

Langage Julia

Cours ENPC - Pratique du calcul scientifique



- Langage de programmation créé en 2009 au MIT
- Par Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman
- Version 1.8.5 depuis le 08/01/2023
- Documentation générale de **Julia**: <https://docs.julialang.org/en/v1/>

Installation de **Julia** → deux options

1. Versions individuelles depuis
<https://julialang.org/>



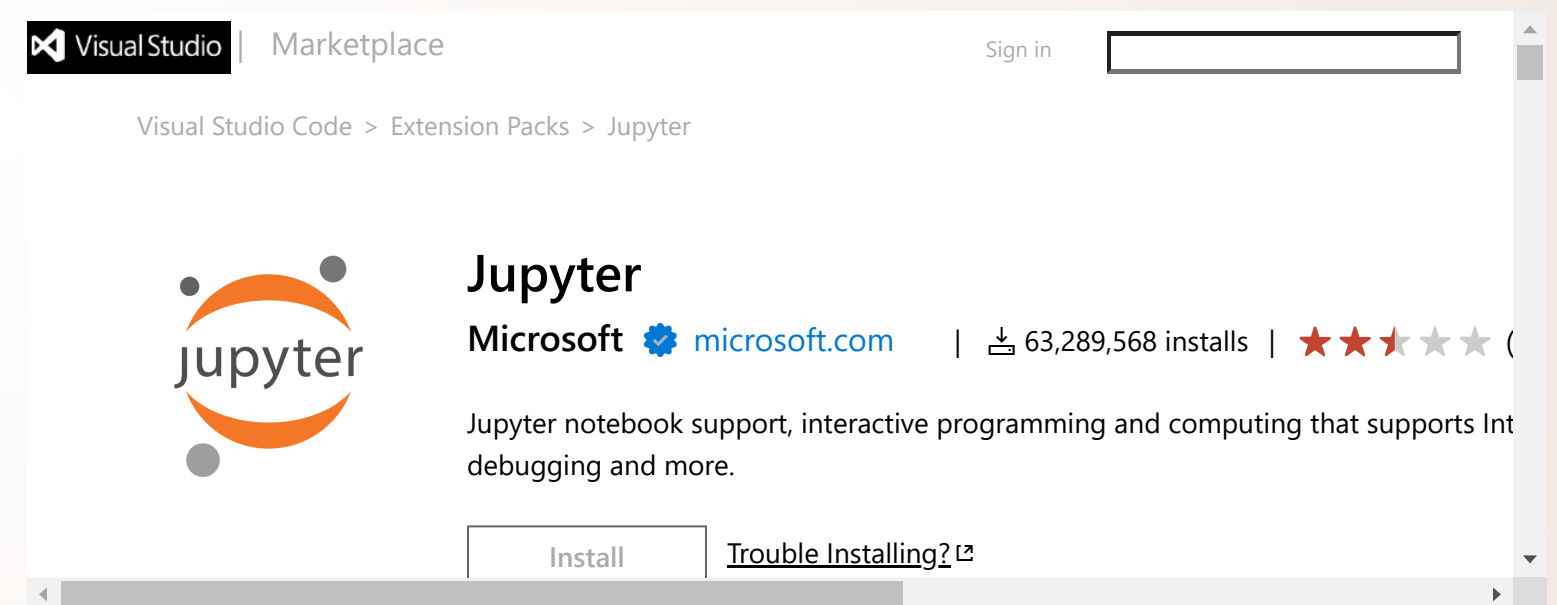
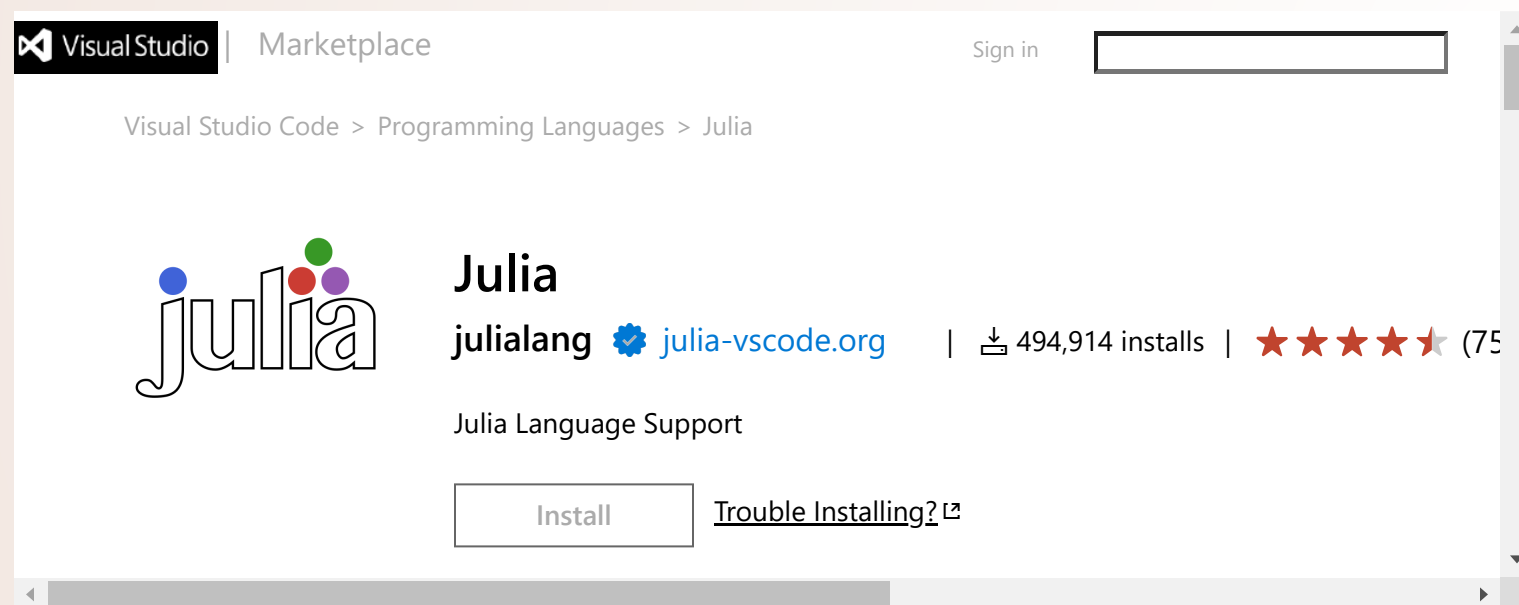
2. Gestionnaire de versions **juliaup** (future méthode officielle)

