Comparaison des performances entre Python, MATLAB et Julia

Raphaël Sala
Encadré par
Fabrice Deluzet

5 août 2023

Table des matières

1 Introduction **3**

2 Langages **3**
- 2.1 Python . . . 3
- 2.2 MATLAB . . . 3
- 2.3 Julia . . . 4

3 Processus d'exécution **4**
- 3.1 AOT (Ahead-Of-Time) : compilation anticipée . . . 4
- 3.2 Interpréteur . . . 4
- 3.3 JIT (Just-In-Time) : compilation à la volée . . . 5
- 3.4 Exemple . . . 5

4 Optimisations **6**
- 4.1 Python . . . 6
  - 4.1.1 Numpy . . . 6
  - 4.1.2 Numba . . . 6
  - 4.1.3 Cython . . . 7
- 4.2 MATLAB . . . 7
- 4.3 Julia . . . 7

5 Méthodologie **9**
- 5.1 Codes . . . 9
- 5.2 Mesures des performances . . . 9

6 Résultats **10**
- 6.1 Calculs basiques (addition, multiplication, etc.) . . . 10
- 6.2 Algorithmes répétitifs . . . 11
  - 6.2.1 Algorithme non-vectorisable : Descente de gradient . . . 11
  - 6.2.2 Algorithme vectorisable : Descente de gradient . . . 11
  - 6.2.3 Algorithme récursif sans calcul complexe : Tri fusion . . . 12
  - 6.2.4 Comparaisons des résultats . . . 14
- 6.3 Manipulation de grandes données . . . 15
  - 6.3.1 Assemblage de matrices creuses . . . 15
  - 6.3.2 Equation différentielle ordinaire (EDO) . . . 15
  - 6.3.3 Résultats . . . 17
  - 6.3.4 Suivis des optimisations . . . 18

7 Conclusion **19**
- 7.1 Synthèse des résultats . . . 19
- 7.2 Résumé des optimisations clefs . . . 20

1 Introduction

La recherche scientifique fait de plus en plus appel à des langages de programmation de haut niveau, offrant une facilité d'utilisation et une grande flexibilité. Parmi ces langages, Python et MATLAB se sont rapidement imposés comme des outils populaires dans la communauté scientifique, tandis que Julia, un langage plus récent, cherche à se faire une place. Toutefois, il ne suffit pas d'être 'simple'. Les performances obtenues lors de l'exécution de programmes sont un facteur crucial pour de nombreuses applications scientifiques telles que la modélisation, la simulation, l'analyse de données, ...

L'optimisation des performances devient ainsi un enjeu majeur dans le domaine scientifique, où chaque seconde compte pour obtenir des résultats précis et efficaces. Il est donc essentiel de comparer les performances de ces langages de programmation afin de déterminer celui qui convient le mieux aux besoins spécifiques de chaque application.

2 Langages

2.1 Python

Python est un langage interprété et orienté objet. Créé en 1991 et depuis en open source , celui-ci a su créer une forte communauté et a pu, même si ce n'était pas son objectif initial, de se faire une place dans le domaine scientifique.

Au cours de ce rapport nous utiliserons Python 3.11.0. Ce langage typé dynamiquement utilise le *reference counting* et *tracing garbage collection* pour la gestion de la mémoire. Ainsi, des conflits peuvent survenir lorsques plusieurs threads essaient de modifier la mémoire en même temps. C'est pourquoi, un mécanisme est intégré pour y pallier : le GIL (Global Interpreter Lock) ; permettant de limiter l'exécution du programme à un unique thread. Toutefois, malgré des optimisations faites, notamment à partir de Python 3.2 [Pit09] , des techniques, telles que la parallélisation, sont encore restreinte par le GIL et donc coûteuse en temps malgré la capacité disponible des processeurs actuels. C'est pourquoi, des librairies externes seront nécessaires pour obtenir des performances satisfaisantes, dont nous parlerons plus tard.

2.2 MATLAB

MATLAB est un langage de programmation de haut niveau et un environnement de calcul numérique. Il a été créé en 1984 et est distribué en tant que produit commercial nécessitant une licence pour son utilisation, ce qui limite sa portabilité.

Au cours de ce rapport, nous utiliserons MATLAB R2023a. Il alloue dynamiquement la mémoire, bien que les détails internes de son fonctionnement ne soient pas explicitement documentés. Enfin, depuis la version 2015a, MATLAB introduit la compilation JIT (Just-In-Time) [ML20]. Il a été concu pour être une "extension" au code Fortran, un langage compilé. Ainsi, il est possible d'utiliser

des fonctions écrites en Fortran dans MATLAB, dans des fichiers appelés MEX. Ceux-ci permettent d'obtenir de meilleures performances. De plus, son architecture a été pensée pour les opérations sur les matrices, ce qui en fait un outil puissant pour les manipulations matricielles et les calculs numériques. [ML20]

2.3 Julia

Julia est un langage orienté objet distribué en libre accès. Récent dans le domaine scientifique, il a été créé en 2012 avec l'objectif de combler les lacunes des langages scientifiques déjà existants.

