TP4-5 Calcul Numérique :

LAPLANCHE Alexis

Introduction :

Durant le TP4 nous avons appris à comment utiliser des algorithmes spécifiques sur des structures de matrice tels que les matrices creuse ou les matrices symétriques. On a remarqué que lorsqu’on utilise les bons algorithmes sur les bonnes structures de données on peut diminuer le temps de calcul de manière significative.

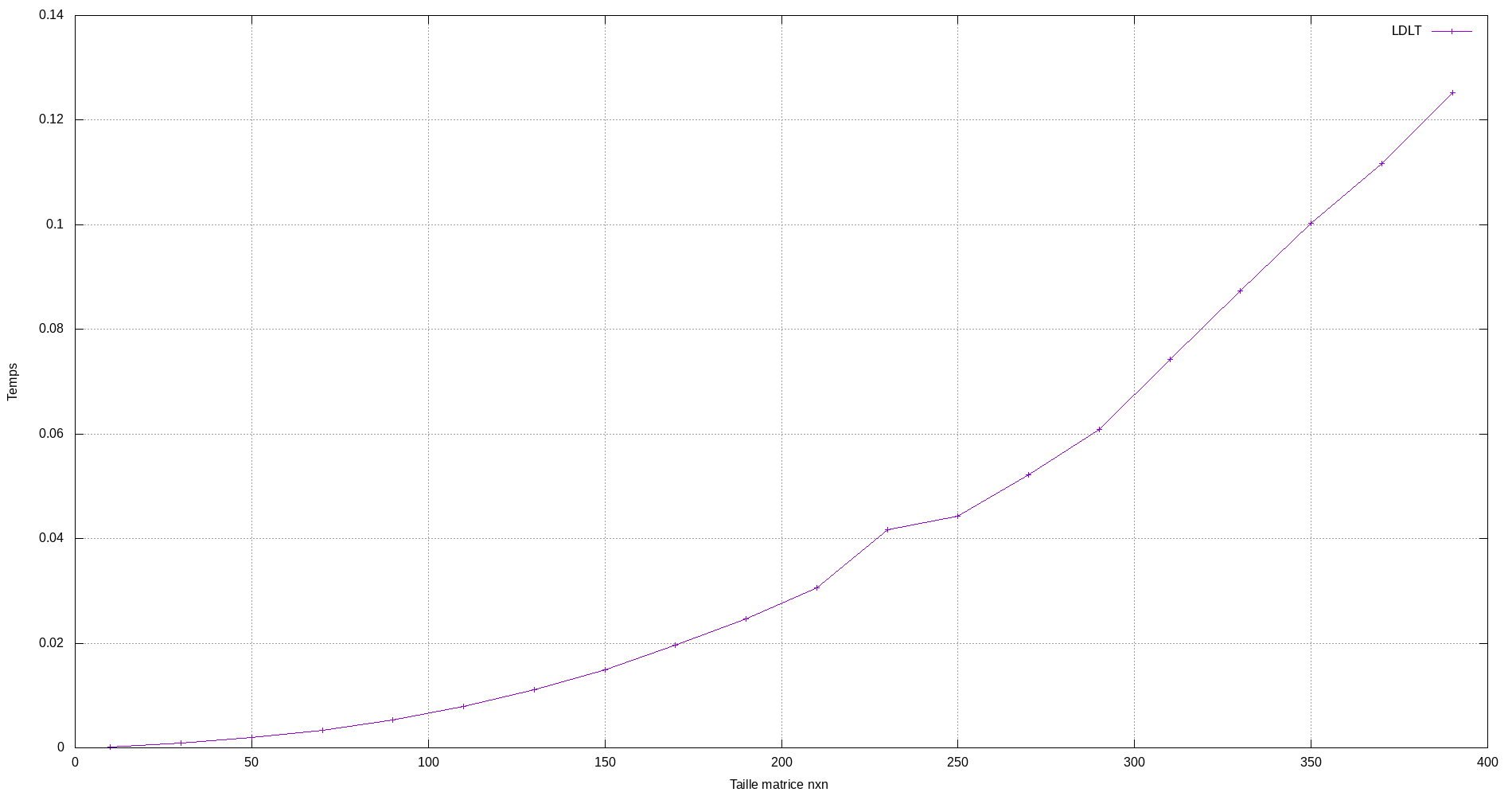
Pendant le TP5 nous avons appris à utiliser les bibliothèques de calcul matriciel BLAS et LAPACK en implémentant quelque algorithme.