- installer **juliaup** depuis
<https://github.com/JuliaLang/juliaup>
- ajouter des versions, par exemple la dernière

```
PS [path] juliaup add release
```

Editeur de développement (IDE)

- Option préconisée : **VSCode** → <https://code.visualstudio.com/>
- Installer **Jupyter** → <https://jupyter.org/install>
- Installer la bibliothèque **IJulia** → <https://github.com/JuliaLang/IJulia.jl>
- Installer les extensions **Julia** et **Jupyter** pour **VSCode**

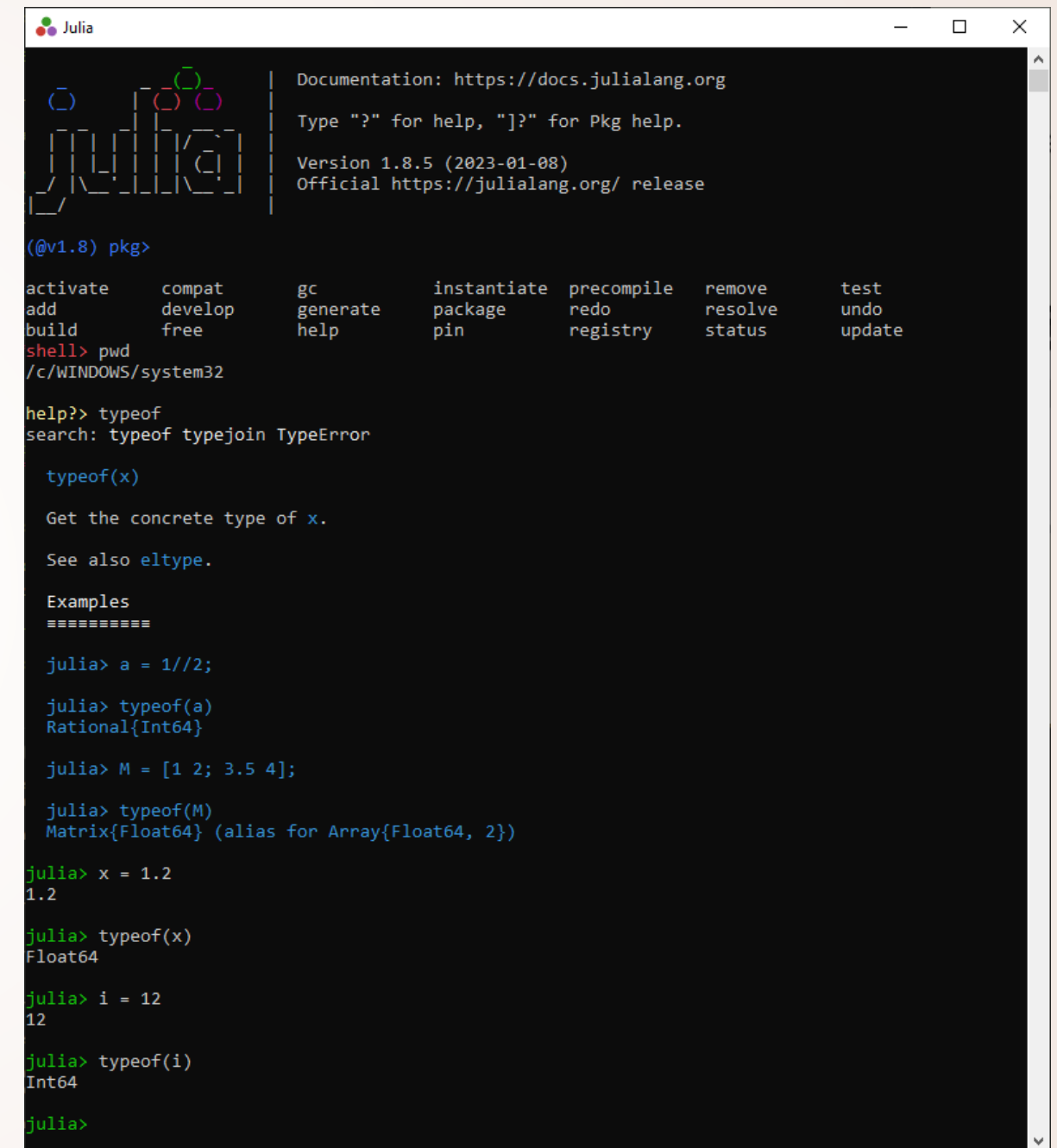


- Voir vidéos de D. Anthoff sur l'utilisation de **VSCode** pour **Julia**, p. ex. [lien 1](#) ou [lien 2](#)

Le REPL et quelques commandes importantes à connaître

- REPL = read-eval-print loop → console de Julia
- Le REPL est accessible
 - de manière autonome en tant qu'exécutable ou en tapant `julia` dans une console (si `PATH` à jour)
 - intégré dans `VSCode` en tapant `Maj+Ctrl+P` puis en cherchant `Julia: Start REPL`