Au cours de ce rapport, nous utiliserons Julia 1.9.1, un langage à typage dynamique qui met en œuvre la *Generation garbage collection* pour la gestion de la mémoire. De plus, il s'exécute via un compilateur Just-In-Time (JIT), tirant pleinement parti des capacités de LLVM. Un ensemble de compilateurs permettant le passage du code source en code machine optimisé. L'une des forces de Julia réside dans son accès bas niveau à la mémoire, ce qui permet de développer des librairies performantes avec des surcoûts réduits liés à l'allocation et à la gestion de la mémoire Cette caractéristique rend Julia adapté aux calculs scientifiques et aux manipulations de données intensives. Enfin, cela permet une intégration efficace à d'autres langages tels que Fortran, C, Python, ...[Bez+17]

3 Processus d'exécution

3.1 AOT (Ahead-Of-Time) : compilation anticipée

Lors de l'exécution d'un programme, son code source doit être traduit en code machine, pour pouvoir être lu par l'ordinateur. Ainsi, pour les langages dit compilés, lors de la compilation, le code est minutieusement parsé de sorte à détécter les erreurs (syntaxe, sémantique, typage, ...) et à générer un code machine profondément optimisé. Bien que l'argument de la performance est à faire valoir, de nombreux désavantages sont à noter. Les langages sont bien moins dynamiques, voire même statiques. Ceci, limite la flexibilité du code et augmente également la complexité du développement. Ceci entraîne une perte de temps et nécessite des connaissances supplémentaires. Aussi, il convient de prendre en compte le temps de la compilation ainsi que le débogage qui peut s'avérer relativement plus difficile.

3.2 Interpréteur

Ainsi, une méthode adopté notamment par Python, est de plutôt interpréter le programme. C'est-à-dire traduire ligne par ligne le code source en code machine. Cette méthode permet de détecter les erreurs plus facilement. En effet, les modifications de code peuvent être rapidement testées et implémentées sans avoir à recompiler. Toutefois, la partie optimisation ne peut pas être faite ! Le résultat est très lent. C'est pourquoi de nos jours, les interpréteurs traduisent plutôt en un objet code, code intermédiaire (IR). C'est un langage ayant pour

objectif de ne perdre aucune information du code source tout en le synthétisant. Dans le cas de Python, c'est l'interpréteur CPython qui est utilisé et attribue des tokens aux variables, expressions, déclarations, ...Ces tokens permettent de construire un AST (Abstract Syntax Tree). Cette structure de données permet de faire bien plus d'optimisations que précedemment. Enfin, cela est traduit en bytecode, qui permet l'exécution du programme initial.

3.3 JIT (Just-In-Time) : compilation à la volée

D'ailleurs, un processus hybride est possible. Lors de l'exécution, le code source est échantillonné de sorte à détecter les zones critiques, *hotspots* lors du *warm-up*. Elles sont appelées "hots". Ces zones, qui sont un ensemble de lignes de codes, sont compilées en code machine. Ainsi, elles reçoivent l'optimisation qui leur est due. Cependant cette compilation est faite en amont lors de la première exécution. Puis, le reste du code est traité comme s'il était interprété. Cela permet de conserver la flexibilité du code interprété tout en ayant, quasiment, les performances d'un code compilé. Il est cependant important de noter que Julia et MATLAB utilisent des approches différentes en matière de compilation JIT. Dans le cas de MATLAB, quelques détails sont fournis mais sans trop de détails car le JIT est en constante évolution et nous ne sommes pas censé "tordre" notre code pour en bénéficier au maximum [Shu07].Enfin, Julia est plus transparent. La méthode principale est la dévirtualisation. C'est-à-dire remplacer les appels dynamiques par des appels directs lorsqu'il est possible de typer les arguments [Pel+21].

3.4 Exemple

Afin de mieux comprendre ces fonctionnements, nous allons nous servir d'extensions de Python permettant de voir l'aspect des différents processus d'exécution. Nous utiliserons Numba qui utilise un JIT et Cython, un AOT. Nous en parlerons plus en détail dans la prochaine section.

```
def mainInterp():
  a=0
  n=1000000
  for i in range(n):
    a+=1
  return 0
```

```
@jit(nopython=True)
def mainJIT():
  a = 0
  n=1000000
  for i in range(n):
    a += 1
  return 0
```

```
cdef int mainAOT():
  cdef int a = 0
  cdef int n = 1000000
  cdef int i
  for i in range(n):
    a += 1
  return 0
```

%timeit mainInterp()	%timeit mainJIT()	%timeit mainAOT()
49.6 ms	78.7 ns	39.9 ns

FIGURE 1 – Comparaison des performances d'un code Python selon différents modes d'exécution : Interpréteur, JIT et AOT

On remarque clairement le désavantage de CPython (l'interpréteur de base de Python) qui exécute ligne par ligne sans pour autant essayer une quelconque optimisation non locale contrairement au JIT de Numba ou l'AOT de Cython, qui lui en plus, grâce au typage statique diminue, par 2 son temps d'exécution par rapport à Numba.

4 Optimisations

4.1 Python

4.1.1 Numpy

Nous venons de remarquer que CPython n'est pas très performant. Toutefois, des libraires permettent de palier ce problème sans pour autant changer le mode d'exécution. C'est le cas de Numpy, une librairie de calcul scientifique qui permet de manipuler, à l'aide de la vectorisation, de larges tableaux de données. Ceci est possible dû au fait que les tableaux soient homogènes et stockés de manières contigüs [Numb]. Ainsi, cela permet l'optimisation des accès mémoires et d'utiliser des fonctions C et Fortran compilées, permettant du parallélisme (SIMD) via les libraires BLAS et LAPACK

Numpy est au coeur de nombreuses librairies de calcul scientifique, tel que Pandas, Scipy, Scikit-learn, ...C'est pour cela qu'au cours de ce rapport nous ne parlerons pas de PyPy, un JIT semblerait-il performant [Ben], puisque Numpy n'est pas compatible. Toutefois, Numpy traite uniquement les tableaux. C'est pour cela que des extensions, précedemment parlées, permettent d'optimiser le reste du code.