TP4 : Exploitation des structures et calcul creux

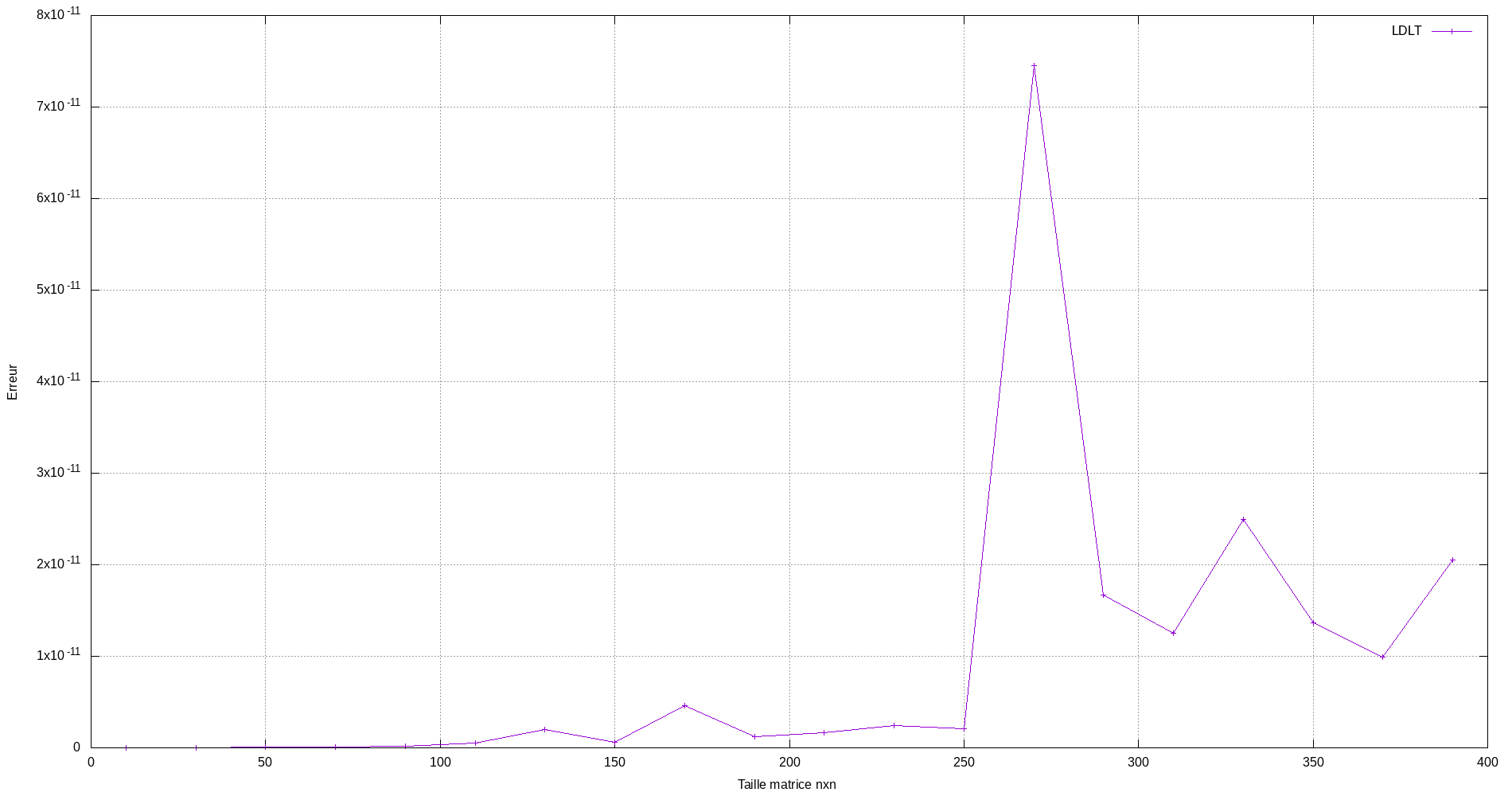
Exercice 1 : Factorisation LDLT

La factorisation LDLT est une méthode de factorisation d’une matrice symétrique qui permet de résoudre l’équation Ax = b avec A une matrice symétrique, x et b un vecteur. La résolution du problème se fait par étape de la façon suivante : Dans un premier temps on résout LTx = y, On résout ensuite Dy = z puis enfin Lz = b.

Pour l’algorithme de la factorisation LDLT  on observe une première boucle j allant de 1 à n. Avec n le nombre de ligne de la matrice A. Dans cette boucle il y a une boucle allant de 1 à j-1, on effectue donc ½ n répétition de cette boucle au total. Il y a ensuite une troisième boucle allant de j+1 à n qui effectue aussi ½ n répétition. La complexité est donc de n(1/2n + 1/2n) donc en O(n²). L’allure de la courbe (*fig1*) est en accord avec la complexité que l’on a trouvé : le temps d’exécution évolue de manière quadratique par rapport à la taille de la matrice.

