Rapport de Phase 2 du Groupe 1

Shimi Adam & Megna Anaël & Lemarchand Benoît

Vendredi, 29 Mai 2015

Table des matières

1 Introduction

Nous arrivons à la fin de ce projet, et il est maintenant temps d'analyser le travail qui a été fait par notre groupe, à la fois pour présenter nos résultats, mais aussi pour pouvoir nous améliorer.

Dans un premier temps, nous avons eu à implémenter deux méthodes d'analyse d'un modèle physique fourni (dérivée des Shallow Water Equations) :

— La reconstruction, qui consiste à partir de multiples itérations du modèle pour déduire le comportement de celui-çi pour des conditions initiales données.
— La classification, qui elle part aussi de multiples itérations du modèle, mais cette fois-çi pour déduire un paramètre d'entrée de celui-çi.

Notre travail personnel dans la mise en place de ces deux méthodes a été principalement d'implémenter un algorithme efficace de calcul des valeurs singulières et de choisir une méthode itérative pour la reconstruction.

En ce qui concerne l'algorithme pour la décomposition en valeurs singulières, il sera explicité dans la section suivante.

Quand à la méthode itérative que nous avons utilisée pour déterminer α dans la reconstruction, il s'agit d'une steepest descent sur l'équation normale $U_0^t * U_0 * \alpha = U_0^t * Z_0$.

Erratum. *Nous avons utilisé la steepest descent dans cette phase 1, alors que le critère d'arret sur la recherche des valeurs propres assure dans l'immense majorité des cas que toutes les valeurs singulières de U sont positives, et donc que* $U0^T * U0$ *est inversible.*

Nous pouvions donc directement résoudre l'équation normale (comme cela a été fait dans la correction).

Nous nous excusons encore une fois pour cette erreur stupide.

Dans un second temps, nous avons du implémenter la subspace Iteration Method en fortran, pour accélerer celle-çi, puis l'interfacer avec Matlab.

Enfin, nous avons conduit quelques tests sur ces implémentations, ainsi que d'autres sur une modification liée au modèle physique.

Maitenant, nous allons passer au rapport proprement dit.

2 Implémentation de la Subspace Iteration method en Fortran

Tout d'abord, nous rappelons le pseudo code fourni dans le compte-rendu de la phase 1, avant d'expliquer l'algorithme en lui-même, ainsi que de préciser l'implémentation en Fortran.

Algorithme 1 : Méthode du sous-espace singulier dominant

$V = matrice\ orthogonale\ quelconque \in \mathbb{R}^{m*l}$
$niter = 0$
$converged = 0$
$PercentReached = 0$
$normeA = \left\| Z^T * Z \right\|$
repeat
  for $i = 1, p$ **do**
    $V = Z^T * Y$
    $V = Z * Y$
  end
  $V = \text{Gram} - \text{Schmidt}(V)$ [a]
  $H = Z^T * V$
  $H = H^T * H$
  $[X, \Lambda] = decomposition\ spectrale\ de\ H$
  $[X, \Lambda] = ordonnancement\ decroissant\ de\ [X, \Lambda]$
  $V = V * X$
  for $i = converged + 1, n$ **do**
    if $\dfrac{\left\| Z * Z^T * V(i) - \Lambda(i,i).V(i) \right\|}{normeA} \leq \varepsilon$ **then**
      $converged = converged + 1$
      $PercentReached = 1 - \dfrac{\left\| Z - \sum_{j=1}^{i} \sqrt[4]{\Lambda(i,i)} V(i) V(i)^T Z^T \right\|}{\sqrt{normeA}}$
    else
      break
    end
  end
  $niter = niter + 1$
until $PercentReached \leq PercentInfo\ or\ niter < MaxIter$

a. Nous ne rappellons pas l'algorithme d'orthogonalisation de Gram-Schmidt, mais celui-çi est implémenté dans le code

Il est temps d'expliquer plus précisément cet algorithme. Comme expliqué dans le sujet de la phase 1, il est adapté de la power method, qui permet de calculer une à une les valeurs propres dans l'ordre décroissant d'une matrice carrée.

