Chapitre 3 : Modèle hybride linéarisé dans le cas $1dz - 3dv$

J'ai ajouter une commande pour des commentaires, qui sera simple à modifier pour que tout le texte apparaisse normalement (ou le supprimer complètement) au lieu d'enlever tous les `\textcolor{color}{text}` :

- **Ceci est un commentaire anonyme :** `\commentaire{Test}`
- Ceci est un commentaire rédigé par Anaïs : `\Anais{Test}`.
  C'est un alias de `\commentaire[Anais]{Test}`
- Ceci est un commentaire rédigé par Nicolas : `\Nicolas{Test}`.
  C'est un alias de `\commentaire[Nicolas]{Test}`
- Ceci est un commentaire rédigé par Josselin : `\Josselin{Test}`.
  C'est un alias de `\commentaire[Josselin]{Test}`
- Ceci est un commentaire rédigé par quelqu'un d'autre, son nom s'affiche alors en note en bas de page : `\commentaire[Bob]{Test}` [1]

Dans ce chapitre, l'introduction, la présentation du modèle, la section sur les schémas numériques ainsi que les premiers résultats numériques seront une reprise (plus ou moins étoffée) de l'article. Dans l'immédiat j'avais envie d'écrire des trucs un peu plus neufs, et savoir comment l'étoffer.

1 Introduction

L'objectif de ce chapitre est de mettre en place les différentes stratégies de résolution numérique présentées dans le chapitre 2 sur un modèle hybride linéairsé $1dz - 3dv$.

2 Présentation du modèle

$$\begin{cases} \partial_t j_{c,x} &= \Omega_{pe}^2 E_x - j_{c,y} B_0 \\ \partial_t j_{c,y} &= \Omega_{pe}^2 E_y + j_{c,x} B_0 \\ \partial_t B_x &= \partial_z E_y \\ \partial_t B_y &= -\partial_z E_x \\ \partial_t E_x &= -\partial_z B_y - j_{c,x} + \int v_x f_h \, \mathrm{d}v \\ \partial_t E_y &= \partial_z B_x - j_{c,y} + \int v_y f_h \, \mathrm{d}v \\ \partial_t f_h &= -v_z \partial_z f_h + (E_x + v_y B_0 - v_z B_y)\partial_{v_x} f_h + (E_y - v_x B_0 + v_z B_x)\partial_{v_y} f_h + (v_x B_y - v_y B_x)\partial_{v_z} f_h \end{cases}$$

1. Commentaire rédigé par Bob

3 Schémas numériques

3.1 Méthode de *splitting* haniltonien

3.2 Méthode de Lawson sur le modèle hybride

4 Génération automatique de code

Je propose de mettre ceci sous forme d'une section ici, mais je ne sais pas trop quoi y dire. Il est compliqué de développer plus sans mettre d'extrait de code Python, et je ne sais pas si cela est nécessaire ou non (rentrer plus dans les détails nécessite de parler un peu plus de l'implémentation de SymPy). Je présente ici la génération de code de manière globale, sans parler des problèmes de minimisation des expressions nécessaire dans le cas Padé, et je ne fais que lister les bibliothèques Python que j'utilise. Sachant que ces outils ont déjà été utilisé pour la partie sans approximation de e^{tL}, seulement pour de l'aide à l'écriture.

La simulation d'un système à 7 variables, 6 variables à une dimension, et 1 variable à 4 dimensions, avec une méthode de type Lawson-Runge-Kutta (LRK) d'ordre élevé, nécessite de nombreuses lignes de code dont l'écriture peut s'avérer fastidieuse. Une part importante de l'analyse ayant été réalisée à l'aide de la bibliothèque de calcul symbolique Python : SymPy, il a été décidé de poursuivre son utilisation pour aider à l'écriture du code de simulation. Dans un premier temps cet usage s'est limité à une aide à l'écriture en générant chacune des 7 expressions pour chaque variable, et ce à chaque étage de la méthode LRK (3 étages pour RK(3,3), jusqu'à 5 étages pour une méthode comme DP4(3)). Des outils de méta-programmation ont été utilisés pour obtenir une génération complète du code à partir d'un squelette de code et de l'écriture du schéma LRK que l'utilisateur souhaite utiliser.