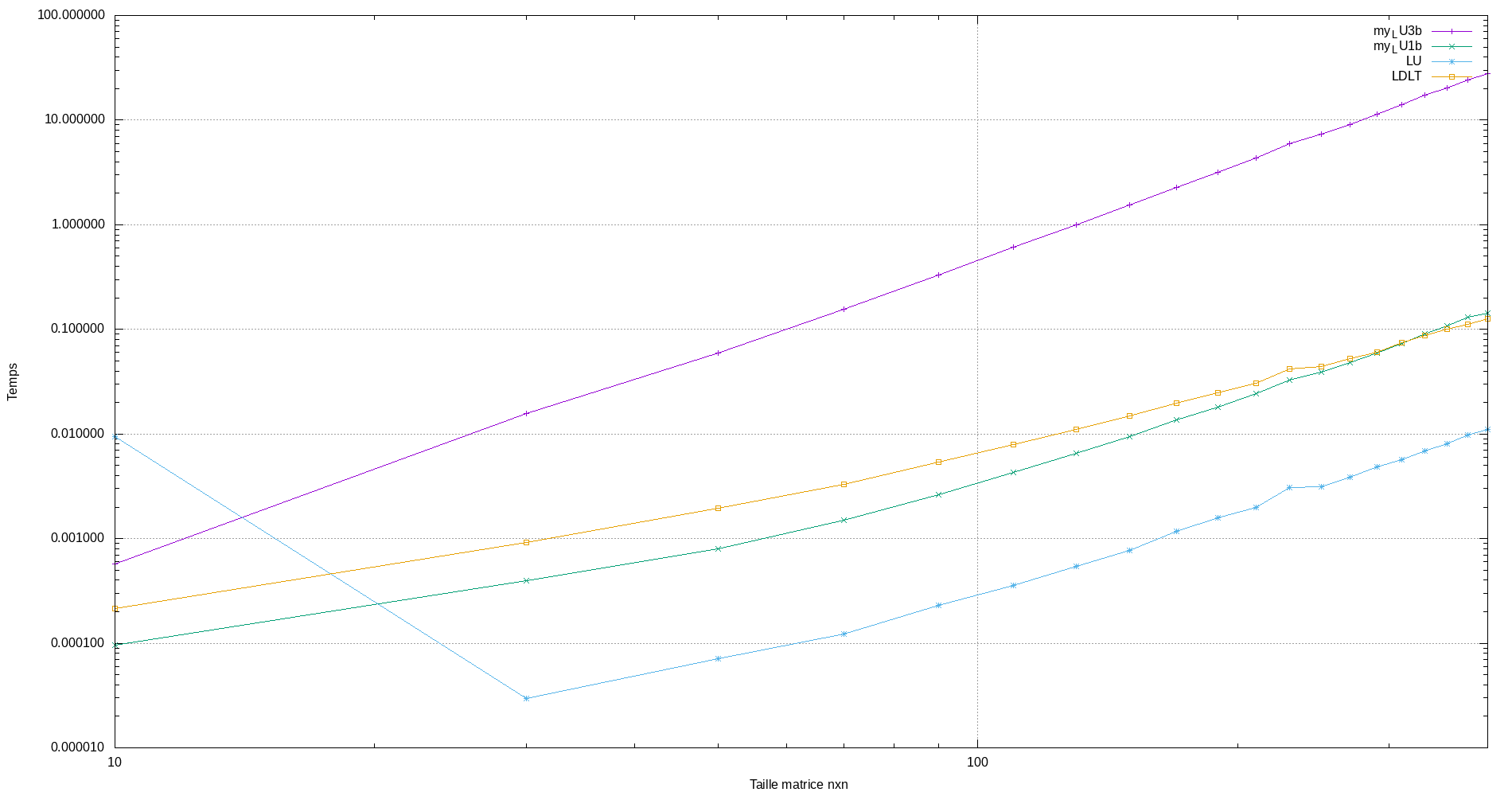


*Fig1 : Courbe du temps d’exécution par rapport à la taille de la matrice*



*Fig2 : Erreur en fonction de la taille de la matrice*

L’erreur généré par notre algorithme de factorisation LDLT semble assez élevé et indépendant de la taille de la matrice. Elle oscille entre 10-11 et 10-10. L’algorithme ne semble donc pas très stable.



*Fig3 : Comparaison du temps d’exécution de LU et LDLT*

*(Ici les échelles des axes x et y sont logarithmique car les différences du temps d’exécution était trop élevé)*

Sur la fig3 le temps d’exécution de l’algorithme de la factorisation LDLT est plus rapide que la version 3 boucles de la factorisation LU. Elle est légèrement plus rapide que la factorisation LU à 1 boucle pour les matrices de petites tailles maisquand on augmente la taille de la matrice LU 1 boucles deviens plus rapide. En revanche la version Scilab de la factorisation LU est plus rapide pour toutes les tailles de matrices notamment car elle fait appel à des noyaux de calcul optimisé.

Par contre l’algorithme de la factorisation LDLT est très intéressant en terme de stockage : il suffit de stocké une « unit lower triangular matrix» et une matrice diagonale. On peut donc stocker le résultat de la factorisation LDLT dans une unique matrice triangulaire inférieur alors que la factorisation LU nécessite une matrice complète pour son stockage.

*(Les codes sources de cet exercice sont trouvables sur le dépôt github dans le dossier TP4-code sous le nom LDLT.sci et test\_ldlt.sci)*

Exercice 4 : Stockage CSR et CSC

Dans cet exercice nous avons réalisé deux types de stockage de matrice creuse : un stockage CSR ainsi qu’un stockage CSC. Le stockage CSR consiste à stocker une matrice creuse dans trois tableau : AA, JA et IA. Le tableau AA contient toutes les valeurs non nulles de la matrice rangé dans l’ordre des lignes. Le tableau JA permet de stocker la colonne de la valeur dans AA au même indice/emplacement dans le tableau. Pour finir le tableau IA permet de stocker le pointeur vers le début de la ligne : c’est-à-dire à la combientième valeur commence la ligne de notre matrice.

Le stockage CSC est construit de la même manière sauf que le les valeurs dans AA sont stocké dans l’ordre des colonnes, JA stock l’emplacement de la ligne de la valeur de AA. IA stocke le pointeur vers le début de la colonne.

On remarque alors que le stockage CSR est plus efficace en termes de stockage pour les matrices ayant le moins de ligne et le stockage CSC pour les matrices ayant le plus de colonne : AA et IA dépendant de nnz (nombres de valeur non-nulles) et IA du nombre de ligne pour CSR et du nombre de colonne pour CSC.

De plus nnz << n donc le stockage des CSC et CSR sont nettement plus avantageux que le stockage plein pour des matrices creuses.

NB : Si l’on prend un stockage CSR d’une matrice et que le considère comme un CSC on obtient la transposer de la matrice : AA stock le nombre de valeur en ligne pour le CSR et en colonne pour le CSC considéré une ligne un stockage en ligne comme une colonne un stockage en colonne nous permet de transposer cette ligne. Si l’on effectue sa sur tout le tableau on obtient la transposer.