Comme dans l'algorithme de la power method, le but est ici de polariser des vecteurs selon les vecteurs propres en les multipliant par la matrice, afin de les faire tendre vers les dit vecteurs propres après suffisament d'itérations. Les principales différences ici étant que nous travaillons sur toutes les valeurs propres en parallèle (la raison pour laquelle notre algorithme peut renvoyer la proportion demandée de valeurs propres en une itération, alors que la power method prendrait dans le meilleur des cas la somme des multiplicités de ces valeurs propres.)

Une autre différence facilement traitable est le fait que nous cherchons ici les valeurs singulières et les vecteurs singuliers gauche d'une matrice rectangle. Il suffit de remarquer que ce sont les valeurs propres et les vecteurs propres de la matrice consistant en le produit de notre matrice rectangulaire avec sa transposée pour se ramener au cas carré.

Comme dit plus haut, la subspace method consiste à polariser non pas un vecteur selon la valeur propre dominante, mais une base de n vecteurs orthogonaux selon les n valeurs propres dominantes. Pour cela, la méthode est la même que dans la power method, c'est à dire une multiplication par la matrice carré dont on cherche les valeurs propres. A noter qu'il est possible de multiplier plusieurs fois la base par cette matrice à chaque itération, pour s'assurer que tous les vecteurs ne sont pas polarisé par la valeur propre dominante.

On n'oublie pas d'orthogonaliser après coup les vecteurs par gram-schmidt, afin de laisser à chaque vecteur "approximé" une polarisation selon un unique vecteur propre.

Ensuite, extraire les approximations de vecteurs propres que sont devenus les vecteurs de la base orthonormale s'avère un peu plus dur que dans la power method, étant donné qu'ils sont "enfouis" dans ces vecteurs.

L'idée est d'exprimer les vecteurs de cette base orthonormale dans la base de vecteurs propres de la matrice carrée. Pour expliquer cela, nous allons recourir aux notations de l'énoncé, c'est-à-dire que la matrice carrée dont nous cherchons les valeurs et vecteurs propres est notée A et que la matrice contenant la base des vecteurs orthogonaux polarisés par A puis orthogonalisés de nouveau est notée V. Ainsi, nous calculons la matrice $V^T * A * V$, aussi connue sous le nom de quotient de Rayleigh de la base V par la matrice A.

Puis, nous calculons les valeurs propres et les vecteurs propres du quotient de Rayleigh. En faisant le simple calcul $V * V^T * A * V * V^T = A$, on voit qu'il suffit de réexprimer les vecteurs propres du quotient de rayleigh dans la base de V (par une multiplication matricielle donc) pour obtenir une approximation des vecteurs propres de A

Enfin, pour décider si un couple valeur et vecteur propre est suffisament proche pour être accepté, nous calculons l'erreur du vecteur propre par la multiplication par A, rescalée par la norme de A pour être indépendante de la valeur de la valeur propre considérée. Plus formellement, si λ_i $v(:,i)$ sont respectivement la valeur propre et le vecteur propre approximé, nous calculons $\frac{\|A*v(:,i)-\lambda_i*v(:,i)\|}{\|A\|}$ et nous le comparons à l'erreur acceptée.

En ce qui concerne l'implémentation en fortran à proprement parler, il a surtout fallut déconstruire les calculs matricels et vectoriels complexes en séquences de primitives.

Aussi, nous avons rendu explicite les astuces utilisées dans la partie 1 pour éviter de calculer la matrice A, étant donné que dans le cas qui nous intéresse , celle-çi cause un Memory overflow.