Les expressions SymPy sont gérer comme des arbres syntaxiques dont les feuilles sont des nombres ou des symboles. Ces derniers vont servir à représenter des variables C++, il est donc nécessaire dans un premier temps de s'assurer que la conversion de ces symboles en chaînes de caractères assure des noms de variables valide en C++. En effet il est fréquent d'utiliser des symboles s'exportant facilement en LaTeX, or un tel symbole n'est pas utilisable de la sorte comme nom de variable, par exemple Δt sera s'exportera par défaut en chaîne de caractères en `"\Delta\ t"`. Les nœuds de l'arbre syntaxique sont des fonctions, il y a alors deux cas à distinguer, soit il s'agit d'une fonction dont la représentation en Python est la même qu'en C++, auquel cas aucune opération particulière n'est nécessaire, c'est le cas par exemple des opérations arithmétiques $+$, $-$, $\times$ et $\div$ qui sont représentées par les opérateurs binaires `+`, `-`, `*` et `/` en Python et C++ ; soit il s'agit d'une fonction dont la représentation Python et C++ diffère, auquel cas il est nécessaire de créer une fonction SymPy qui aura le même nom que la fonction C++ associée, et de substituer le nœud de l'arbre syntaxique par cette nouvelle fonction. La conversion en chaîne de caractère de l'arbre ainsi modifié sera une expression C++ valide. Il est possible d'améliorer l'expression C++ en faisant une évaluation numérique des nombre rationnels (et potentiellement aussi irrationnels) présents, pour limiter le nombre d'opérations dans l'expression finale. Ainsi l'expression `1/3` sera substituée par `0.333333333333333`, cela permet d'éviter des interprétation de fractions comme des divisions entières par le compilateur.

Pour chaque étage de la méthode LRK, il est ainsi possible d'obtenir une expression C++ valide par variable. L'étape supplémentaire pour assumer que l'on est un gros fainéant est d'utiliser un moteur de *template* pour insérer ces expressions dans un squelette de code qui s'adapte automatiquement au nombre d'étages de la méthode LRK, en initialisant et allouant les variables temporaires nécessaires. Ce travail est effectuer par le moteur de *template* Jinja2 qui est une bibliothèque Python permettant d'ajouter des opérations logiques en plus d'une

simple substitution de champs dans un squelette de code préexistant. Le squelette en pseudo-code d'un étage d'une méthode LRK est donné en exemple dans l'algorithme 1

Algorithme 1 Squelette de l'algorithme d'un étage s d'une méthode LRK

$\triangleright$ Calcul des variables $\hat{j}_{c,x}^{(s)}$, $\hat{j}_{c,y}^{(s)}$, $\hat{B}_x^{(s)}$, $\hat{B}_y^{(s)}$, $\hat{E}_x^{(s)}$ et $\hat{E}_y^{(s)}$
pour $i = 0, \dots, N_z$ **faire :**
  $\hat{j}_{h,x,[i]} \leftarrow \sum_{k_x,k_y,k_z} v_{k_x} \hat{f}_{h,[i,k_x,k_y,k_z]} \Delta v$
  $\hat{j}_{h,y,[i]} \leftarrow \sum_{k_x,k_y,k_z} v_{k_y} \hat{f}_{h,[i,k_x,k_y,k_z]} \Delta v$
fin pour
pour $i = 0, \dots, N_z$ **faire :**
  $\hat{j}_{c,x,[i]}^{(s)} \leftarrow \dots$ $\triangleright$ les expressions ici sont données par SymPy
  $\hat{j}_{c,y,[i]}^{(s)} \leftarrow \dots$
  $\hat{B}_{x,[i]}^{(s)} \leftarrow \dots$
  $\hat{B}_{y,[i]}^{(s)} \leftarrow \dots$
  $\hat{E}_{x,[i]}^{(s)} \leftarrow \dots$
  $\hat{E}_{y,[i]}^{(s)} \leftarrow \dots$
fin pour