Nous avons ensuite réalisé un algorithme qui permet de transposer une matrice stocké en CSR sans repasser par sa forme pleine. Cet algorithme prend en entrée une matrice stocker en CSR et renvoie sa matrice transposer toujours stocker en CSR. Cet algorithme possède trois boucles :

On prend ici le pire cas possible pour calculer la complexité de notre algorithme.

La première boucle parcours les lignes de notre matrice elle effectue n itération avec n le nombre de ligne. La deuxième boucle parcourt la taille de notre tableau AA, elle effectue donc nnz itération. Il y a ensuite une troisième boucle qui nous permet de parcourir le tableau IA moins la dernière case, elle effectue donc n itérations. Ces trois boucles sont imbriqués, on obtient donc une complexité en O(n²\* nnz).

*(Les codes sources de cet exercice sont trouvables sur le dépôt github dans le dossier TP4-code sous le nom CSR.sci et CSRt.sci)*

Exercice 5 : Produit Matrice Vecteur Creux

Le but de cet algorithme est d’effectué un produit entre une matrice creuse stocké sous le format CSR avec une vecteur colonne. L’objectif de cette algorithme est d’effectué cet opération sans repasser par le format plein et donc d’éviter le cout de stockage et de transformation de cette matrice. Cet algorithme a l’avantage d’éviter le produit pour des membres nulles de la matrice et donc réduit le nombre d’opération par rapport à une multiplication matrice vecteur classique.

La première boucle de cet algorithme permet de parcourir les colonnes de notre matrice, elle effectue donc m opération, avec m le nombre de colonne de notre matrice creuse. La deuxième boucle parcourt le nombre d’éléments non nuls d’une ligne de notre matrice, à la fin de notre algorithme la deuxième boucle parcours tout le tableau IA qu’une seule fois qui est de taille nnz (nombres de valeurs non nulles de la matrice), la complexité total de notre algorithme est donc en O(m + nnz). Comme prévue, la complexité du produit matrice vecteur creux est nettement inférieur à la complexité d’un produit matrice vecteur plein (en O(n\*m)).

La complexité en stockage de notre algorithme est : 2 \* nnz + n+1 pour la matrice creuse, m pour le vecteur et n pour le résultat. La complexité en stockage est donc de O(2nnz + n+1 + m + n).

La complexité en stockage d’un produit matrice vecteur plein est en O(n\*m + n + m).

On remarque ici que l’algorithme ne génère pas d’erreur de calcul. Cela peut se justifié par le fait que l’on réalise exactement les même opérations que pour un produit matrice vecteur classique, notre algorithme diffère juste de par son stockage et le fait qu’il évite les opérations avec des valeurs nulles.

TP5 : Résolution de l’équation de la chaleur en 1D stationnaire

1) Travail préliminaire : Etablissement d’un cas de test

Exercice 1 :

Le problème de poisson 1D cherche à calculer le gradient de température à l’intérieur d’un mur. Ici on considère que le mur est en une dimension : on le considère infini vers le haut et vers le bas pour simplifier les calculs. On va chercher à calculer la température sur un nombre de point placé sur une ligne. Dans le cas de ce TD/TP on place une température T0 d’un côté du mur et une température T1 différente de T0 de l’autre côté du mur.

2) Méthode directe et stockage bande :

Format générale bande dans BLAS et LAPACK :

Le format générale bande permet de stocker des matrice possédant plusieurs diagonale tel que les matrices tridiagonale. Il existe deux formes de stockage général bande : générale bande en Row Major et en Col Major. Le format générale bande permet de stocker uniquement les diagonales non nuls de notre matrice dans un tableau a deux dimension (voir fig4). Ce type de stockage permet de nettement diminué la taille des matrices stocker : toute les diagonales nulles de la matrice ne sont pas stocker. La taille du stockage en mémoire est de min(n,m) \* (kl + ku + 1), avec n nombre de ligne, m nombre de colonne, kl nombre de diagonale inférieur et ku nombre de diagonale supérieur.

L’avantage de se stockage et qu’il conserve la forme de la matrice : les lignes et les colonnes de la matrice sont garder tel qu’elle et ne sont pas changer. Pour se faire on complète avec des 0 les diagonales plus courte dans notre tableau pour conserver l’alignement des lignes et des colonnes de notre matrice.

Les formes de stockage en Row Major et Col Major permettent de définir la dimension principale de notre stockage, c’est-à-dire si la première dimension de notre tableau GB (générale bande) représente une ligne ou une colonne (voir fig5 et fig6).

Fig4 : Matrice A et son format Générale Bande AB

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | a13 | a24 | a35 | a46 | 0 | a12 | a23 | a34 | a45 | a56 | a11 | a22 | a33 | a44 | a55 | a66 | a21 | a32 | a43 | a54 | a65 | 0 |

Fig5 : Exemple stockage Row Major en C

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | a11 | a21 | 0 | a12 | a22 | a33 | a13 | a23 | a33 | a43 | a24 | a34 | a44 | a54 | a35 | a45 | a55 | a65 | a46 | a56 | a66 | 0 |

Fig6 : Exemple stockage Col Major en C

Explication du code et de son architecture :

Le code se présente de la manière suivante :

Dans le dossier src on peut trouver les fichiers sources de notre code : à l’origine on y trouvait les fichiers suivant : lib\_poisson1D.c, tp2\_poisson1D\_direct.c, tp2\_poisson1D\_iter.c et tp\_env.c.