4.1.2 Numba

A l'instar de Julia, Numba recourt au compilateur LLVM pour convertir le code Python en code machine. Cependant, il utilise une version allégée, pour une intégration efficace entre Python et l'API LLVM quitte à délaisser certaines fonctionnalités [llv]. La conversion nécessite de spécifier *nopython=True*

Ensuite, pour optimiser le code, il est préférable d'écrire le code explicitement en utilisant notamment les boucles *for*. Si certaines parties sont parallélisables, et que le JIT ne les détecte pas, nous pouvons le spécifier via *parallel=True*. Toutefois, cette option désactive le GIL, une attention particulière est alors nécessaire pour prévoir d'indésirables erreurs [Numa]

De nombreuses autres options sont disponibles, en gardant, pratiquement, identique le code Python initial. Il suffit de découper notre code en fonctions pour ensuite décorer celles sujettent à optimisation. Toutefois, Numba permet une réelle optimisation seulement si tous les objets Python peuvent être convertis en code natif. Bien que Numpy soit compatible avec NumPy, pour bénéficier de ses fonctions et structures de données, il peut être nécessaire d'apporter des modifications significatives aux fonctions et aux codes existants pour obtenir une réelle optimisation.

4.1.3 Cython

Finalement, nous avons une dernière manière de procéder, Cython. Une extension permettant de compiler notre code Python en C. Cython tire parti de l'accès direct de fonctions, structures, etc ...écrits en C et au typage statique, bien que le typage dynamique soit toujours possible dû à l'inférence des types. Cette option permet d'avoir un code écrit en Python ainsi qu'en Cython, pour optimiser les goulots d'étranglements, *bottlenecks*. C'est pourquoi nous pouvons spécifier devant les fonctions au lieu de *def*, *cdef* ou *cpdef*. Qui décrivent l'utilisation de la fonction dans le code. Ces pratiques permettent, en le spécifiant, de passer outre le GIL lorsque nos code s'y prête [Cyt] ! Toutefois, Numpy peut utiliser des objets Python qui ne sont pas compatibles avec Cython. Alors, il faudrait utiliser des *Memoryviews*. Ensuite, pour indexer les tableaux, il est préférable de le faire via des variables typés Py_ssize_t

L'utilisation de Cython nécessite soit, pour un notebook, de méthodes magiques soit, pour un script, de créer un fichier *setup.py* pour compiler le code présent dans un fichier *.pyx*, limitant la portabilité que Python possède initialement.

4.2 MATLAB

Contrairement à l'idée des JIT où écrire un code implicite est conseillé en utilsant des boucles *for*, MATLAB indiquent clairement de faire l'inverse en utilisant la vectorisation [MATb]. Puis, comme les autres langages, il est conseillé d'éviter les variables globales et de favoriser la pré-allocation des tableaux [MATa]. Le stockage des tableaux est unidimensionnel. Ainsi, une manière de parser un tableau MATLAB est de le faire en colonne. En effet, comme Julia, MATLAB utilise le *column-major order*. C'est-à-dire que les colonnes d'un tableaux sont cotnigüs dans la mémoire, tandis que Python utilise le *row-major order*, soit les lignes.

```
% Row
n = 10000;
matrice = rand(n, n);
for i = 1:n
  for j = 1:n
    v = matrice(i, j);
  end
end
```

Row	Col
0.881 s	0.194 s

```
% Col
n = 10000;
matrice = rand(n, n);
for i = 1:n
  for j = 1:n
    v = matrice(j, i);
  end
end
```

FIGURE 2 – Comparaison des performances du parsage de tableaux en MATLAB

4.3 Julia

Comme dit précedemment, le JIT de Julia se concentre sur la dévirtualisation. Ainsi, le typage est une pratique principale aux langages statiques qui a

bon goût de faciliter la lecture du code, et qui contrairement à Python (PEP 484), permet surtout une optimisation fortement accrue. De cette manière, nous aimerions avoir des fonctions *stables*, c'est-à-dire que le type de la sortie soit prédicitible de par le type des entrées, et *enracinées* (groundedness) c'est-à-dire que le type des variables locales est constant [Pel+21].

```
function foo1(x)
    return (x>0) ? x : 0
end
function f1()
    sum=0
    for i in 1:100000
        sum += foo1(i/2)
    end
    return sum
end
```

```
@code_warntype f1()
```

```
Body::Union{Float64, Int64}
# Some results...
│%10=sum::Union{Float64, Int64}
│%11 = (i / 2)::Float64
│%12=Main.foo1(%11)::Union{Float64, Int64}
# Some results...
```

Nous constatons donc que le *body* n'est pas défini à cause des variables locales.

```
function foo2(x)
    return (x>0)?x: zero(x)
end
function f2()
    sum::Float64=0
    for i in 1:100000
        sum+=foo2(i/2)
    end
    return sum
end
```

```
@code_warntype f2()
```

```
Body::Float64
# Some results...
│%11 = sum::Float64
│%12 = (i / 2)::Float64
%13 = Main.foo2(%12)::Float64
# Some results...
```

@btime f_1	**@btime** f_2
211.000 μs	105.500 μs

FIGURE 3 – Mesures de performances de codes simples en Julia, non-typé et typé

Une diminution du temps par 2 en typant seulement une partie du code ce n'est pas anecdotique ! Ensuite, n'oublions pas que l'utilisation de la mémoire peut causer des goulots d'étranglements. Ainsi, vaut mieux favoriser la pré-allocation comme dit précedemment et l'utilisation de fonctions mutables ainsi que d'utiliser la syntaxe *dot* pour la vectorisation des opérations Enfin ce n'est qu'une liste non exhaustive pour les optimisations de Julia. La documentation officielle en fournit une bien plus complète [Jul]

5 Méthodologie

5.1 Codes

L'objectif principal de ce rapport est de comparer les performances de Python, Julia et MATLAB. Nous définissons ici que la performance est le temps d'exécution d'un code. Cependant, nous ne délaissons pas la lisibilité, la flexibilité et la mémoire allouée qui sont également des critères qui nous importent ! Ainsi, les codes qui seront présentés essaieront d'être le plus en adéquation avec ses critères. Cependant, nous ne prétendons pas que les codes présentés soient les meilleurs possibles. Même, ils ne le seront probablement pas car les algorithmes seront simples de sorte à pouvoir dégager la caractéristique étudiée facilement, à l'instar de *Julia Micro-Benchmarks*

L'entièreté des codes seront présents sur un GitHub dédié [Sal]. De plus, l'entièreté des programmes seront exécutés à l'aide d'un processeur Intel i5-7440HQ et de 16Go de RAM.