$\triangleright$ Calcul de la variable $\hat{f}_h^{(s)}$
$(f)_{h,[\cdot,k_x,k_y,k_z]} \leftarrow \mathrm{iFFT}_z\left(\hat{f}_{h,[\cdot,k_x,k_y,k_z]}^{(s-1)}\right)$
pour tout $(k_x, k_y, k_z) \in [\![0, N_x]\!] \times [\![0, N_y]\!] \times [\![0, N_z]\!]$ **faire :**
  pour $i = 0, \dots, N_z$ **faire :**
    $a_{v_x} \leftarrow E_{x,[i]} + v_{k_y} B_0 + v_{k_z} B_{y,[i]}$
    $a_{v_y} \leftarrow E_{y,[i]} + v_{k_x} B_0 + v_{k_z} B_{x,[i]}$
    $a_{v_z} \leftarrow v_{k_x} B_{y,[i]} + v_{k_y} B_{x,[i]}$
    $\partial_v f_{h,[i,k_x,k_y,k_z]} \leftarrow \mathrm{WENO}(a_{v_x}, f_{h,[i,k_x-3:k_x+3,k_y,k_z]}) + \mathrm{WENO}(a_{v_y}, f_{h,[i,k_x,k_y-3:k_y+3,k_z]})$
      $+ \mathrm{WENO}(a_{v_z}, f_{h,[i,k_x,k_y,k_z-3:k_z+3]})$
  fin pour
fin pour
pour tout $(k_x, k_y, k_z) \in [\![0, N_x]\!] \times [\![0, N_y]\!] \times [\![0, N_z]\!]$ **faire :**
  $\left(\widehat{\partial_v f}\right)_i \leftarrow \mathrm{FFT}_z(\partial_v f_{\cdot,k_x,k_y,k_z})$
  pour $i = 0, \dots, N_z$ **faire :**
    $\hat{f}_h^{(s)} \leftarrow \dots$ $\triangleright$ l'expression ici est donnée par SymPy
  fin pour
fin pour

La mise en place de l'opération de filtrage dans le pseudo-code 1 nécessite seulement de modifier le calcul des variables de courants chauds $\left(\hat{j}_{h,x}\right)_i$, $\left(\hat{j}_{h,y}\right)_i$ et des vitesses d'advection

a_{v_x}, a_{v_y} et a_{v_z} :

$$\hat{j}_{h,x,[i]} \leftarrow \sum_{k_1,k_2,k_z} (w_1 \cos(B_0\tau^{n,s}) - w_2 \sin(B_0\tau^{n,s}))\hat{g}_{[i,k_1,k_2,k_z]}\Delta w \Delta v_z$$

$$\hat{j}_{h,y,[i]} \leftarrow \sum_{k_1,k_2,k_z} (w_1 \sin(B_0\tau^{n,s}) + w_2 \cos(B_0\tau^{n,s}))\hat{g}_{[i,k_1,k_2,k_z]}\Delta w \Delta v_z$$

$$a_{v_x} \leftarrow E_{x,[i]} \cos(B_0\tau^{n,s}) + E_{y,[i]} \sin(B_0\tau^{n,s}) + v_z B_{x,[i]} \sin(B_0\tau^{n,s}) - v_z B_{y,[i]} \cos(B_0\tau^{n,s})$$

$$a_{v_y} \leftarrow -E_{x,[i]} \sin(B_0\tau^{n,s}) + E_{y,[i]} \cos(B_0\tau^{n,s}) + v_z B_{x,[i]} \cos(B_0\tau^{n,s}) + v_z B_{y,[i]} \sin(B_0\tau^{n,s})$$

$$a_{v_z} \leftarrow -B_{x,[i]}(w_1 \sin(B_0\tau^{n,s}) + w_2 \cos(B_0\tau^{n,s})) + B_{y,[i]}(w_1 \cos(B_0\tau^{n,s}) - w_2 \sin(B_0\tau^{n,s}))$$

où $\tau^{n,s} = t^n + c_s\Delta t$.

Nota Bene : La bibliothèque SymPy contient des fonctions permettant la génération de code en C ou Fortran, mais le fonctionnement de celles-ci s'adapte mal à une intégration dans une boucle d'un code déjà existant. De plus les fonctions ainsi générés ne fonctionnent pas avec un code contenant des *template* C++, pour changer éventuellement de type pour de possibles optimisations. Elles ne prennent en paramètre que des valeurs par copie ou par pointeur, ce qui limite leur usage avec des structures de données évoluées proposées par les librairies C++. Il serait envisageable d'utiliser certains des mécanismes présents dans ces fonctions pour améliorer la génération de code proposé ci-dessus, en utilisant un parcours d'arbre syntaxique pour construire un *Abstract Syntax Tree* (AST) permettant la génération dans n'importe quel langage d'une expression. Et derniers points, ces fonctions sont très mal documentées (je les ai découverte alors que je générais déjà les lignes de code pour le Lawson-RK(3,3) et que celui-ci tournait bien), et elles laissent des 1/2, 1/3 etc. qui peuvent valoir 0 selon les options de compilation ou les compilateurs.