Dans `VSCode`, on peut taper ses lignes de code dans un fichier `monfichier.jl` et les lancer une par une dans la console intégrée par `Maj+Entrée`.
- Depuis le REPL, taper
 - `]` donne accès au gestionnaire de bibliothèques
 - `;` donne accès au *shell mode*
 - `?` donne accès à l'aide en ligne
 - `←` retourne au mode normal
 - `Tab` pour la complétion automatique
 - les flèches haut et bas pour naviguer dans l'historique (éventuellement filtré par le début de ligne tapé)



```
Documentation: https://docs.julialang.org
Type "?" for help, "]" for Pkg help.

Version 1.8.5 (2023-01-08)
Official https://julialang.org/ release

(v1.8) pkg>
activate  compat  gc      instantiate  precompile  remove  test
add       develop  generate  package     redo         resolve  undo
build     free     help     pin         registry    status   update
shell> pwd
/c/WINDOWS/system32

help?> typeof
search: typeof typejoin TypeError

typeof(x)

Get the concrete type of x.

See also eltype.

Examples
=====

julia> a = 1//2;

julia> typeof(a)
Rational{Int64}

julia> M = [1 2; 3.5 4];

julia> typeof(M)
Matrix{Float64} (alias for Array{Float64, 2})

julia> x = 1.2
1.2

julia> typeof(x)
Float64

julia> i = 12
12

julia> typeof(i)
Int64

julia>
```

Pourquoi **Julia** ?

- **Julia** est facile à apprendre, à lire et à maintenir
- **Julia** est rapide
- **Julia** est efficace
- **Julia** est bien adapté au calcul scientifique

Julia est facile à apprendre, à lire et à maintenir

- Typage dynamique

```
x = 1 ; y = π ; z = 1//2 ; t = x + y + z
4.641592653589793

for v ∈ (x,y,z,t) println("$v is a $(typeof(v))") end
1 is a Int64
π is a Irrational{π}
1//2 is a Rational{Int64}
4.641592653589793 is a Float64
```

- Mais possibilité de typage statique

```
f(x) = 2x
f(x::Float64) = 3x
@show f(1)
@show f(1.) ;
```

```
f(1) = 2
f(1.0) = 3.0
```

- Définition simplifiée de fonctions courtes (équivalent du `lambda` de python)

- Bibliothèque de base contient les tableaux et leurs opérations courantes (+, -, *, \) puis `LinearAlgebra` pour `det`, `eigen`...

- Compréhension de tableau (`for` dans le tableau)

```
A = [63/(i+2j) for i ∈ 1:3, j ∈ 1:3]
```

```
3×3 Matrix{Float64}:
21.0  12.6  9.0
15.75 10.5  7.875
12.6   9.0  7.0
```

```
x = [1.2, 3.4, 5.6]
b = A*x
A\b
```

```
3-element Vector{Float64}:
1.20000000000000342
3.3999999999998263
5.6000000000000162
```

```
@show A\b == x
@show A\b ≈ x ;
```

```
A \ b == x = false
```

```
A \ b ≈ x = true
```

- Numérotation commence à 1 et remplissage par colonne

- Caractères unicode

```
G(x,μ=0.,σ=1.) = 1/σ√(2π) * exp(-(x-μ)^2/2σ^2)
```

```
G (generic function with 3 methods)
```

```
@show Σaij2 = sum(A.^2)
@show √Σaij2 ;
```

```
Σaij2 = sum(A .^ 2) = 1389.848125
√Σaij2 = 37.28066690658846
```

- Nouveaux opérateurs binaires

```
⊕(α,β) = α + β
5 ⊕ 7
```

```
12
```

```
⊕(α::Array{T,2}, β::Array{T,2}) where {T} = α + β'
```

```
⊕ (generic function with 2 methods)
```