5.2 Mesures des performances

En Python, il existe des méthodes magiques, notamment *%timeit* et *%memit* qui permettent de mesurer, respectivement, le temps d'exécution et la mémoire allouée par l'appel d'une fonction. Pour *%timeit*, le code est exécuté à plusieurs reprises jusqu'à ce que le temps total atteigne 5 secondes, afin de calculer une moyenne. On en déduit donc que cette méthode est la plus adaptée pour les codes s'exécutant rapidement. Ensuite, pour les codes plus conséquent, nous utilisons un *line_profiler*. Cette outil décrit le nombre de fois que chaque ligne est exécutée et le temps associé. Nous devons spécifier les fonctions à profiler à l'aide du décorateur *@profile*.

MATLAB utilise les méthodes *tic* et *toc* pour mesurer le temps d'exécution d'un code. La plupart des codes seront exécutés un grand nombre de fois, entre 1 000 et 1 000 000, afin d'obtenir une moyenne plus précise en un temps raisonnable. Il existe également la méthode *timeit*. Cependant, elle est imprécise pour des fonctions s'exécutant rapidement. Puis à l'instar de Python, les codes plus conséquents seront analysés à l'aide du Profiler intégré à MATLAB.

En Julia, il existe une macro intégrée *@time*. Toutefois, nous ne l'utiliserons pas puisqu'elle prend également en compte le temps de compilation. De plus, pour des exécutions rapides, cet outil devient imprécis. C'est pourquoi nous allons utiliser une libraire externe *BenchmarkTools* dont un exemple a déjà été présenté par *@btime*. Le résultat renvoyé est la meilleure performance au cours d'une série d'exécution, en terme de temps et d'allocation mémoire. L'idée de choisir le minimum est de ne pas être influencé par les autres processus qui peuvent tourner en fond. Ainsi ce résultat se rapproche au mieux d'une exécution idéale. Toutefois, cette manière de faire n'est pas parfaite. Par exemple, si notre fonction dépend de valeurs aléatoires, alors notre test devient plus une statistique qu'une mesure de performance. C'est pourquoi la macro *@benchmark* permet d'obtenir bien plus d'informations. Quant au profiler de Julia, il est éga-

lement intégré à l'environnement *@profview*. Cependant, c'est un *Flame Graph* rendant plus difficile la lecture des performances.

6 Résultats

6.1 Calculs basiques (addition, multiplication, etc.)

Notre première comparaison est à propos d'opérations simples. Soient la somme, la multiplication et la logique. Nous présentons les algorithmes sous forme de pseudo-code.

```
function sum(a, b)
  return a + b
```

```
function prod(a, b)
  return a * b
```

```
function logic(a, b)
  si a>0.5 et b<0.5
    return a
  sinon, si a<0.25 ou b>0.5
    return b
  sinon
    return 0
```

FIGURE 4 – Pseudo-code des fonctions de *OperationsBasiques*

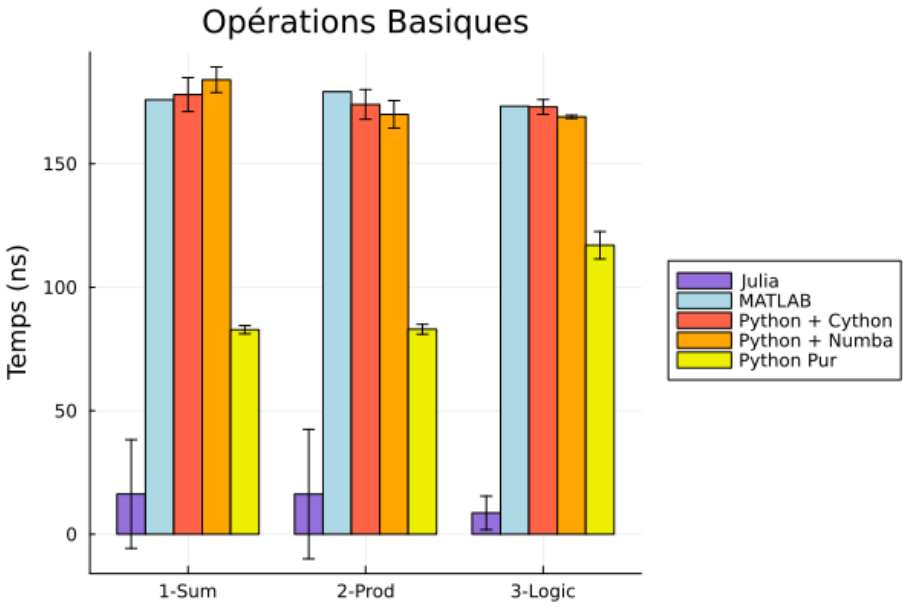

FIGURE 5 – Résultats des fonctions de *OperationsBasiques*

De nombreuses choses sont intéressantes à observer. Tout d'abord, les ver-

sions "optimisées" de Python sont presque 2 fois plus lentes que la version de base. Cela s'explique par le fait que les opérations de base sont déjà optimisées par le langage. Ainsi, en essayant de les optimiser de nouveau, nous ajoutons des opérations supplémentaires qui ralentissent l'exécution. Ensuite, nous remarquons que Julia est nettement plus rapide ! Cela provient principalement du *multiple dispatch*, multiméthode, qui permet de choisir lors de l'exécution la fonction la plus adaptée selon les types des arguments [Bez+18]. Contrairement à Python qui utilise le *duck typing*, typage canard, utilisant un *conditionnal dispatch*.