5 Résultats numériques

6 Approximation de la partie linéaire

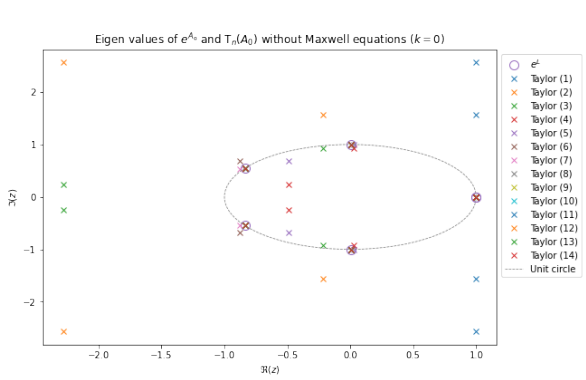
L'obtention, à l'aide d'un logiciel de calcul formel, de l'exponentielle de la partie linéaire n'est pas toujours envisageable. Il est possible de recourir à une méthode d'approximation pour obtenir une formulation formel de celle-ci qui sera possible d'utiliser pour l'écriture du code de simulation. On s'intéressera dans cette section à la partie linéaire L définie par :

$$L = \begin{pmatrix} 0 & -B_0 & 0 & 0 & \Omega_{pe}^2 & 0 & 0 \\ B_0 & 0 & 0 & 0 & 0 & \Omega_{pe}^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \partial_z & 0 \\ 0 & 0 & 0 & 0 & -\partial_z & 0 & 0 \\ -1 & 0 & 0 & -\partial_z & 0 & 0 & 0 \\ 0 & -1 & \partial_z & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -v_z\partial_z \end{pmatrix}$$

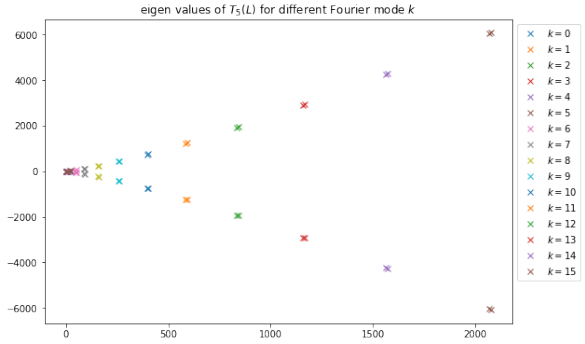
Cette matrice est de la forme :

$$L = \begin{pmatrix} A & 0 \\ 0 & -v_z\partial_z \end{pmatrix}$$

matrice diagonale par blocs, dont seul le bloc A pose problème pour calculer formellement l'exponentielle. Ainsi on s'intéressera surtout à la sous-matrice A obtenue après une transformée



(a) Les valeurs propres de e^{A_0} et de $T_5(A_0)$



(b) Les valeurs propres de e^A et de $T_5(A)$ pour différentes valeurs de $k \in [\![0, 15]\!]$, par symétrie on obtient aussi celles pour $k < 0$

FIGURE 1 – Valeurs propres de e^A et de $T_5(A)$ pour $k = 0$ (sans les équations de Maxwell) à gauche, et pour différentes valeurs de $k \in [\![0, 15]\!]$ à droite.

de Fourier en z du système :

$$A = \begin{pmatrix} 0 & -1 & 0 & 0 & 4 & 0 \\ 1 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & ik \\ 0 & 0 & 0 & 0 & -ik & 0 \\ -1 & 0 & 0 & -ik & 0 & 0 \\ 0 & -1 & ik & 0 & 0 & 0 \end{pmatrix}$$

Par abus de notation, nous noterons A_0, la matrice A pour $k = 0$, ce qui revient à une partie linéaire sans les équations de Maxwell, ceci sera utile lors de la comparaison des résultats entre les méthodes.