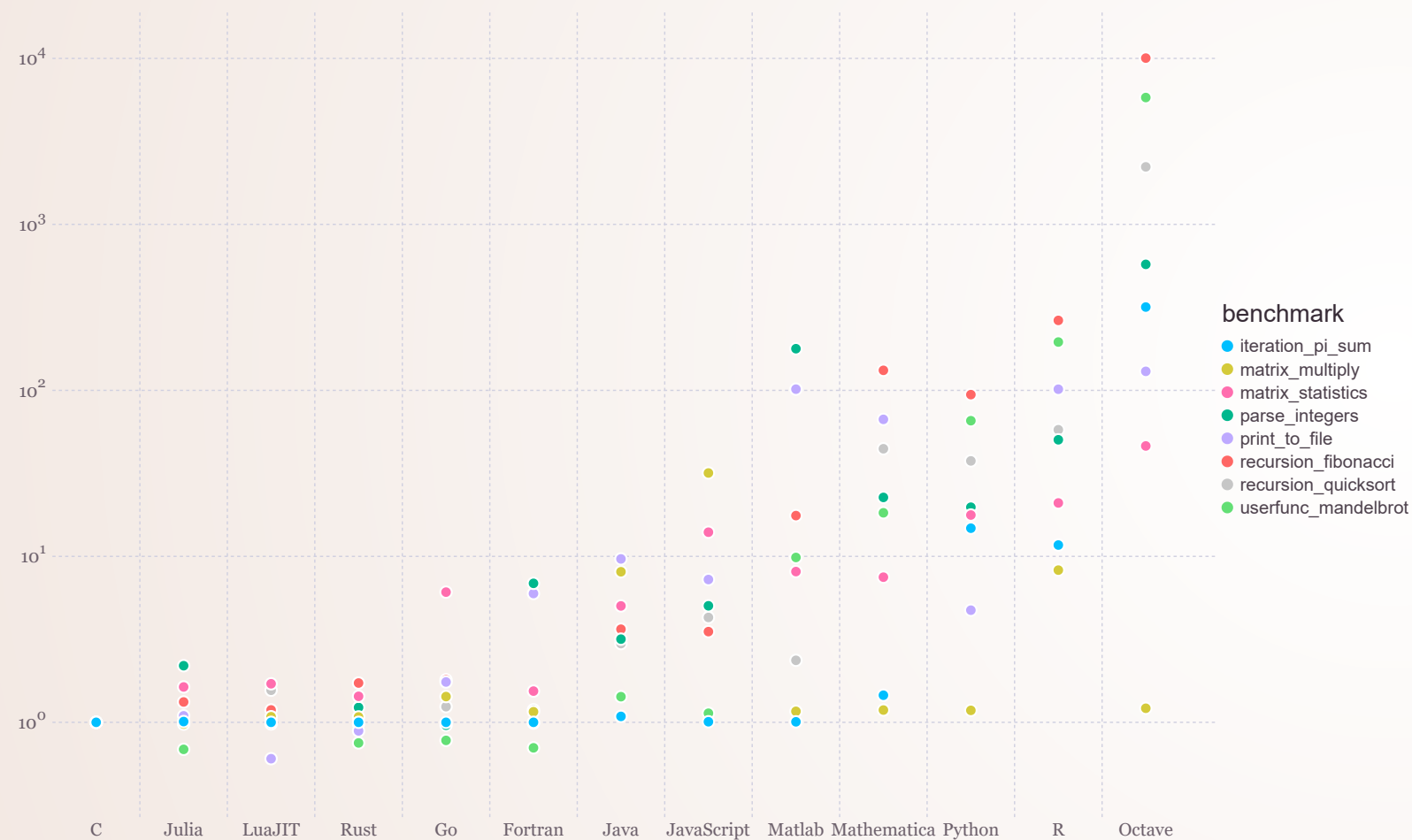
```
display(A + A)
display(A ⊕ A)
```

```
3×3 Matrix{Float64}:
42.0  25.2  18.0
31.5  21.0  15.75
25.2  18.0  14.0
```

```
3×3 Matrix{Float64}:
42.0  28.35  21.6
28.35  21.0  16.875
21.6  16.875  14.0
```

Julia est rapide

- Compilation à la volée (*Just-in-time compilation*)
- Parangonnage (cf. <https://julialang.org/benchmarks>)