3 Test de l'implémentation Fortran

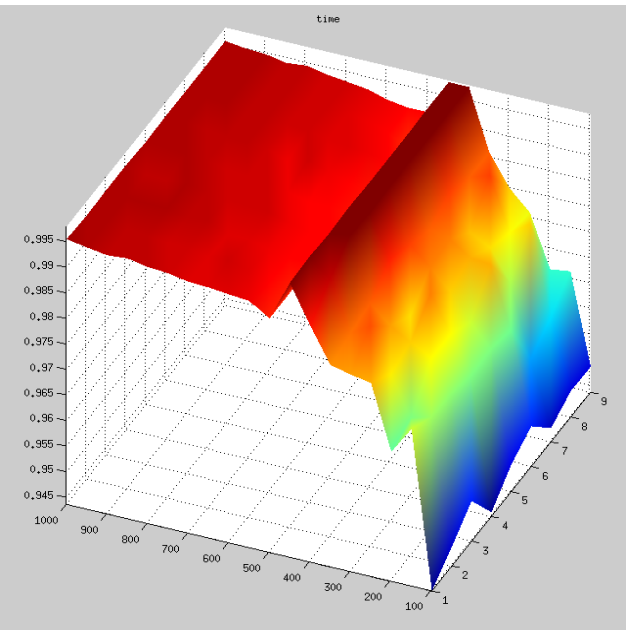
Dans cette partie, il s'agit d'analyser les résultats des comparaisons entre le temps mis par notre implémentation fortran, la même implémentation en version matrice symétrique (c'est-à-dire avec le calcul explicite de la matrice $A = Z * Z^T$) et la méthode dédiée de LAPACK, fortement optimisée par des années de recherche.

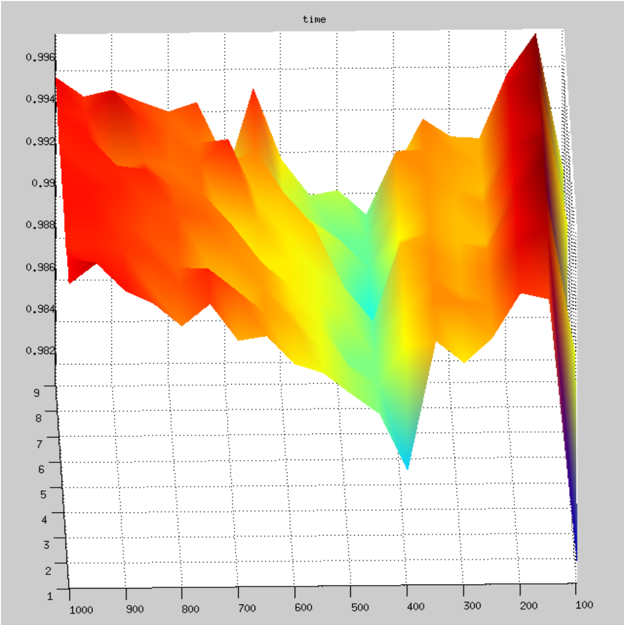
Tout d'abord, nous avons remarqué que la comparaison qui faisait le plus sens était celle de notre implémentation et de LAPACK. En effet, la version symétrique, en plus d'être la version de notre algorithme inapplicable à notre problème de départ , ne dévie de notre implémentation que par un facteur constant, le quotient du temps mis pour faire deux multiplications matricielle par le temps mis pour faire une seule multiplication matricielle, mais par le produit des deux matrices précédentes.