6.2 Algorithmes répétitifs

6.2.1 Algorithme non-vectorisable : Descente de gradient

Pour cet exemple, nous avons choisi la descente de gradient non vectorisable. Afin notamment de voir le comportement des langages sur des boucles *for*. Nous utilons un jeu de données de 100 points générés aléatoirement, sur un nombre d'itération de 1000 et un pas de 0.01.

```
function gradientDescentNonVec(x, y, alpha, num_iter)
  theta_1, theta_2 = 0, 0
  Grad_1, Grad_2 = 0, 0
  m = length(x) # Nombre de points
  for i in 1:num_iter
    for j in 1:m
      Grad_1 += theta_1 + theta_2 * x[j] - y[j]
      Grad_2 += (theta_1 + theta_2 * x[j] - y[j]) * x[j]

    theta_1 -= alpha/m * Grad_1
    theta_2 -= alpha/m * Grad_2
    Grad_1, Grad_2 = 0, 0

  return theta_1, theta_2
```

FIGURE 6 – Pseudo-code de la descente de gradient non vectorisable de *AlgorithmesRepetitifs*

6.2.2 Algorithme vectorisable : Descente de gradient

Ensuite, son équivalent vectorisable permettant de voir les différences des 2 approches.

```
function gradientDescentVec(x, y, alpha, n)
  theta_1, theta_2 = 0, 0
  X = [ones(length(x)) ; x]
  # La matrice X :
  # | Colonne 1 | Colonne 2 |
  # |     1     |    x[1]   |
  # |     1     |    x[2]   |
  # |     1     |    x[3]   |
  # |    ...    |    ...    |
  y = reshape(y, (length(y), 1)) # y en colonne

  for i in 1:n
    gradient = transpose(X) * (X * theta - y)
    theta -= alpha/m * gradient

  return theta_1, theta_2
```

FIGURE 7 – Pseudo-code de la descente de gradient vectorisable de *AlgorithmesRepetitifs*

6.2.3 Algorithme récursif sans calcul complexe : Tri fusion

Puis, un algorithme qui ne dépend pas d'opérations mathématiques et n'étant pas vectorisable mais néanmoins parallélisable.

```
# l(eft) et r(ight) sont les indices des sous-parties
# Aux, un tableau auxiliaire limitant l'allocation
function merge(A, Aux, l, r)
  if (r > l)
    m = (l+r) ÷ 2 # Milieu
    # Divise à gauche et à droite
    merge(Aux, A, l, m)
    merge(Aux, A, m + 1, r)

    i = l
    j = m + 1
    for k in l:r
      # Fusion des parties
      si i > m
        Aux[k] = A[j]
        j += 1
      sinon, si j > r
        Aux[k] = A[i]
        i += 1
      # Tri
      sinon, si A[j] < A[i]
        Aux[k] = A[j]
        j += 1
      sinon
        Aux[k] = A[i]
        i += 1

function mergeSort(A)
    Aux = copy(A)
    merge(Aux, A, 1, length(A))
```

FIGURE 8 – Pseudo-code du tri fusion de *AlgorithmesRepetitifs*

6.2.4 Comparaisons des résultats

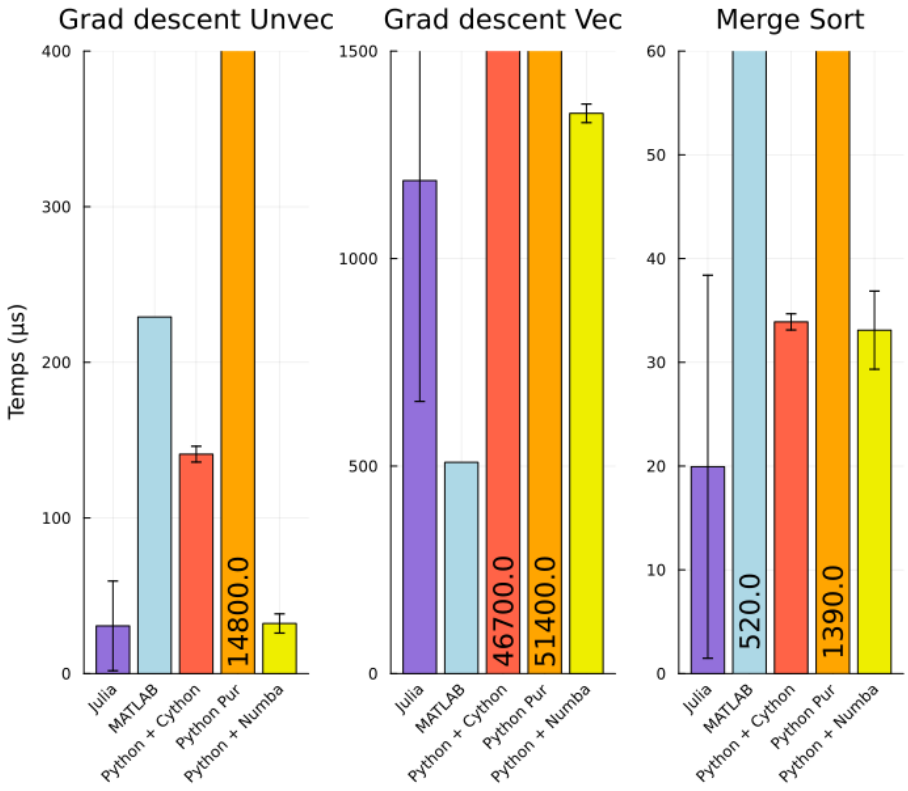

FIGURE 9 – Résultats des fonctions de *AlgorithmesRepetitifs*

Certaines valeurs ont été tronquées pour des raisons de lisibilité. Leur valeur sont tout de même annotées sur le graphique.