6.1 Troncature de la série de Taylor

On peut définir e^{tA} par la série de Taylor :

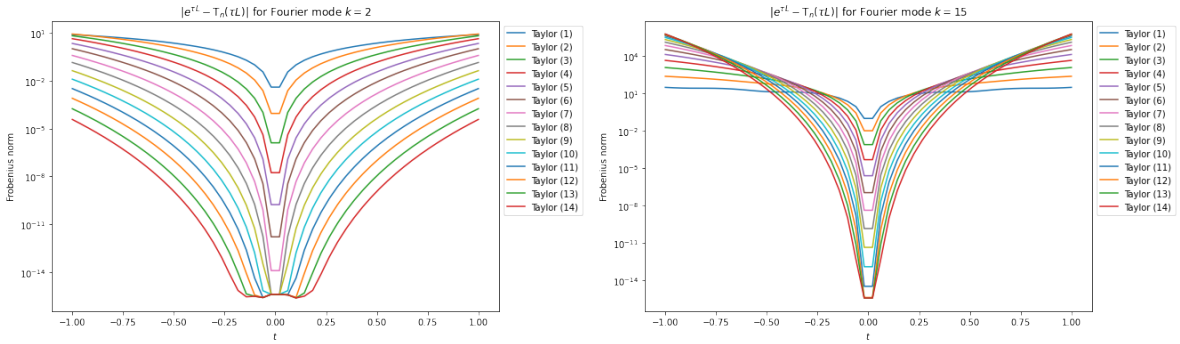
$$e^{tA} = \sum_{n=0}^{\infty} \frac{t^n A^n}{n!}.$$

Une troncature d'ordre suffisamment élevé permet d'obtenir une approximation de l'exponentielle e^{tA} à un ordre plus élevé que la méthode LRK(s,n) où elle sera utilisé garanti que l'erreur de troncature reste inférieur à n, l'ordre de la méthode en temps. On définit la troncature de la série de Taylor à l'ordre p par :

$$T_p(A) = \sum_{k=0}^{p} \frac{A^k}{k!}$$

On sait que les valeurs propres de A sont imaginaires pures, cela signifie que les valeurs propres de e^A sont de norme 1.

Je ne sais pas trop quoi dire sur les figures, donc je vais les mettre là un peu en vrac, on pourra discuter de leur intérêt plus tard, mais je pense qu'un petit calcul juste dire que les valeurs propres de $T_p(A)$ ne sont pas de module 1 serait plus intéressant.



(a) L'erreur absolue locale $\|e^{tA} - T_p(tA)\|$ pour le mode de Fourier $k = 2$

(b) L'erreur absolue locale $\|e^{tA} - T_p(tA)\|$ pour le mode de Fourier $k = 15$

FIGURE 2 – Erreur absolue locale $\|e^{tA} - T_p(tA)\|$ pour deux modes de Fourier $k = 2$ à gauche et $k = 15$ à droite.