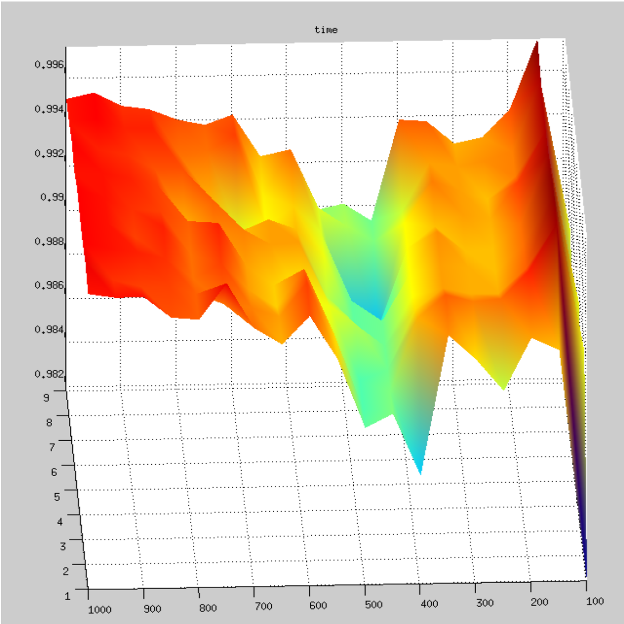
Ensuite, il y a la question de quels paramètres faire varier. p est un choix inévitable, étant donné que nous ne le faisons pas varier dans les tests du matlab. Pour avoir une nappe en trois dimensions, il manque une variable, que nous posons comme n.

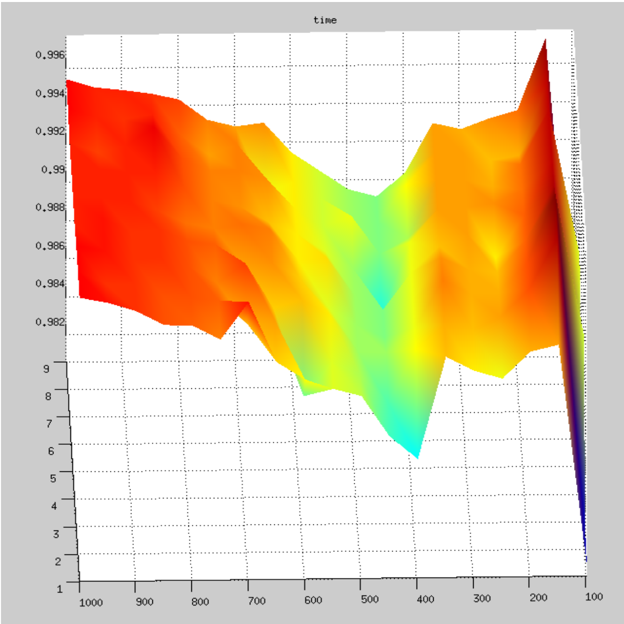
En effet, fixer m fait sens, étant donné que c'est ce qu'il se passera dans le problème où la méthode sera utilisée. La taille du sous-espace invariant de même.

Enfin, on refait les tests pour les 4 valeurs d'imat, afin de généraliser les resultats.




time
0.996
0.994
0.992
0.99
0.988
0.986
0.984
0.982
9
8
7
6
5
4
3
2
1
1000
900
800
700
600
500
400
300
200
100


time
0.996
0.994
0.992
0.99
0.988
0.986
0.984
0.982
9
8
7
6
5
4
3
2
1
1000
900
800
700
600
500
400
300
200
100


time
0.996
0.994
0.992
0.99
0.988
0.986
0.984
0.982
9
8
7
6
5
4
3
2
1
1000
900
800
700
600
500
400
300
200
100

Nous remarquons que les resultats avec les générateurs de matrice 2,3 et 4 sont assez similaires, alors que les resultats obtenus à l'aide du générateur de matrice 1 sont plus lisses. Nous choisissons de nous attarder sur les résultats pour imat = 4, la nappe la plus lisse des trois nappes les plus générales.

En ce qui concerne l'interpretation des resultats, nous allons le faire en deux temps.

Premièrement, les variations en fonction de n expriment comment les générateurs de matrice répartissent les valeurs singulières. En effet, la principale raison pour laquelle notre routine est plus lente que celle de LAPACK est dans un cas où il faut calculer beaucoup de valeurs singulières, et donc où l'aspect itératif montre ses limites.

Dans un second temps, les variations en fonction de p indiquent une notion assez intéressante, qui est celle de l'équilibre dans la répartition des valeurs singulières.

