Une petite présentation de Julia

Adrien Brochier

9 November 2020

Présentation de Julia

- Julia est un langage de programmation open source relativement récent (développement démarré en 2009, première version en 2012)
- Utilisé de façon interactive, soit depuis la console, soit via une interface graphique (Jupyter, Juno, VSCode, ...)
- Langage de haut niveau (comparable à Python, R,..) avec des performances comparables à celles de C
- Pensé pour le calcul scientifique, plutôt pour de l'analyse numérique, mais il se prête particulièrement bien au calcul formel
- Principales caractéristiques:
  - code compilé à la volée
  - support natif et simple du parallélisme
  - gestion très fine des types, à la fois statique et dynamique
- Paraphrasé depuis https://marksaroufim.substack.com/

L'une des principales raisons pour lesquelles Julia est rapide est sa communauté: si vous déclarez publiquement sur internet que Julia est lent, des membres de la communauté réécriront la totalité de votre code juste pour prouver que vous avez tort.

- Sens de la citation: Julia est rapide à condition de faire un peu attention à la façon dont on écrit le code.
- Syntaxe de base assez similaire à Python et qui d'une façon générale permets d'écrire du code très compact
- Support natif des nombres complexes, grands entiers, grand rationnels
- Algèbre linéaire, manipulation de listes/vecteurs et vecteurs creux/tableaux, structures de données..
- Gestionnaire de paquets intégrés, plein de paquets disponibles y compris en maths
- Paquets "individuels" qui implémentent des outils ou structures basiques, mais aussi Nemo, qui aspire à devenir un paquet de calcul formel complet, et Oscar qui fournit une interface avec GAP, Singular et autres.
- Possibilité de faire de la "méta-programmation" (générer du code à la volée) et système de macro
- Macros de base pour, par exemple, débogage, profilage et parallélisme.

- Manipulation de vecteur, créations de fonctions à la volée...

```
julia> a=rand(-10:10,10);
[-1, -2, 10, 8, 9, 1, -2, 3, 9, 5]
julia> [x^2 for x in a if x > 5]
[100, 64, 81, 81]
julia> filter(x-> 0< x < 5,a)
[1, 3]
```

- Vectorisation

```
julia> f=x->x+2
#15 (generic function with 1 method)
julia> f.(a)
[1, 0, 12, 10, 11, 3, 0, 5, 11, 7]
```

- Parallélisme

```
1 Threads.@threads for i in truc
2     #du code
3 end
```

- Le gros point fort de Julia c'est sa gestion des types.
- On peut définir des fonctions en ne spécifiant pas de type, en spécifiant un type concret ou un type abstrait

```
function f(x)
    #type non spécifié/typage dynamique
end
function f(x::Int, y::Integer,z::Number)
    #Int est un type concret, Integer est un type abstrait
    ↪  qui inclut toutes les tailles d'entiers, Number un
    ↪  type abstrait qui inclut aussi Float, Complex, ...
end
function f(x::Union{Int,Rational})
    #On peut aussi prendre des unions de types
end
```

- Julia supporte aussi le "dispatch multiple" qui permet d'avoir des fonctions avec un même nom, mais un nombre/des types de paramètres différents
- C'est très souvent pratique, en particulier pour redéfinir des fonctions de base pour des types personnalisés (j'y reviendrai)
- On peut aussi stocker un type dans une variable, et/ou les récupérer directement via la déclaration d'une fonction

```
1 function f(x::T, y::T,z::S) where {T,S<:Number}
2    #T et S sont des types indéterminés, mais on impose que x
     ↪  et y ont le même et que S doit être un type de
     ↪  nombre. On peut ensuite utiliser T et S dans le corps
     ↪  de la fonction.
3 end
```

- Dans tous ces cas, la première fois qu'on utilise une fonction Julia en compile une version "statique" pour le type des arguments utilisés.
- C'est ce qui lui permet d'être à la fois générique et rapide.
- En revanche il faut que le compilateur soit capable d'inférer le type renvoyé par la fonction. Exemple idiot:

```
1 function pasBien(x)
2    if x>0
3       return x
4    else
5       return 0 #Si x n'est pas un entier le type renvoyé est
        ↪  instable
6    end
7 end
```