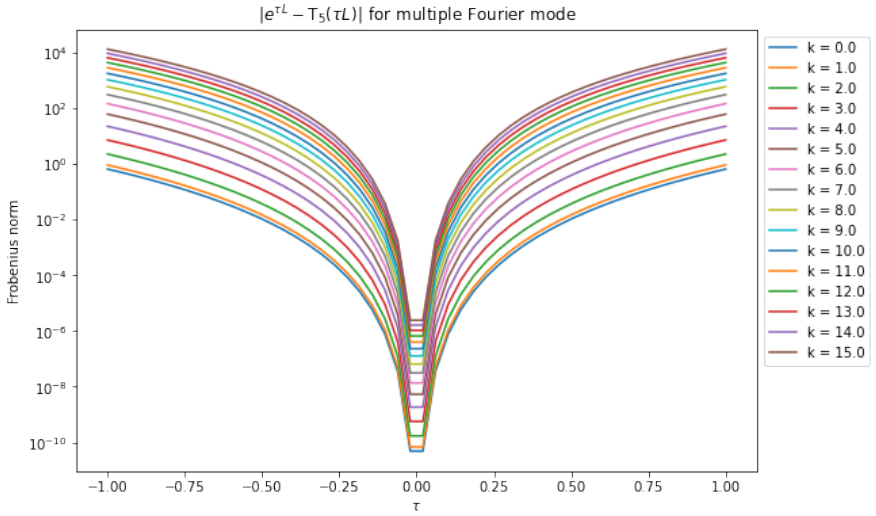
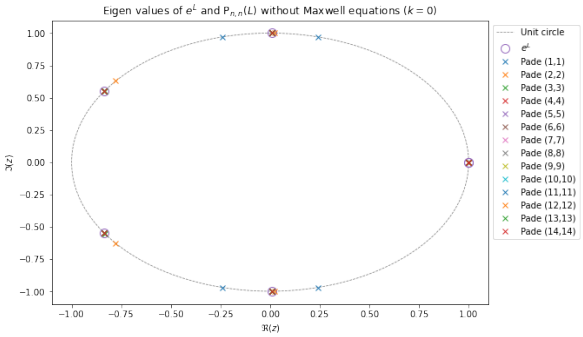
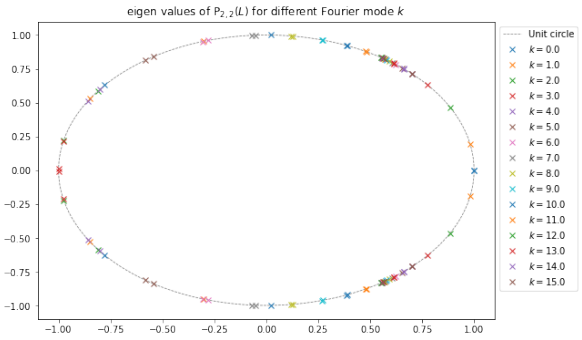


FIGURE 3 – L'erreur absolue locale $\|e^{tA} - T_5(tA)\|$ pour différents mode de Fourier



(a) Les valeurs propres de e^{A_0} et de $P_{n,n}(A_0)$



(b) Les valeurs propres de e^A et de $P_{2,2}(A)$ pour différentes valeurs de $k \in [\![0, 15]\!]$, par symétrie on obtient aussi celles pour $k < 0$

FIGURE 4 – Valeurs propres de e^A et de $P_{n,n}(A)$ pour $k = 0$ (sans les équations de Maxwell) à gauche, et pour différentes valeurs de $k \in [\![0, 15]\!]$ à droite.

6.2 Approximant de Padé

Pour approcher une fonction, au lieu d'utiliser un polynôme comme dans le cadre des séries des développement limités, il est possible de construire une fraction rationnelle. L'approximant de Padé de la fonction exponentielle est la meilleure approximation de la fonction exponentielle par une fraction rationnelle et est définie par :

$$h_{p,q}(x) = \sum_{i=0}^{p} \frac{\frac{p!}{(p-i)!}}{\frac{(p+q)!}{(p+q-i)!}} \frac{x^i}{i!}$$

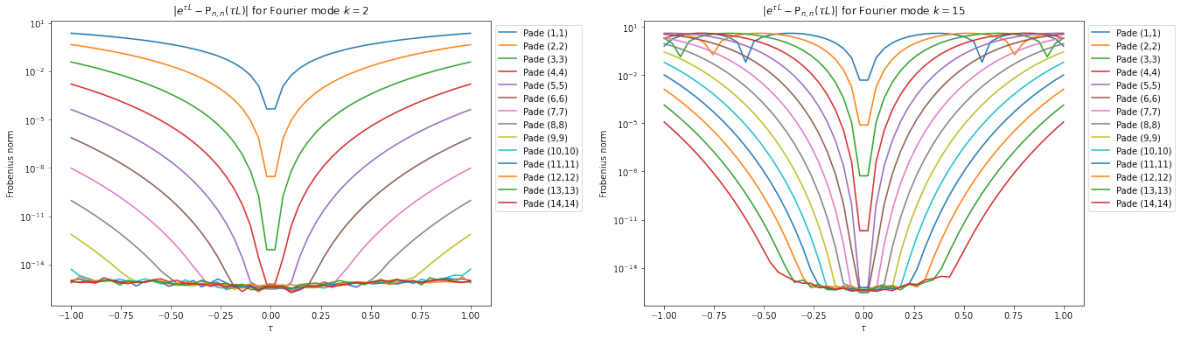
$$k_{p,q}(x) = \sum_{j=0}^{q} (-1)^j \frac{\frac{q!}{(q-j)!}}{\frac{(p+q)!}{(p+q-j)!}} \frac{x^j}{j!}$$