Nous constatons que la descente de gradient non vectorisée est plus rapide que sa version vectorisée. Cela est dû au fait que les codes vectorisées ont des boucles *for* implicites, que des variables temporaires sont allouées et désallouées et que les JIT ainsi que les compilateurs, peuvent avoir plus de mal à optimiser le code [Joh17]. Des exceptions existent, notamment lorsque le code vectorisé fait appel à des fonctions BLAS/LAPACK. En Julia, les codes non vectorisées sont donc conseillés. Des macros permettent même d'accélérer les boucles simplement. *@turbo* de la librairie *LoopVectorization* effectuant une optimisation de la vectorisation en choisissant l'ordre des boucles, pour minimiser le temps de calcul prévu. Puis, *@inbounds*, présente également pour Cython, sous le nom de *boundscheck(False)* permet d'éviter la vérification des indices des tableaux. Numba, en plus des optimisations de son JIT, cotoyant le temps d'exécution de

Julia, possède un argument *fastmath=True*. Celui-ci réarrange l'ordre de certaines opérations mathématiques et introduisant des techniques d'accélération de calculs pouvant, néanmoins, créer des erreurs d'arrondis [Numa].

Nous remarquons également l'avantage de MATLAB, par rapport aux autres langages, sur des codes vectorisés, étant spécialisé dans la vectorisation. Ensuite, Cython est au même stade que Python Pur [1]. Cela est dû au fait que Numpy est déjà vectorisé et que Cython perd son optimisaiton sur la conversion des types. Toutefois, en utilisant la fonction *cythonize()* nous pouvons obtenir un fichier *.c* qui une fois compilé est 10x plus rapide que Python. Cependant, ce ne sont pas des performances suffisantes pour pouvoir rivaliser, donc nous ne ferons qu'énoncer cette technique.

Finalement, sur un algorithme dénué de calculs et de vectorisations. Julia parvient à être considérablement plus rapide dû aux fonctions spécialisées et notamment mutables contrairement à MATLAB qui pêche lourdement sur ce point. En effet, les performances sont perdues lors des allocations.

6.3 Manipulation de grandes données

6.3.1 Assemblage de matrices creuses

Pour mieux évaluer la réelle ergonomie des langages, nous nous interessons à des algorithmes moins simplets que précedemment.

Le pseudo code est omis pour des raisons de lisibilité et de pertinence. L'objectif de l'alogrithme est de comparer la solution analytique et numérique : discrétisation d'une équation aux dérivées partielles ; en manipulant des matrices creuses.

6.3.2 Equation différentielle ordinaire (EDO)

Puis, la résolution d'EDO est un bon exemple pour évaluer la performance sur un algorithme plus complexe. Nous utilisons des fonctions spécialisées pour chaque langage, donc pour que les tests soient comparables les résolveurs seront similaires. *ode45* pour MATLAB, *DP5* pour Julia et *RK45* pour Python [Scia] [Scib].

1. Optimisation du code non concluante. En attente de retours en cas d'éventuelles erreurs

```
Paramètres : abstol, reltol, beta, nbPointsX, nbPointsY
function Sol(x, y):
    ... calculs ...
function champMagNormaliseRenverse(Y, p, t):
    ... calculs ...

function condition(u, t, integrator):
    if value == u[2]*(1-u[2])
        stop

function propagation(x, y)
    solveur = ode45 / DP5 / RK45
    intervalle = (0, 15)
    fun = ODEFunction(champMagNormaliseRenverse)
    prob = ODEProblem(fun, (x,y), condition, intervalle)
    return solve(prob, solveur , relTol, abstol)

function main()
    X = [0;1] avec NpointsX points
    Y = [0;1] avec NpointsY points
    # Parse tous les points de la grille hormis les bords
    for (i,j) in [1;nbPointsX-1] X [1;nbPointsY-1]
        x, y = X[i], Y[j]
        solTheorique[j, i] = Sol(x, y)

        foot1, foot2 = propagation(x, y)
        solPropagee[j, i] = Sol(foot1, foot)

    return |solTheorique - solPropagee| # Erreur absolue
```