- Deux façon correctes de procéder:

```
1  function bien(x)
2     if x>0
3        return x
4     else
5        return zero(x) #Renvoit le zéro du type de x
6     end
7  end
8  function bien2(x::T) where {T}
9     if x>0
10       return x
11    else
12       return T(0)
13    end
14 end
```

- Julia supporte aussi des types à paramètres.
- Ce paramètre est souvent (mais pas forcément) un autre type
- Par exemple les types Complex, Rational ont en paramètre le type de leurs coefficients, les tableaux ont en paramètre le type de leur contenu.

```
julia> typeof([1,2,3])
Array{Int64,1}
julia> a=BigInt(2) #crée un entier en précision arbitraire
2
julia> b= (a+im*3)//5 # // indique qu'on veut faire du
↪  calcul exact
2//5 + 3//5*im
julia> typeof(b)
Complex{Rational{BigInt}}
```

- Enfin et surtout, on peut créer des types personnalisés, par exemple:

```
struct Mod{p} <: Number
   val::Int
end
```

- On peut ensuite redéfinir les opérations de base

```
Base.:+(x::Mod{p}, y::Mod{p}) where {p} =
↪   Mod{p}(Int(x.val)+y.val)
```

- Faire en sorte que les opérations entre un entier et un entier mod p soient automatiquement bien définies en expliquant comment convertir l'un en l'autre

```
Base.promote(x::Mod{p}, y::Integer) where {p}=(x,Mod{p}(y))
```

- Etc...

- Le fait que le modulo soit un paramètre du type rends pas mal de choses plus simples et plus transparentes.
- Toutes les fonctions pour lesquelles ça a un sens fonctionneront nativement avec ce nouveaux type.
- Sans rien faire de plus, le type Complex me permets de travailler sur $\mathbb{Z}/p\mathbb{Z}[i]$, le paquet "Polynomials" de travailler sur $\mathbb{Z}/p\mathbb{Z}[X]$, etc..
- Tout ceci de façon aussi rapide qu'avec les types de base.

Un exemple de calcul

- Conjecture de Deligne–Drinfeld: relation entre quatre algèbres de Lie
  - L'algèbre de Lie $\mathfrak{l}$ du "groupe de Galois motivique": c'est une algèbre de Lie graduée libre avec un générateur en chaque degré impair ≥ 3 (Deligne, Goncharov).
  - L'algèbre de Lie $\mathfrak{grt}$ du groupe de Grothendieck-Teichmüller, qui apparait comme groupe de symétries des associateurs de Drinfeld, des objets fondamentaux en topologie et en quantification par déformation.
  - L'algèbre de Lie $\mathfrak{dmr}$ du groupe des "double mélanges régularisés" qui encode les relations entre les fonctions multi-zeta (Racinet):

$$\zeta(s_1, \ldots, s_k) = \sum_{n_1 > n_2 > \cdots > n_k > 0} \frac{1}{n_1^{s_1} \cdots n_k^{s_k}}$$

  - L'algèbre de Lie $\mathfrak{krv}$ du problème de Kashiwara–Vergne, une conjecture ancienne démontré récemment sur la série de Baker–Campbell–Hausdorff et l'isomorphisme de Duflo en théorie de Lie.

- Les associateurs sont par définition des objets de la forme $\exp(\phi(a, b))$ ou ϕ est une série formelle de Lie en deux générateurs, qui satisfont des équations reliés à la géométrie des espaces de modules des surfaces de Riemann de genre 0 avec n points marqués.
- $\mathfrak{l}$ agit naturellement sur l'algèbre de Lie libre par dérivation, et Drinfeld montre que cela induit un morphisme

$$\mathfrak{l} \longrightarrow \mathfrak{grt}$$

- La conjecture de Deligne-Drinfeld affirme que c'est un isomorphisme.
- Brown a démontré que ce morphisme est injectif (c'est en gros une version motivique du théorème de Beyli qui dit que le groupe de Galois absolu des rationnels agit fidèlement sur le complété profini de F_2, càd le groupe fondamental étale de $\mathbb{P}_1(\mathbb{C})$ moins trois points)

- Il existe un associateur particulier, construit de façon analytique, qui est en quelque sorte une série génératrice pour les fonctions multi-zeta. On obtient de cette façon un morphisme $\mathfrak{grt} \to \mathfrak{dmr}$. Furusho montre que ce morphisme est injectif.
- Alekseev-Torrossian ont résolu la conjecture de Kashiwara-Vergne en utilisant la théorie des associateurs. En particulier, ils obtiennent une inclusion $\mathfrak{grt} \subset \mathfrak{krv}$.
- Un théorème de Schneps implique que cette inclusion se factorise en

$$\mathfrak{grt} \subset \mathfrak{dmr} \subset \mathfrak{krv}.$$

- On conjecture que ces inclusions sont également des isomorphismes.