$$p_{p,q}(x) = \frac{h_{p,q}(x)}{k_{p,q}(x)} \approx e^x$$

Pour utiliser cet approximant de Padé, qui est une fraction rationnelle, avec des matrices il faut utiliser la définition suivante :

$$e^M \approx \mathrm{P}_{p,q}(M) = h_{p,q}(M) \cdot (k_{p,q}(M))^{-1}$$

On effectue la même étude qu'avec une troncature de la série de Taylor. On regarde donc dans un premier temps sur la figure 4 les valeurs propres dans le cas $k = 0$ et pour différentes valeurs de k.



(a) L'erreur absolue locale $\|e^{tA} - T_p(tA)\|$ pour le mode de Fourier $k = 2$

(b) L'erreur absolue locale $\|e^{tA} - T_p(tA)\|$ pour le mode de Fourier $k = 15$

FIGURE 5 – Erreur absolue locale $\|e^{tA} - T_p(tA)\|$ pour deux modes de Fourier $k = 2$ à gauche et $k = 15$ à droite.

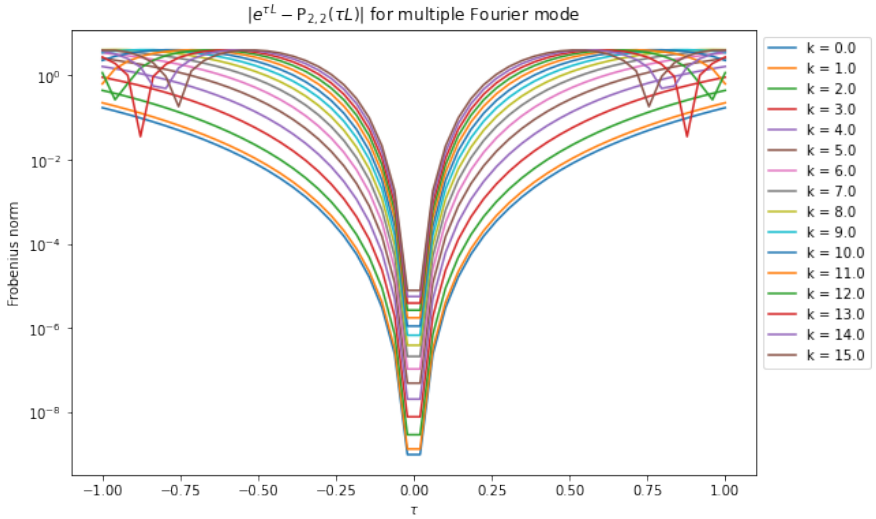


FIGURE 6 – L'erreur absolue locale $\|e^{tA} - T_5(tA)\|$ pour différents mode de Fourier

7 Les sections manquantes sur de l'optimisation de code

Je n'ai toujours pas touché à l'optimisation avec OpenMP. J'ai trouvé des références pour faire en sorte que ça se passe bien avec FFTW, donc ça ne devrait pas prendre trop de temps (je l'espère).

On avait aussi évoqué avec Nicolas de tester de la *mixed precision* c'est-à-dire de ne travailler qu'avec des `float` sur f_h et des `double` sur les autres variables, le problème (pour que ce soit performant) est d'avoir des algorithmes efficaces pour faire en sorte qu'une somme de `float` donne un `double`, j'ai cru voir des choses dans ce sens là dans la bibliothèque C++ que j'utilise : Boost.

N'est pas du tout évoqué une utilisation de MPI, les multiples FFT (algorithme non local) ne sont pas favorables à une telle mise en place. Pas de portage sur GPU n'est pas évoqué non plus. On remarque que dans l'algorithme 1 sur un étage que la partie sur f_h peut être traitée indépendamment des champs spatiaux, donc potentiellement sur GPU (qui gèrent par défaut des `float` et non des `double`) pendant que les champs sont calculés sur CPU, permettant des sorties de monitoring plus faciles des grandeurs intégrales. Bref c'est une ouverture possible.

8 Références bibliographiques