Dans le dossier include on peut trouver les « headers » de notre programme : à l’origine on y trouvait les fichiers suivant : blaslapack\_headers.h, lib\_poisson1D.h et tp\_env.h. Un headers permet de définir les fonctions que l’on va par la suite coder dans un fichier source. Ils permettent aussi notamment d’inclure les bibliothèques les externes ainsi que les bibliothèques standard du langage C.

Le fichier src/tp\_env.c permet de tester si l’environnement est bien configurer et que toutes les bibliothèques sont installées.

Le headers include/tp\_env.h ne contient aucune déclaration de fonction mais en revanche inclue plusieurs bibliothèques : <stdio.h> est une librairie standard du langage c, elle permet de gérer les entrées/sorties. Ici elle nous sert a affiché les résultats qui permettent de vérifier les bilbiothèques. <stdlib.h> est une bibliothèque standard qui contient de nombreuse fonction utile à la programmation. <math.h> est une bibliothèque mathématique, elle ajoute notamment de nombreuse fonction de calcul plus complexe et/ou avancé que les opérations de base. <float.h> est une bibliothèque qui permet d’ajouter différentes macro et constante, en lien avec les flotant simple et double précision. Par exemple ici, on utilise pour récupérer les valeurs maximum stockables dans des flottants simples et doubles précisions ainsi que la précision de ces flotants. La bibliothèque <limite.h> est similaire à la bibliothèque « float.h » mais pour les entier. Elle permet de récupérer la taille de chaque type d’entier en C. « blaslapack\_headers.h» est une bibliothèque qui n’est pas standard au C, elle permet « d’importer » ou d’inclure le contenu et donc les fonctions de ce fichier.

Le fichier src/lib\_poisson1D est le fichier qui comporte toute les fonctions utiles à la réalisation du TP. Ces fonction sont les suivantes : print\_GB(int size, double \*AB) n’est pas une fonction qui été a l’origine dans le programme. Cet fonction permet d’afficher le contenu d’une matrice GB « AB », elle prend en argument « size » qui est la taille de la matrice, c’est-à-dire le nombre de diagonal multiplié par le nombre de valeurs sur une diagonal (respectivement la et lab).

Les fonctions set\_GB\_operator\_rowMajor\_poisson1D(double \*AB, int \*lab, int \*la, int \*kv) et set\_GB\_operator\_colMajor\_poisson1D(double \*AB, int \*lab, int \*la, int \*kv) permettent de générer une matrice générale poisson 1D. La première génère la matrice en format « row major » alors que la seconde en format « col major ». Les paramètres de ces fonctions sont les suivants : AB : matrice de sortie, lab : nombre de lignes dans le tableau (nombre de diagonal + kv), la : nombre de valeurs sur la diagonale (nombre de colonnes de notre tableau GB), kv : présence ou non d’une ligne vide servant lorsque la matrice passe dans des algorithmes qui utilisent un pivot.

set\_dense\_RHS\_DBC\_1D(double\* RHS, int\* la, double\* BC0, double\* BC1) permet de générer le « Right hand side » de notre système, c’est-à-dire b dans l’équation Ax = b. set\_analytical\_solution\_DBC\_1D(double\* EX\_SOL, double\* X, int\* la, double\* BC0, double\* BC1) est la fonction qui nous permet de générer la solution exacte du problème, la solution exacte nous sert ensuite a mesuré l’erreur générer par notre algorithme. set\_grid\_points\_1D(double\* x, int\* la) permet de générer un vecteur qui nous sert pour l’affichage.

write\_GB\_operator\_rowMajor\_poisson1D(double\* AB, int\* lab, int\* la, char\* filename) et write\_GB\_operator\_colMajor\_poisson1D(double\* AB, int\* lab, int\* la, char\* filename) sont deux fonction qui permettent d’écrire une matrice GB dans un fichier. La première sert pour les matrices den row major et la seconde pour les matrices en col major.

La fonction write\_vec(double\* vec, int\* la, char\* filename) permet d’écrire un vecteur dans un fichier.

write\_xy(double\* vec, double\* x, int\* la, char\* filename) permet d’écrire dans un fichier deux vecteurs « côte à côte», cet fonction nous sert notamment à écrire la solution de l’équation Ax = b.

Le fichier inclue les headers suivant : <stdio.h>, <stdlib.h>, <math.h>, <float.h>, <limits.h> et "blaslapack\_headers.h".

Le fichier src/tp2\_poisson1D\_direct.c contient la fonction « main » du programme permettant de tester les solutions du problème obtenue à partir de méthode directe.

Dans le fichier tp2\_poisson1D\_direct nous avons effectué deux appelles a la fonction dgbsv provenant de la libraire LAPACKE. Ainsi qu’une appelle à la fonction dgbmv de la libraire BLAS pour chaque format générale bande.

Explication et validation de la fonction dgbsv :

La fonction dgbsv (pour double general band matrix factorization and multiple right-hand side solve) renvoie la solution de l’équation AX=B avec A une matrice générale bande,  B et X des matrices « classique ». Cette fonction est en fait une factorisation LU avec un pivot partiel pour ensuite résoudre le problème par remonter puis redescente de la manière suivante : Ly = Pb et Ux = y. Avec P notre matrice de permutation.