En effet, plus p est grand, plus il augmente le temps que prend une itération de la routine, étant donné qu'il augmente le nombre de produits matriciels.

Mais, pour certaines valeurs de p, plus de valeurs singulières peuvent converger en même temps, ce qui baisse le nombre d'itérations de la routine.

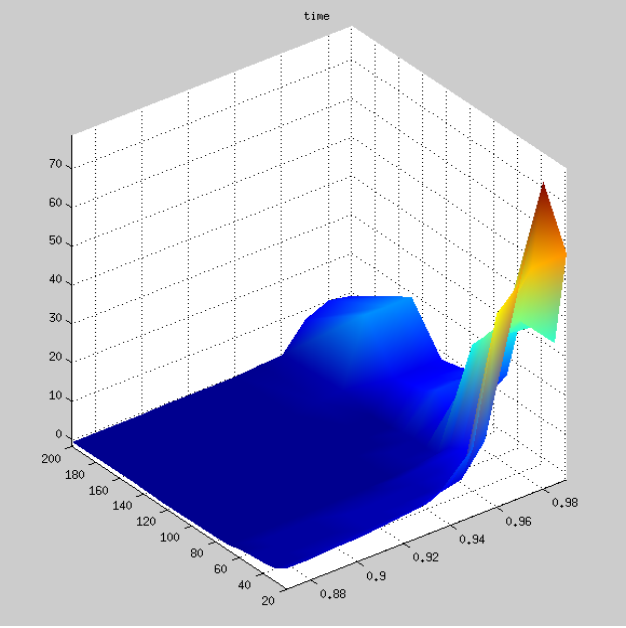
Comme nous le verrons dans la suite du rapport, adapter p à la matrice est le genre d'amélioration dont notre routine a besoin.

4 Test de l'implémentation Matlab

Les tests que nous avons mis en place pour comparer l'efficacité en temps et en précision de notre implémentation fortran face à la *svd* consiste, à l'image de ceux de la section précédente, au calcul, pour la même matrice, de l'erreur relative et du temps relatif par rapport à la *svd*

En ce qui concerne les paramètres variant, nous avons considéré qu'il était acceptable de prendre p fixe, étant donne que l'utilité d'un p différent de 1 dans certains cas a été étudié dans les expériences précédentes.

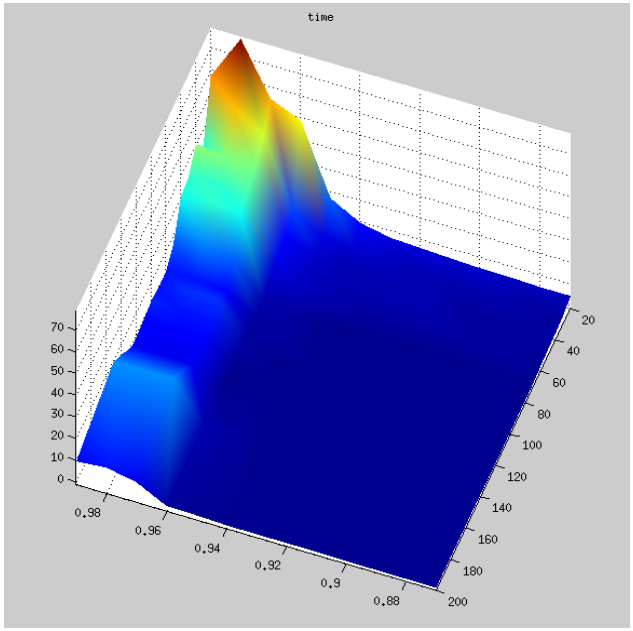
De plus, cela nous permet de construire des nappes de resultats en trois dimensions, selon Nens et percentInfo.