The vertical axis shows each benchmark time normalized against the C implementation. The benchmark data shown above were computed with Julia v1.0.0, SciLua v1.0.0-b12, Rust 1.27.0, Go 1.9, Java 1.8.0_17, Javascript V8 6.2.414.54, Matlab R2018a, Anaconda Python 3.6.3, R 3.5.0, and Octave 4.2.2. C and Fortran are compiled with gcc 7.3.1, taking the best timing from all optimization levels (-O0 through -O3). C, Fortran, Go, Julia, Lua, Python, and Octave use OpenBLAS v0.2.20 for matrix operations; Mathematica uses Intel MKL. The Python implementations of matrix_statistics and matrix_multiply use NumPy v1.14.0 and OpenBLAS v0.2.20 functions; the rest are pure Python implementations. Raw benchmark numbers in CSV format are available under <https://github.com/JuliaLang/Microbenchmarks>.

These micro-benchmark results were obtained on a single core (serial execution) on an Intel Core i7-3960X 3.30GHz CPU with 64GB of 1600MHz DDR3 RAM, running openSUSE LEAP 15.0 Linux.

⚠ Ces résultats ne tiennent pas compte du temps de compilation.

Julia est efficace

Un atout majeur est le *multiple dispatch*

Exemple tiré de la conférence *The Unreasonable Effectiveness of Multiple Dispatch* de Stefan Karpinski

```
abstract type Animal end
struct Chien <: Animal; nom::String end
struct Chat <: Animal; nom::String end

function rencontre(a::Animal,b::Animal)
    verb = agit(a,b)
    println("$ (a.nom) rencontre $ (b.nom) et $verb")
end

agit(a::Chien,b::Chien) = "le renifle" ; agit(a::Chien,b::Chat) = "le chasse"
agit(a::Chat,b::Chien) = "s'enfuit" ; agit(a::Chat,b::Chat) = "miaule"

medor = Chien("Médor") ; 🐕 = Chien("🐕")
felix = Chat("Félix") ; 🐱 = Chat("🐱")

rencontre(medor, 🐕)
rencontre(🐕, felix)
rencontre(felix, 🐕)
rencontre(🐱, felix)
```

```
Médor rencontre 🐕 et le renifle
🐕 rencontre Félix et le chasse
Félix rencontre 🐕 et s'enfuit
🐱 rencontre Félix et miaule
```

Exemple adapté du [blog](#) de Mosè Giordano

```
abstract type HandShape end
struct Rock <: HandShape end
struct Paper <: HandShape end
struct Scissors <: HandShape end
play(::Type{Paper}, ::Type{Rock}) = println("Paper VS Rock ⇒ Paper wins")
play(::Type{Scissors}, ::Type{Paper}) = println("Scissors VS Paper ⇒ Scissors wins")
play(::Type{Rock}, ::Type{Scissors}) = println("Rock VS Scissors ⇒ Rock wins")
play(::Type{T}, ::Type{T}) where {T<: HandShape} = println("Tie between $(T), try again")
play(a::Type{<:HandShape}, b::Type{<:HandShape}) = play(b, a) # Commutativity
play(Paper, Rock)
play(Scissors, Scissors)
play(Paper, Scissors)
```

```
Paper VS Rock ⇒ Paper wins
Tie between Scissors, try again
Scissors VS Paper ⇒ Scissors wins
```

Il est aisé d'ajouter de nouveaux types *a posteriori*

```
struct Well <: HandShape end
play(::Type{Well}, ::Type{Rock}) = "Well VS Rock ⇒ Well wins"
play(::Type{Well}, ::Type{Scissors}) = "Well VS Scissors ⇒ Well wins"
play(::Type{Paper}, ::Type{Well}) = "Paper VS Well ⇒ Paper wins"
play(Scissors, Well)
```

```
"Well VS Scissors ⇒ Well wins"
```