On remarque que dans notre format général bande, on a besoin d’une ligne vide pour pouvoir effectuer les algorithmes de pivot, on définit donc kv qui permet de définir combien de ligne vide sont dans notre stockage GB, lignes qui peuvent être utilisé par les algorithmes contenant un pivot.

Le prototype de la fonction est le suivant :

info = LAPACKE\_dgbsv (matrix\_layout, n, kl, ku, nrhs, a, lda, ipiv, b, ldb)[1](https://www.ibm.com/docs/en/essl/6.2?topic=blaes-sgbsv-dgbsv-cgbsv-zgbsv-general-band-matrix-factorization-multiple-right-hand-side-solve). Dans cette fonction *info*  est une valeur de retour qui permet de savoir comment c’est passé l’exécution de notre programme. Si lors de l’exécution de l’algorithme, une erreur c’est produise, un code d’erreur sera écris dans la variable info. Si info = 0 l’exécution c’est passer normalement, si info > 0, un zéro est présent sur la diagonale de la matrice triangulaire supérieur U, aux coordonnés info,info. Dans ce cas-là la factorisation LU a été faite mais la solution n’a pas été calculée. Si info < 0, le –(info)ème argument a une valeur interdite. L’argument *matrix\_layout* permet de spécifié sous quel format générale bande est notre matrice. Cet argument peut prendre seulement deux valeurs : LAPACK\_ROW\_MAJOR ou LAPACK\_COL\_MAJOR.  *n*  est le nombre de lignes de la matrice B et l’ordre de la matrice A. « kl » est le nombre de diagonal inférieur de notre matrice A. « ku » est le nombre de diagonal supérieur de notre matrice A. *nrhs* est le nombre de « right-hand sides » c’est-à-dire le nombres de colonnes de la matrice B. *a*  est la matrice générale bande A. *lda* dimension principale de A, c’est-à-dire le décalage que l’on doit appliquer à notre matrice générale bande pour changer de ligne. *ipiv* est une vecteur de taille n contenant les indices des pivots. *b* contient la matrice B, c’est-à-dire les nhrs right hand side. *ldb* la dimension principale de la matrice B.

Pour valider l’appelle à dgbsv, on va résoudre un problème dont on connait déjà la solution. On utilise la ligne suivante pour générer la solution du problème Poisson 1D :

En row major : LAPACKE\_dgbsv(LAPACK\_ROW\_MAJOR,la, kl, ku, NRHS, AB, la, ipiv, RHS, NRHS) avec *la* le nombre de la valeur sur la diagonal. Ici *kl*=1 et *ku*=1, *NRHS* = 1, *AB* est notre matrice poisson1D en format générale bande, *RHS* notre vecteur de température, *NRHS*=1.

En col major : LAPACKE\_dgbsv(LAPACK\_COL\_MAJOR,la, kl, ku, NRHS, AB, lab, ipiv, RHS, la) avec *lab* = 4.

Pour mesuré l’erreur relative générer par dgbsv on utilise la formule suivante : . Pour cela on réalise d’abord l’opération x – avec la ligne de code suivante :

cblas\_daxpy(la, -1.0, RHS, 1, EX\_SOL, 1), daxpy (y = α \* x + y, avec a un scalaire, x et y deux vecteurs). Ici *la* est la taille de nos vecteur *RHS* et *EX\_SOL*, *-1* est le scalaire a, les deux *1* sont respectivement l’incrémentation de x et de y, dans notre cas l’incrémentation est a 1 car les deux vecteurs sont ranger dans un tableau et il n’y a pas de décalage entre deux valeurs consécutive de notre vecteur. On prend ensuite la norme de cette opération, on obtient donc le résultat du calcul suivant :

||x – ||. Dans notre programme on utilise les deux fonctions suivantes pour effectuer ce calcul :

relres = cblas\_ddot(la, EX\_SOL, 1, EX\_SOL,1) et relres = sqrt(relres). Avec relres l’erreur relative. Il nous reste plus qu’a divisé par la norme de notre solution exacte, ce qu’on fait de la manière suivante : temp = cblas\_ddot(la, RHS, 1, RHS,1) et temp = sqrt(temp) pour le calcul de la norme de la solution exacte. On réalise ensuite la division suivante : relres = relres / temp qui nous donne l’erreur relative générer par notre appelle à dgbsv.

Pour l’appelle à dgbsv en col major on obtient l’erreur suivante : 5.889846e-15 et pour l’appelle en row major : 5.889846e-15. On remarque ici que l’erreur est très petite, on peut donc conclure que notre appelle à dgbsv nous donne une solution juste. On remarque aussi que l’erreur en col et row major sont identiques, ce qui est logique car l’algorithme reste le même, les deux tableaux ne sont justes pas parcouru de la même manière. Les calculs effectué restes donc les même qu’importe la forme de notre générale bande.

Explication et validation de la fonction dgbmv :

La fonction dgbmv (pour Matrix-Vector Product for a General Band Matrix) cette fonction permet d’effectué le calcul suivant : y = βy + αAx. On aussi la possibilité de demandé à la fonction de prendre la transposer de A.