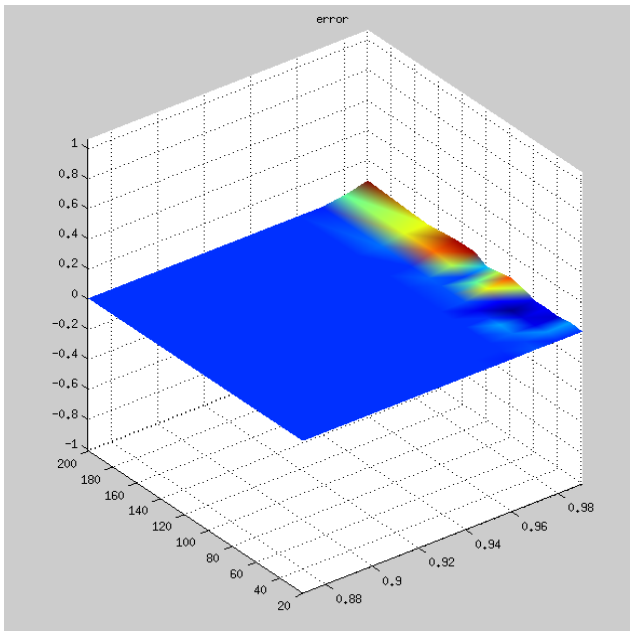
Une première observation sur ce graphique est que dans la grande majorité des cas traité, la nappe reste relativement proche du plan $z = 0$. Cela nous indique que si notre implémentation ne bat presque jamais la *svd* de matlab, il lui arrive de l'égaler.

Bien sur, ce qui nous intéresse vraiment concerne les valeurs des paramètres pour lequels notre fonction va clairement moins vite que *svd*.

Nous pouvons remarquer en premier lieu que plus Nens est faible, à percentInfo fixé, plus la *svd* gagne du terrain. Cela s'explique par le fait que réduire Nens revient à réduire le nombre de valeurs singulières, donc à avoir soit des valeurs singulières plus proches, auquel cas notre méthode aura du mal à converger pour toutes les valeurs singulières nous intéressant, soit à devoir prendre une grosse majorité des valeurs singulières (surtout que nous testons avec des percentInfo élevés), auquel cas notre algorithme aux multiples itérations se fait évidemment battre par un algorithme qui calcule toutes les valeurs singulières d'un coup.



Cet angle nous montre maintenant que la précision à partir de laquelle la *svd* surpasse largement notre algorithme est 0.96. Cela fait sens qu'un trop grand percentInfo nous pose des soucis, étant donné que l'on retombe dans ce cas dans le problème de la majorité des valeurs singulières : quel intéret de les calculer séparément si nous les voulons toutes ?



Enfin, l'erreur relative est très proche de zero, même aux extrêmes du graphe. Ainsi, notre implémentation, à défaut d'être toujours rapide, est presque toujours précise.

Nous remarquons tout de même que l'erreur relative augmente légèrement pour de grands Nens et percentInfo.

5 Addendum : prise en compte du modèle physique

Une fois l'implémentation en fortran et les comparaisons avec *svd* faites, nous avons voulu voir si nous pouvions aller plus vite que matlab. Evidemment, pour une matrice quelconque, ce désir est assez illusoire.

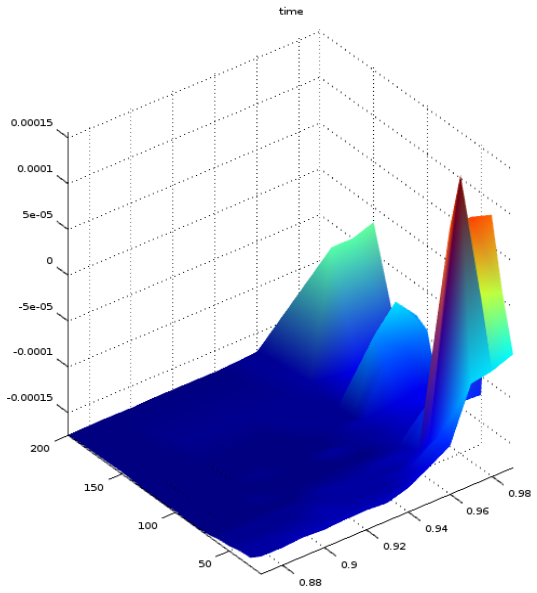
Mais nous avons un avantage par rapport à matlab en ce qui concerne les matrices sur lesquelles nous travaillons : nous savons qu'elles découlent d'un modèle physique, et donc nous avons une information de premier choix sur la manière dont les coefficients de cette matrice sont correlés.