- Un point crucial est que toutes ces algèbres de Lie sont graduées, et que ces inclusions sont compatibles avec le degré.
- $\mathfrak{krv}$ est la "plus grosse", et en un sens la plus simple à définir, parmi ces trois. Il est donc tentant de s'amuser à calculer sa dimension degré par degré, et de comparer avec les dimensions de $\mathfrak{l}$ qui sont bien sûr facile à calculer. En particulier, si les dimensions sont les mêmes, c'est à fortiori vrai aussi pour $\mathfrak{grt}$ et $\mathfrak{dmr}$.

- On note $\mathfrak{f}_2$ la complétion de l'algèbre de Lie libre en deux générateurs a, b, c'est-à-dire l'algèbre des séries formelles de Lie en ces deux variables.
- On note A_2 l'algèbre associative des séries formelles en a, b. On rappelle que $A_2 = U(\mathfrak{f}_2)$.
- Une dérivation u de $\mathfrak{f}_2$ est dite *tangentielle* si il existe $F, G \in \mathfrak{f}_2$ telles que

$$u(a) = [a, F] \qquad\qquad u(b) = [b, G].$$

- On dit que u est *spéciale* si on a en plus

$$u(a + b) = 0.$$

- On vérifie facilement que tout élément X de A_2 s'écrit de façon unique

$$X = a_0 + w_a a + w_b b.$$

où $w_a, w_b \in A_2$. On pose alors

$$\partial_a(X) := w_a \qquad\qquad \partial_b(X) := w_b.$$

- Ce sont des dérivations de A_2 (des analogues non-commutatifs des dérivées partielles).

- Soit cyc_2 l'espace vectoriel quotient de A_2 par le sous espace vectoriel (pas l'idéal !) $[A_2, A_2]$.
- Cet espace s'identifie aux combinaisons linéaires formelles de classes d'équivalence de permutations cycliques de mots en a, b. On note Tr_2 la projection évidente

$$A_2 \longrightarrow \mathrm{cyc}_2 \,.$$

- Finalement, soit $D = (F, G)$ une dérivation tangentielle, on pose

$$\mathrm{div}_2(D) = \mathrm{Tr}(a\partial_a F + b\partial_b G).$$

Théorème (Alekseev–Torrossian)

L'ensemble des dérivations spéciales u telles que $\mathrm{div}_2(u) = 0$ *est une sous algèbre de Lie de l'algèbre de Lie des dérivations de* $\mathfrak{f}_2$. *On la note* $\mathfrak{krv}^0$.