L’appelle à la fonction se fait de la manière suivante :

cblas\_dbgmv (cblas\_layout, cblas\_transa, m, n, ml, mu, alpha, a, lda, x, incx, beta, y, incy)[2](https://www.ibm.com/docs/en/essl/6.2?topic=mvs-sgbmv-dgbmv-cgbmv-zgbmv-matrix-vector-product-general-band-matrix-its-transpose-its-conjugate-transpose), avec *cblas\_layout* = CblasRowMajor pour une matrice GB en row major et *cblas\_layout*= CblasColMajor pour une matrice GB en col major. *cblas\_transa*= CblasNoTrans si on ne souhaite pas transposer la matrice A et *cblas\_transa*= CblasTrans si l’on veut transposer A. *m* est le nombre de ligne de la matrice A, *n* est le nombre de colone de A. *ml* est le nombre de diagonale inférieur et *mu* le nombre de diagonal supérieur. *alpha* est le scalaire constant α. *a* est la matrice A en format GB avec *lda* sa dimension principale. *x* est les vecteur x et *incx* est le pas entre deux valeurs de x. *beta* est le scalaire β. *y* est le vecteur y avec *incy* le pas entre deux valeurs dans le vecteur y.

Pour valider l’appelle à dgbmv on va effectuer le cacule suivant : Ax – b qui doit renvoyer un vecteur remplit de 0 si la fonction a été exécuter correctement car on sait que x est la solution de Ax = b. On peut ensuite calculer la norme de ce vecteur résultat et s’assurer que la norme soit bien égale 0. On effectue donc l’appelle à dgbmv en row major et en col major pour vérifier que l’on obtient bien le résultat attendu.

Appelle à dgbmv en col major :

On appelle dgbmv de la façon suivante : cblas\_dgbmv(CblasColMajor,CblasNoTrans,la,la,kl,ku,alpha,AB,lab,EX\_SOL,incx,beta,RHS\_cpy,incy) avec *la* la dimension de notre matrice (ici notre matrice est carrée), *kl* = 1, *ku* = 1, alpha = 1, *AB* est notre matrice poisson1D générale bande, *lab*= kv+kl+1, *EX\_SOL* la solution exacte au problème poisson1D. *incx* = 1 ; *bêta* = -1pour pouvoir effectué la soustraction, RHS\_cpy est notre vecteur température et *incy* = 1.

Résultat dgbmv en col major :

dgbmv nous renvoie un vecteur y. On calcul sa norme en effectuant un produit scalaire de y avec lui-même et en prenant la racine carrée du résultat. On trouve que la norme de y est 0, l’appelle à dgbmv est donc correcte et nous renvoie le bon résultat.

Appelle à dgbmv en row major :

Dans notre programme on appelle dgmv en row major de la manière suivante :

cblas\_dgbmv(CblasRowMajor,CblasNoTrans,la,la,kl,ku,alpha,AB,la,EX\_SOL,incx,beta,RHS\_cpy,incy). L’appelle à dgbmv pour une matrice GB tow major se fait quasiment de la même manière que pour une en col major mit à part la dimension principale de AB et la constante CblasRowMajor.

Résultat dgbmv en row major :

Ici lorque l’on calcul la norme du vecteur on obtient le résultat suivant : 3.605780e+01. On peut donc en conclure qu’il y a une erreur avec l’appelle de dgbmv en row major.

Factorisation LU pour les matrices tridiagonales :

Les sources de cet exercice sont les fichiers src/lib\_lutri.c et src/tp2\_facto\_lu.c. Le fichier lib\_lutri.c contient toute les fonctions nous permettant d’effectué la factorisation LU ainsi que des tentatives de méthodes de remonté et redescente. Le fichier tp2\_facto\_lu.c est le code source de nos test permettant de validé notre factorisation LU.

On a réalisé ici la version compacte de LU c’est-à-dire que notre fonction prend en entrée une matrice GB A et nous renvoie une matrice GB A contenant L et U.

Lors de l’implémentation de la factorisation LU pour une matrice générale bande, on observe que le diagonal supérieur ne doit pas être modifié. On remarque ensuite que le calcul du diagonal inférieur dépend uniquement d’elle-même et de la valeur de la diagonal de la matrice. On peut donc obtenir la formule générale suivante : A[2\*la+i] = A[i]/A[la+i]. Pour la diagonal de la matrice on trouve la forme générale suivante A[la + i + 1]=A[la+i+1]-A[i + 1] \* A[la\*2+i]. Il nous suffit ensuite de parcourir tout le tableau GB et appliqué ses formules. Pour cela il nous suffit d’une seule boucle faisant (la-1) itération avec *la* le nombre de valeur dans une diagonal. On obtient donc une complexité en O(la). Il est donc plus efficace pour une matrice tridigonale d’effectué une factorisation LU en GB pour les matrices tridiagonale que d’utiliser un algorithme de factorisation LU classique.

L’appelle à notre factorisation LU se fait de la manière suivante :

my\_GB\_tri\_facto\_lu(double \* A, int la), avec *A* la matrice GB tridiagonale et *la* la taille des diagonales.

Pour vérifier notre algorithme de fonction LU nous avons deux solutions, la première aurait été de résoudre le problème Ax = b à l’aide d’une factorisation LU et d’une méthode de remonter et de redescente. En utilisant ça sur notre problème poisson1D on aurait pu comparer le résultat avec la solution générale qui est connue et mesuré l’erreur. En revanche, n’ayant pas réussi à coder les méthodes de remontés et redescente en c. J’ai utilisé une autre méthode qui consiste a retrouvé A avec la formule suivant : A = L\*U. Pour valider notre fonction LU on mesure donc l’erreur relative suivante : . Avec notre programme on obtient une erreur relative nulle. On peut donc en conclure que notre factorisation LU est fonctionnel.

