'exécution de commandes externes :

- avec `run(`...`)`.

```
julia> run(`echo hello`)  
hello
```

- avec `;` pour passer en mode console shell.

```
julia> ;  
shell> echo hello  
hello  
  
julia>
```

Dans les deux cas, le résultat n'est pas renvoyé dans **julia**. Pour récupérer le résultat d'une commande, il y a la fonction `readall`.

```
julia> res = readall(`echo hello`)  
"hello\n"  
  
julia> res  
"hello\n"
```

Appel de fonctions Python

L'appel de fonctions Python s'effectue avec le paquet / module **PyCall**.


```
Pkg.add("PyCall")  
using PyCall
```

Les commandes **pyeval** et **pycall** permettent l'évaluation et l'exécution de commandes python, provenant d'une importation avec **@pyimport**.

```
julia> pyeval("1+1")  
2  
  
julia> @pyimport math # OR math = pyimport(:math)  
PyObject <module 'math' from '/usr/lib64/python2.7/lib-dynload/math.so'>  
  
julia> pycall(math["sin"], Float64, 1)  
0.8414709848078965  
  
julia> math.sin(math.pi / 4) - sin(pi / 4)  
0.0
```

```
@pyimport matplotlib.pyplot as plt  
x = linspace(0,2*pi,1000); y = sin(3*x + 4*cos(2*x));  
plt.plot(x, y, color="red", linewidth=2.0, linestyle="--");  
plt.show()
```

Appel de fonctions C

L'appel de fonctions C est possible nativement par .

Il passe par l'utilisation d'un **tuple** contenant les informations sur la fonction et la librairie dans laquelle elle se trouve.

```
(:function, "library")
```

```
("function", "library")
```

L'utilisation d'une fonction C passe par **ccall**

```
ccall(  
    (:function, "library"), # tuple d'importation de la fonction  
    ReturnType, # type du resultat renvoye par la fonction  
    (InputType) # un tuple des types fournies en entree de la fonction  
)
```

```
julia> t = ccall( (:clock, "libc"), Int32, () )  
34650000
```

```
julia> typeof(ans)  
Int32
```

Sommaire

7 [Aller plus loin avec Julia](#)

Aller plus loin avec Julia

- Ce document est disponible sur [GitHub](#)
- Le site officiel : <http://julialang.org/>
- La communauté : <http://julialang.org/community/>
- La documentation complète : <http://docs.julialang.org/en/latest/>
- Diverses sources d'informations : <http://julialang.org/learning/>

Sommaire

8 Références

Références



Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. (2012).

Julia : A fast dynamic language for technical computing.

arXiv preprint arXiv :1209.5145.