FIGURE 10 – Pseudo-code d'une EDO de *LargesMatrices*

6.3.3 Résultats

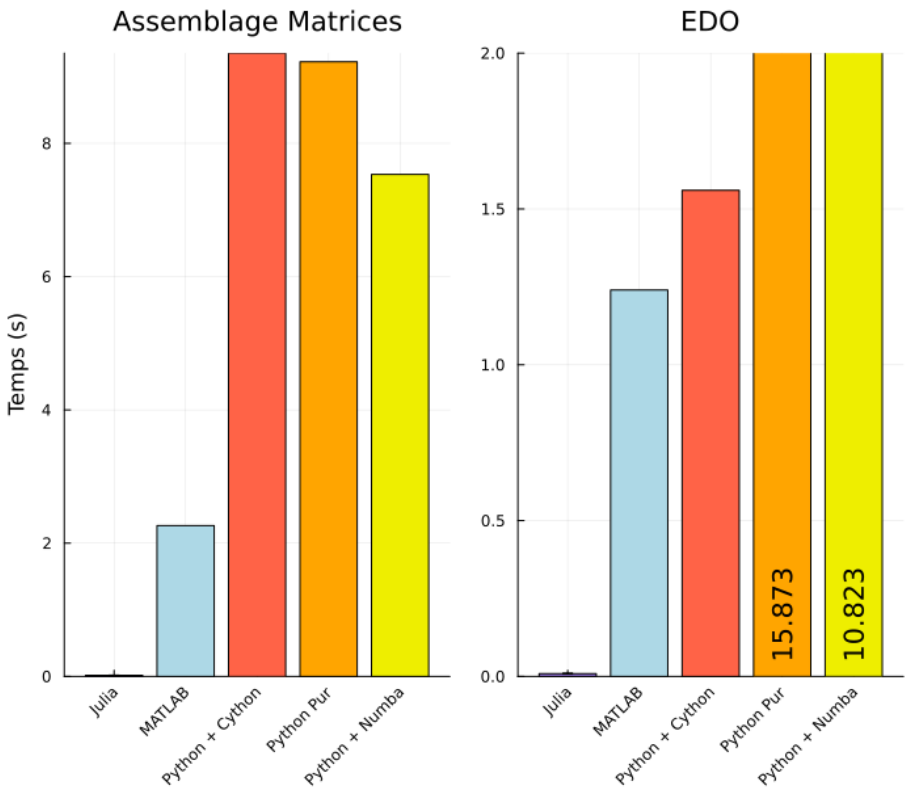

FIGURE 11 – Résultats des fonctions de *LargesMatrices*

Sur des exemples concrets nous remarquons les "vraies" performances des langages. En effet, ces algorithmes plus complexes permettent de mettre en avant leurs défauts et leurs qualités. Ainsi, nous pouvons voir que Julia est nettement plus rapide, d'un facteur allant de 100 à 1000. Cette rapidité est le résultat d'un design bien pensé, qui repose sur des fondements solides, permettant aux bibliothèques et aux structures complexes de Julia d'être fortement optimisées [Bez+18].

Puis, Cython et Numba ont obtenu des résultats médiocres. La compatibilité avec des bibliothèques externes, ainsi que de structures non intégrées peuvent poser des problèmes, empêchant ainsi l'optimisation d'une partie du code, voire son entièreté. Pour espérer ne serait-ce que de meilleurs résultats que Python leurs codes ont été plus ou moins tortueusement modifié. Ainsi, bien que les codes peuvent sûrement être encore améliorés, leur performance ne risque pas de changer radicalement. Le temps de développement et l'aisance d'écriture est relativement laborieuse pour ces résultats.

Enfin pour MATLAB, bien qu'il a été plus simple lors du codage grâce à ses fonctions spécifiques intégrées et sa syntaxe, les résultats sont tout de même décevants. Il semblerait que son temps d'exécution soit perdu lors des opérations d'allocations. La pré-allocation de la mémoire permet uniquement d'y pallier mais partiellement...

6.3.4 Suivis des optimisations

Au vu des résultats précédents, Julia est nettement meilleur en terme de performance. C'est ainsi, que le détail des optimisations sera fait à propos de ce langage afin de montrer la pertinence des optimisations conseillées lors de ce rapport.

Les codes sont dans *_EvolutionPerf* où un fichier texte liste les résultats, l'allocation mémoire et les paramètres choisis. La colonne *Temps* représente le temps moyen d'exécution, obtenu par *@benchmark*. Enfin, chaque code reprend les optimisations des lignes au-dessus de lui. Ainsi, la dernière ligne représente le temps d'exécution du code utilisé pour le graphique 11

Optimsations	**Temps**
BasicCode	1.814 s
Pré-allocation	78.446 ms
FonctionsMutables	21.071 ms
StaticArrays	17.500 ms
InboundsViews	16.827 ms

Optimsations	**Temps**
BasicCode	89.408 ms
Typed	41.247 ms
StaticArrays	9.081 ms

FIGURE 12 – Suivis des optimisations de *LargesMatrices*

Le premier code, *BasicCode*, est extrêmement lent puisque des opérations d'allocation sont continuellement effectuées. Puis, une fois réglé, de par les fonctions mutables, la mémoire utilisée a grandement chuté, de 10.10 GB à 16.64 MiB [2].

L'utilisation de structures adaptées permettent un meilleur accès aux données, ainsi que des opérations plus rapides. Toutefois, l'inférence de type ne peut toujours pas être réalisée, particulièrement pour les variables globales : indiqué par le résultat *Typed* du second tableau. Il est donc nécessaire de spécifier leur type afin d'obtenir de meilleurs résultats.

Ensuite, l'utilisation des tableaux de *StaticArrays* exploite de manière plus efficace la mémoire RAM de l'ordinateur. Elle est divisée en deux parties : la pile (stack) et le tas (heap). La pile offre un espace de stockage plus restreint mais avec un ordre de traitement rapide. Elle est utilisée entre autre pour les variables locales. Tandis que le tas permet l'allocation dynamique de mémoire, mais ses accès sont plus lents avec un risque de fragmentation. Avec les *StaticArrays*, la taille, et parfois les valeurs, sont connues à la compilation, ce qui permet d'effectuer des optimisations spécifiques, notamment pour son stockage dans la pile.

2. Lors de la dernière optimisation, nous sommes à 9.98 MiB !

Enfin, en utilisant de simples macros, ici *@inbounds*, et en retournant des *views* de tableaux au lieu de les copier, de légères améliorations sont possibles. L'efficacité de ces dernières optimisations augmente avec la taille des tableaux.

7 Conclusion

7.1 Synthèse des résultats

Sans surprise Python Pur est dernier de ce benchmark. Ce n'est pas étonnant puisque cela n'a jamais été son but initial. C'est un outil hautemant polyvalent, utile pour de petits tests où le temps est négligeable mais pour du calcul scientifique, il en devient inutilisable.

Tandis que ses extensions [3] essaient de combler ses lacunes. Toutefois, une expression provenant de la communauté résume bien leur utilisation : "Ces extensions sont de bonnes alternatives, jusqu'à ce qu'on les essaie réellement" En effet, dans le cas de Cython, certaines opérations ne sont tout simplement pas optimisables, les types de données doivent être spécifiés, parfois de manière accrobatique pour satisfaire la compatibilité entre les bibliothèques, Python et Cython. Si l'objectif est d'accélerer légèrement le code, l'option *Cythonize()* permet, comme dit précedemment, d'obtenir une version C de son code Python. Bien qu'elle ne soit pas modifiable, un gain de performance de l'ordre de 10 peut être espéré sans avoir à coder davantage.