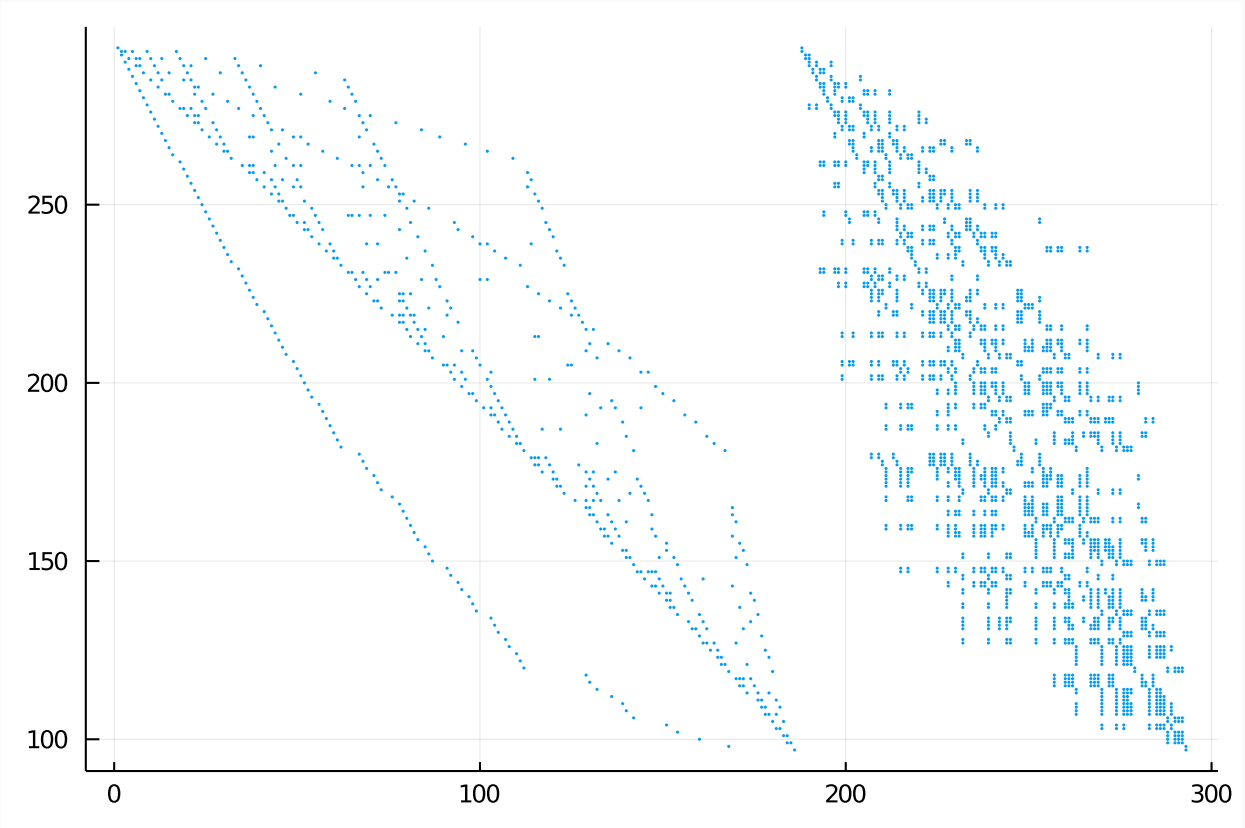
- La "vraie" algèbre de Lie $\mathfrak{krv}$ est une extension de $\mathfrak{krv}^0$, mais on sait que la dimension de $\mathfrak{krv}/\mathfrak{krv}^0$ est 0 en degré pair et 1 en dimension pair. Donc il suffit de calculer les dimensions graduées de $\mathfrak{krv}^0$.
- En degré n il faut donc
  - produire une base en degré n et $n+1$ de $\mathfrak{f}_2$ et en degré n de cyc_2
  - écrire l'application linéaire

$$\begin{aligned} \mathfrak{f}_2[n] \oplus \mathfrak{f}_2[n] &\longrightarrow \mathfrak{f}_2[n+1] \oplus \mathrm{cyc}_2[n] \\ (F, G) &\longmapsto ([a, F] + [b, G], \mathrm{div}_2(F, G)) \end{aligned}$$

  comme une matrice dans cette base et calculer son rang.
- Astuce: l'inclusion $\mathfrak{l} \subset \mathfrak{krv}$ donne une minoration de sa dimension, il suffit donc de prouver que le rang attendu minore le rang de cette matrice, ce qu'on peut faire en travaillant modulo un petit premier p.