Julia est bien adapté au calcul scientifique

- Différentiation automatique

```
using Zygote
f(x) = log(x)
f'(2), f''(2)

(0.5, -0.25)
```

- Calculs dimensionnels

```
using Unitful
m = u"m" ; cm = u"cm" ;
@show 1.0m + 1.0cm
println()
@show 1.0u"MPa" + 2.0u"bar" + 3.0u"daN/cm"
println()
@show uconvert(u"MPa", 1u"bar")
;
```

```
1.0m + 1.0cm = 1.01 m

1.0 * u"MPa" + 2.0 * u"bar" + 3.0 *
u"daN/cm^2" = 1.5e6 kg m^-1 s^-2

uconvert(u"MPa", 1 * u"bar") = 1//10 MPa
```

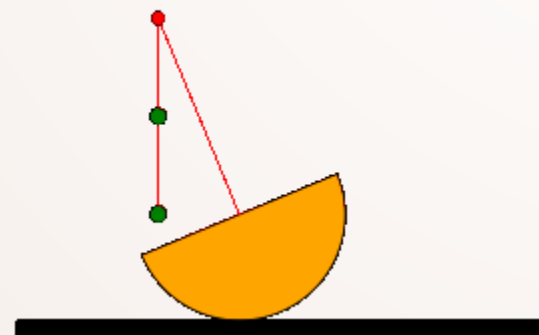
- Equations différentielles

```
using ModelingToolkit, Symbolics, DifferentialEquations, Plots
@parameters t
@_∂(x) = y -> expand_derivatives(Differential(x)(y))
d_dt = ∂_∂(t)
q = @variables θ(t) φ(t) ψ(t)
θ; φ; ψ = q = d_dt.(q)
q'' = d_dt.(q)
@parameters g R M m1 m2 l1 l2 L ;
```

... éq. d'Euler-Lagrange $\frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{q}_i} \right) - \frac{\partial \mathcal{L}}{\partial q_i} = 0$

```
K = J^I * θ^2 / 2 + m1 * (ẋ1^2 + ẏ1^2) / 2 + m2 * (ẋ2^2 + ẏ2^2) /
V = -M * g * yG * cos(θ) + m1 * g * y1 + m2 * g * y2
ℒ = K - V
EL = [d_dt(∂_∂(v)(ℒ)) - ∂_∂(v)(ℒ) for (v; v) ∈ zip(q; q)] ;
```

...



- Calcul tensoriel

```
using SymPy, LinearAlgebra, TensND
Spherical = coorsys_spherical()
θ, φ, r = getcoords(Spherical)
e^θ, e^φ, e^r = unitvec(Spherical)
@set_coorsys Spherical
ℐ, ℐ, ℐ = ISO() ; 1 = tensId2()
k, μ = symbols("k μ", positive = true)
λ = k - 2μ/3
u = SymFunction("u", real = true)
ε = SYMGRAD(u(r) * e^r)
ε |> intrinsic
```

```
(u(r)/r)e^θ⊗e^θ + (u(r)/r)e^φ⊗e^φ + (Derivative(u(r),
r))e^r⊗e^r
```

```
σ = λ * tr(ε) * 1 + 2μ * ε
eq = DIV(σ) · e^r
sol = dsolve(eq, u(r))
```

$$u(r) = \frac{C_1}{r^2} + C_2 r$$

```
T^ = tfactor(tsimpify(tsubs(e^r · σ · e^r, u(r) => sol.rhs(
```

$$\frac{-4C_1\mu+3C_2kr^3}{r^3}$$

Références pour auto-formation

- La documentation officielle → <https://docs.julialang.org/en/v1/>
- *Learn X in Y minutes* → <https://learnxinyminutes.com/docs/julia>
- *Intro to Julia tutorial (version 1.0) by Jane Herriman* → <https://youtu.be/8h8rQyEpiZA>
- *The Unreasonable Effectiveness of Multiple Dispatch* → <https://www.youtube.com/watch?v=kc9HwsxE1OY>