L'idée derrière notre amélioration est d'échantillonner à nouveau toutes les colonnes de Z, en considérant, comme indiquée par les équations différentielles du modèle, que z(.,.,t) est dérivable au voisinage de tout point x,y de la grille. Ainsi, chaque point de la grille observé à l'instant t contient dans son z des informations sur ses voisins.

D'où l'hypothèse que prendre un coefficient de chaque colonne de Z sur trois suffit à nous donner les informations nécessaires, tout en réduisant la taille de la matrice à traiter.

Nous échantillonnons Z de cette manière avant de la fournir à notre méthode fortran, puis nous reconstruisons une U de la bonne taille en remplissant une matrice du format de Z, remplie de zeros, par les lignes de U, toutes les trois lignes.

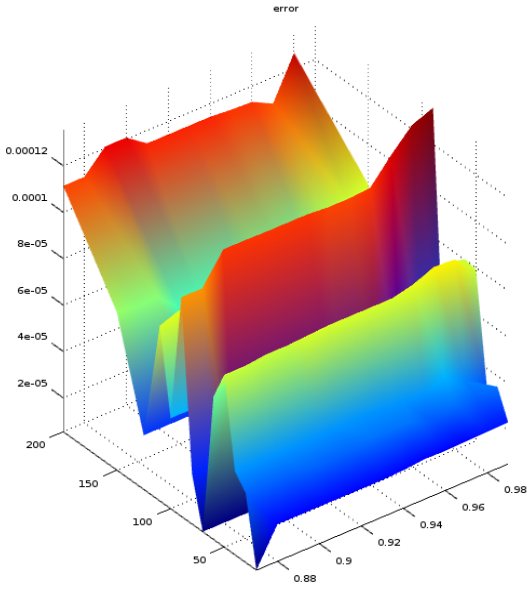
Voici les resultats :



Tout de suite, nous remarquons que cette nappe a la même forme que celle de notre méthode non modifiée, si ce n'est l'homotéthie qui rend cette nappe-ci majoritairement négative.

Cela signifie donc que dans la vaste majorité des cas, cette méthode est plus rapide, voir significativement plus rapide que la *svd*.

Cependant, tout ça n'a d'intérêt que si l'erreur reste acceptable.



Clairement, l'erreur relative est plus importante que pour notre méthode classique, mais si l'on regarde l'axe vertical, on remarque qu'elle reste très faible, bien que positive.

Un autre point intéressant est l'aspect de cette nappe d'erreur, avec une sorte d'oscillation selon l'axe des Nens. Pour l'instant, nous n'avons pas pu expliquer cet aspect.

Au final, l'idée d'échantillonner en tenant compte du modèle physique permet de gagner un temps significatif, tout en ayant une erreur relative très faible. Peut-être une piste à creuser.

6 Conclusions

Après avoir travaillé dessus durant tout un projet, la méthode du subspace itération nous semble à la fois puissante et adaptée à la situation, avec un bémol toutefois : la fixité de p.

Il manque vraiment à cette méthode un moyen de trouver le p "optimal" pour une matrice donnée, ce qui permettrait de "poncer" en grande partie les pics du temps relatif par rapport à la *svd*.

E n ce qui concerne le projet en général, il a eu le mérite de nous montrer une véritable application des techniques d'ALN, et de nous introduire à ce point de vue vectoriel et matriciel sur le monde physique, rempli de sous espaces significatifs et de polarisations selon une base.