Ensuite, Numba a donné sur ce benchmark de bons résultats en plus d'avoir un temps de développement relativement plaisant. Les codes testés sont sur de petites fonctions, un format qui s'y prête bien. Or, sur des projets plus conséquents, la liaison des diverses fonctions et l'utilisation d'objets/structures plus complexes exigent généralement un retour au *mode Python*, ralentissant considérablement l'optimisation. C'est pour cela que dans le cas de Python, je déconseille son utilisation mis à part peut-être Numba sur de petits codes.

En ce qui concerne, MATLAB bien qu'il soit complet pour le domaine scientifique, ses résultats sont devancés par Julia. Son environnement de développement et son code propriétaire limite son évolution. Une des ses froces est également que de nombreux outils complexes soient disponibles, mais il est probable que, d'ici peu de temps, ils soient implémentés ailleurs via d'autres langages, si ce n'est pas déjà fait. Malgré sa longévité, je trouve qu'aujourd'hui des outils permettent de se passer de MATLAB.

En effet, Julia en est un. Ce nouveau langage a fait ses preuves lors de ce benchmark. Lors de la rédaction de ce rapport, c'est celui pour lequel le temps de développement a été le plus court, avec de bons résultats de performances. Sa communauté est grandissante et de nombreuses améliorations sont continuellement apportées. Son écriture est similaire à celle de MATLAB et en moins contraignante. De plus, elle est simple et expressive à l'instar de Python. Récemment créé, Julia a su adopter une architecture permettant un entre deux parmi les langages bas niveaux et haut niveaux créés avant les années 2000.

3. Numba, Cython, PyPy, Pythran, ...

7.2 Résumé des optimisations clefs

A la suite de cette synthèse, nous constatons que les langages liant performance et dynamisme sont ceux disposant d'un mode d'exécution JIT. Conséquemment, une partie des optimisations mentionnées dans la liste ci-dessous, est dédiée à ces langages. Lors de l'écriture des codes, de nombreuses optimisations ont été testées et les plus importantes, enfin celles amenant simplement et rapidement des gains de performances, sont les suivantes :

- **Typer** les variables, afin de faciliter l'inférence de types. Notamment, les **variables globales**, qui, une fois typées, évitent des allocations répétées.
- Ecrire du code **non-vectorisé**. En général, cette pratique évite l'allocation excessive pour les variables temporaires et elle prend en charge la localité de la mémoire. C'est-à-dire, que les accès à la RAM sont limités pour plutôt utiliser efficacement les caches et registres du CPU. Puis, le code étant plus explicite, le compilateur peut plus facilement réorganiser, fusionner et donc optimiser les tableaux [Joh17].
- La **pré-allocation** et les **fonctions mutables** permettent, respectivement, d'éviter le coût de la réallocation dynamique et la création de copies de données, en modifiant directement l'argument passé.

Codes sources

[Sal] Sala2Code. *MicroBenchmark-Julia-MATLAB-Python*. URL : https://github.com/Sala2Code/MicroBenchmark-Julia-MATLAB-Python.

Articles

[Bez+17] Jeff Bezanson et al. "Julia : A Fresh Approach to Numerical Computing". In : *SIAM Review* 59.1 (2017). DOI : 10.1137/141000671. eprint : https://doi.org/10.1137/141000671. URL : https://doi.org/10.1137/141000671.

[Bez+18] Jeff Bezanson et al. "Julia : Dynamism and Performance Reconciled by Design". In : *Proc. ACM Program. Lang.* 2.OOPSLA (oct. 2018). DOI : 10.1145/3276490. URL : https://doi.org/10.1145/3276490.

[ML20] Cleve Moler et Jack Little. "A History of MATLAB". In : *Proc. ACM Program. Lang.* 4.HOPL (juin 2020). DOI : 10.1145/3386331. URL : https://doi.org/10.1145/3386331.

[Pel+21] Artem Pelenitsyn et al. "Type Stability in Julia : Avoiding Performance Pathologies in JIT Compilation". In : *Proc. ACM Program. Lang.* 5.OOPSLA (oct. 2021). DOI : 10.1145/3485527. URL : https://doi.org/10.1145/3485527.

Documentations

[Cyt] CYTHON. *Faster code via static typing*. URL : https://cython.readthedocs.io/en/latest/src/quickstart/cythonize.html.

[Jul] JULIA LANGUAGE. *Performance Tips*. URL : https://docs.julialang.org/en/v1/manual/performance-tips/.

[llv] LLVMLITE. *llvmlite*. URL : https://llvmlite.readthedocs.io/en/latest/.

[MATa] MATLAB. *Techniques to Improve Performance*. URL : https://fr.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html.

[MATb] MATLAB. *Vectorization*. URL : https://fr.mathworks.com/help/matlab/matlab_prog/vectorization.html.

[Numa] NUMBA. *Performance Tips*. URL : https://numba.readthedocs.io/en/stable/user/performance-tips.html.

[Numb] NUMPY. *Internal organization of NumPy arrays*. URL : https://numpy.org/doc/stable/dev/internals.html.

[Scia] SCIML. *ODE Solvers - DifferentialEquations.jl Documentation*. URL : https://docs.sciml.ai/DiffEqDocs/stable/solvers/ode_solve/.

[Scib] SCIPY. *scipy.integrate.RK45*. URL : https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.RK45.html.

Autres

[Shu07] Loren SHURE. *Keep your code natural. Don't write unnatural code just to take advantage of this*. Blog post. 2007. URL : https://blogs.mathworks.com/loren/2007/03/22/in-place-operations-on-data/.

[Pit09] Antoine PITROU. *[Python-Dev] Reworking the GIL*. http://mail.python.org/pipermail/python-dev/2009-October/093321.html. 2009.

[Joh17] Steven G. JOHNSON. *More Dots : Syntactic Loop Fusion in Julia*. 21 jan. 2017. URL : https://julialang.org/blog/2017/01/moredots/.

[Ben] PyPy BENCHMARK. *Speed Comparison : PyPy vs CPython*. URL : https://speed.pypy.org/.