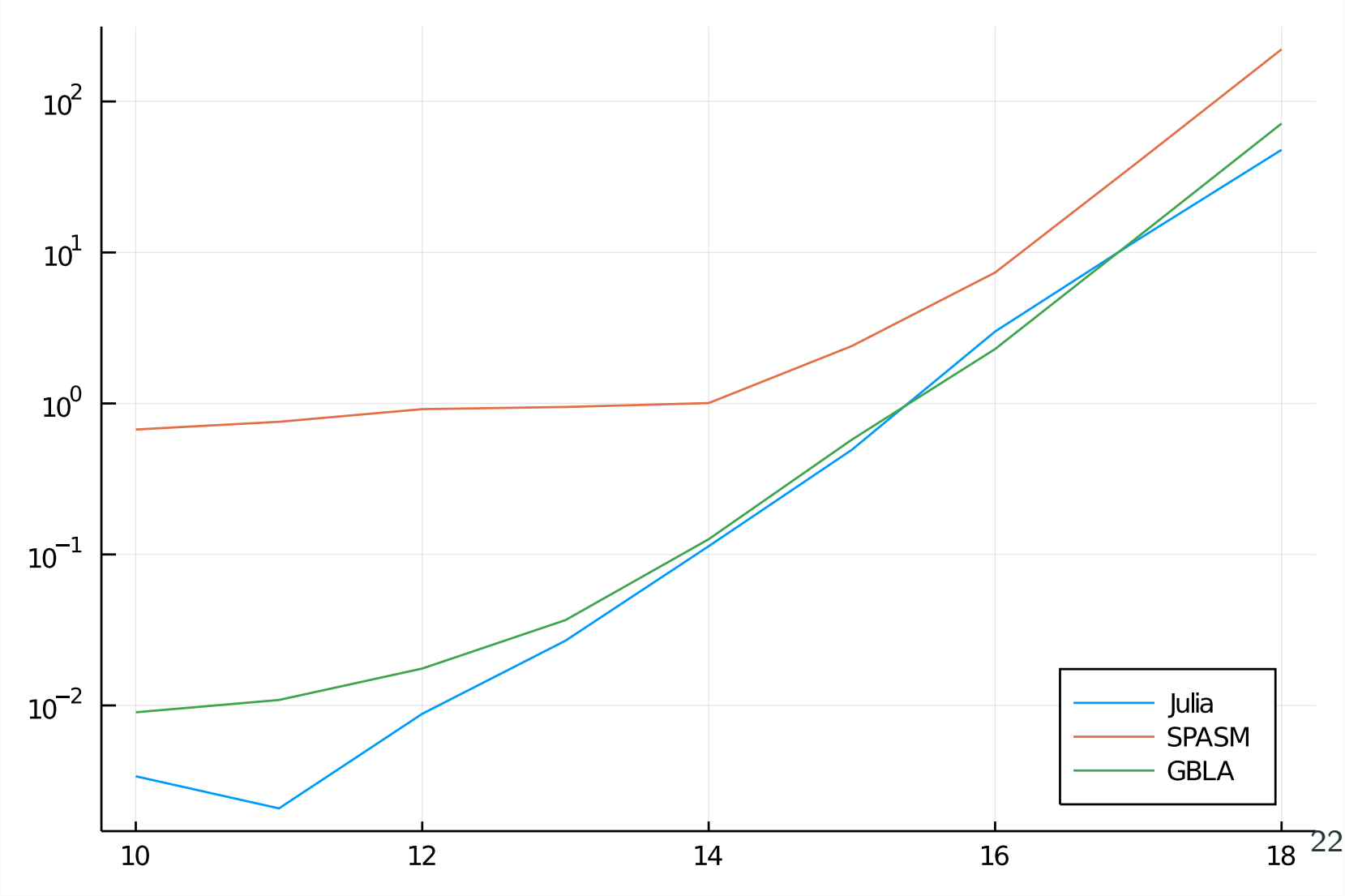
- À titre de comparaison forcément injuste: Albert–Harinck–Torossian ont fait ce calcul en 2008 avec une combinaison de Maple, Matlab et CAML. Il leur a fallu "18h sur un ordinateur puissant" pour produire la matrice en degré 16.
- En 2020, avec Julia sur mon ordinateur personnel, le calcul en degré 16 prend une quinzaine de secondes.
- On arrive sans trop forcer jusqu'au degré 23 (au delà on atteint les limites de la mémoire des machines de calculs à disposition à l'IMJ-PRG).

- Julia permets relativement facilement d'implémenter l'algèbre de Lie et l'algèbre associative libre en n générateurs dont les coefficients sont de types quelconque (en particulier on peut travailler $\mod p$ dès le départ).
- On obtient une matrice qui ressemble à ça:



- C'est donc une matrice très creuse, presque triangulaire, mod p, qui ressemble un peu aux matrices de base de Gröbner. Faugère–Lachartre ont développé un algorithme pour calculer le rang de ce genre de matrices qui est implémenté dans un programme en C (GBLA).
- Bouillaguet–Delaplace en ont une version beaucoup plus sophistiquée (plus économe en mémoire, mais un peu plus couteuse en calcul) implémenté dans un logiciel en C également (SPASM).

- Écrire l'algorithme de FL en Julia ne prend que quelques lignes de code (contre plusieurs centaines en C) pour des performances similaires (mais j'ai dû utiliser SPASM en degré 22 et 23 pour des questions de mémoire).



Merci de votre attention !