3) Méthode de résolution itérative :

Les méthodes de résolution itérative permettent de trouver une solution approché au problème Ax = b en générant une suite qui va converger vers la solution x. A chaque itération on calcul une solution au problème et on calcule le résidu de cette solution (l’erreur relative) et on la compare à notre seuil de tolérance. Si l’erreur est inférieure à notre seuil, on a calculé la solution au problème avec une erreur inférieur à notre seuil de tolérance.

Méthode de Jacobi :

La méthode de Jacobi est une méthode itérative. Elle repose donc sur les mêmes principes énoncés précédemment. La méthode de Jacobi repose sur la formule suivant, qui sera exécuté à chaque itération : xk+1 = xk + D-1 (b - Axk). Pour pouvoir utiliser la méthode de Jacobi, il ne doit y avoir aucun zéro sur la diagonale et donc que det(D) ≠ 0. S’il y a un zéro sur la diagonale de A on ne pourra pas inverser la diagonale D.

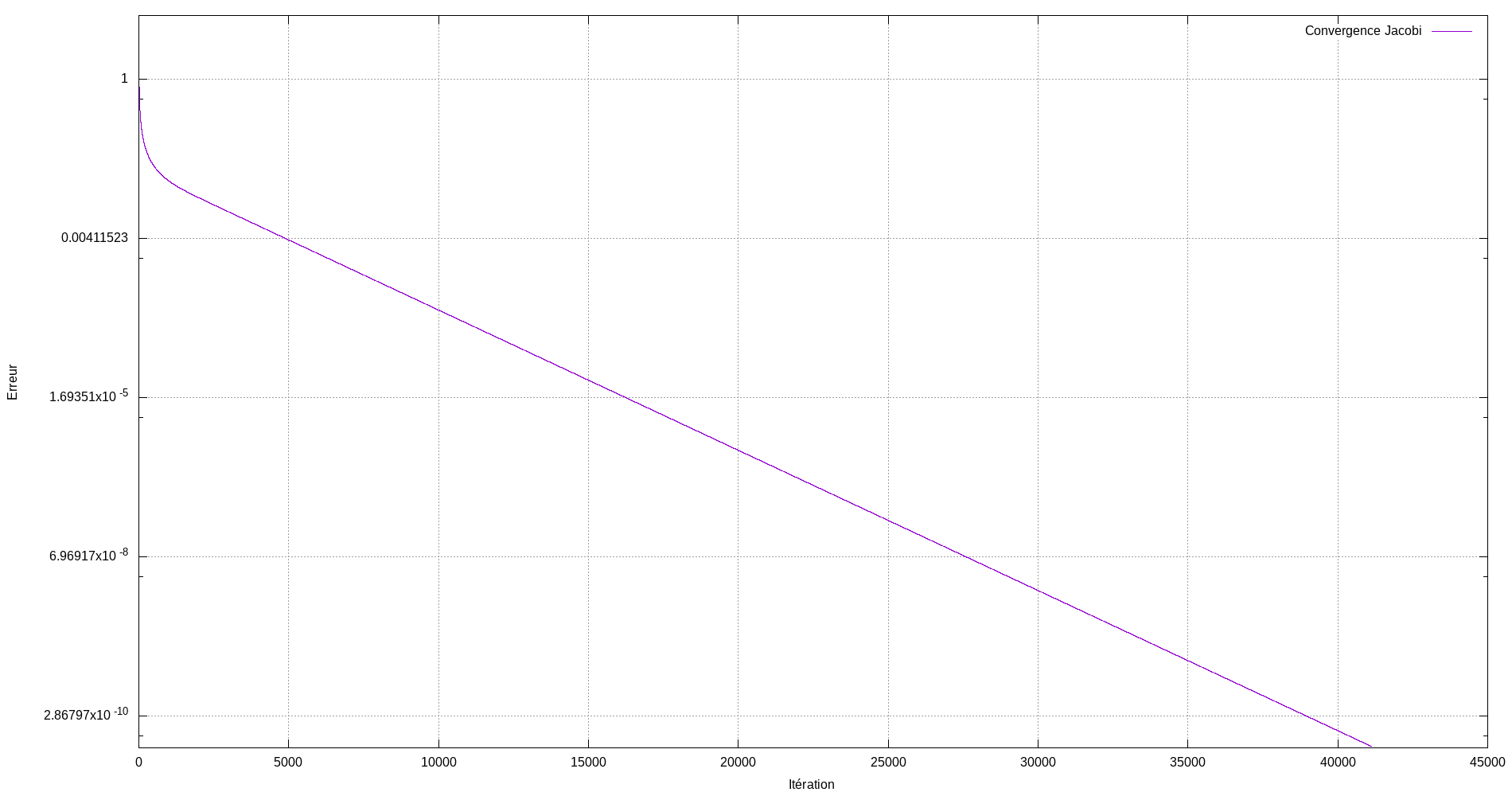


Fig7 : Convergence méthode de Jacobi

Sur la figure 7 on observe que la méthode de Jacobi converge pour le cas de Poisson 1D.

Pour que la méthode de Jacobi converge il faut que la diagonale de notre matrice soit dominante : c’est-à-dire qu’une valeur sur la diagonale doit être supérieure à la somme des valeurs sur sa ligne ainsi qu’être supérieur à la somme des valeurs de sa colonne.

Méthode de Richardson :

La méthode de Richardson est aussi une méthode itérative, comme Jacobi elle repose sur les principes énoncés précédemment. A chaque itération la méthode de Richardson exécute la formule suivante :

xk+1 = xk + α(b - Axk).

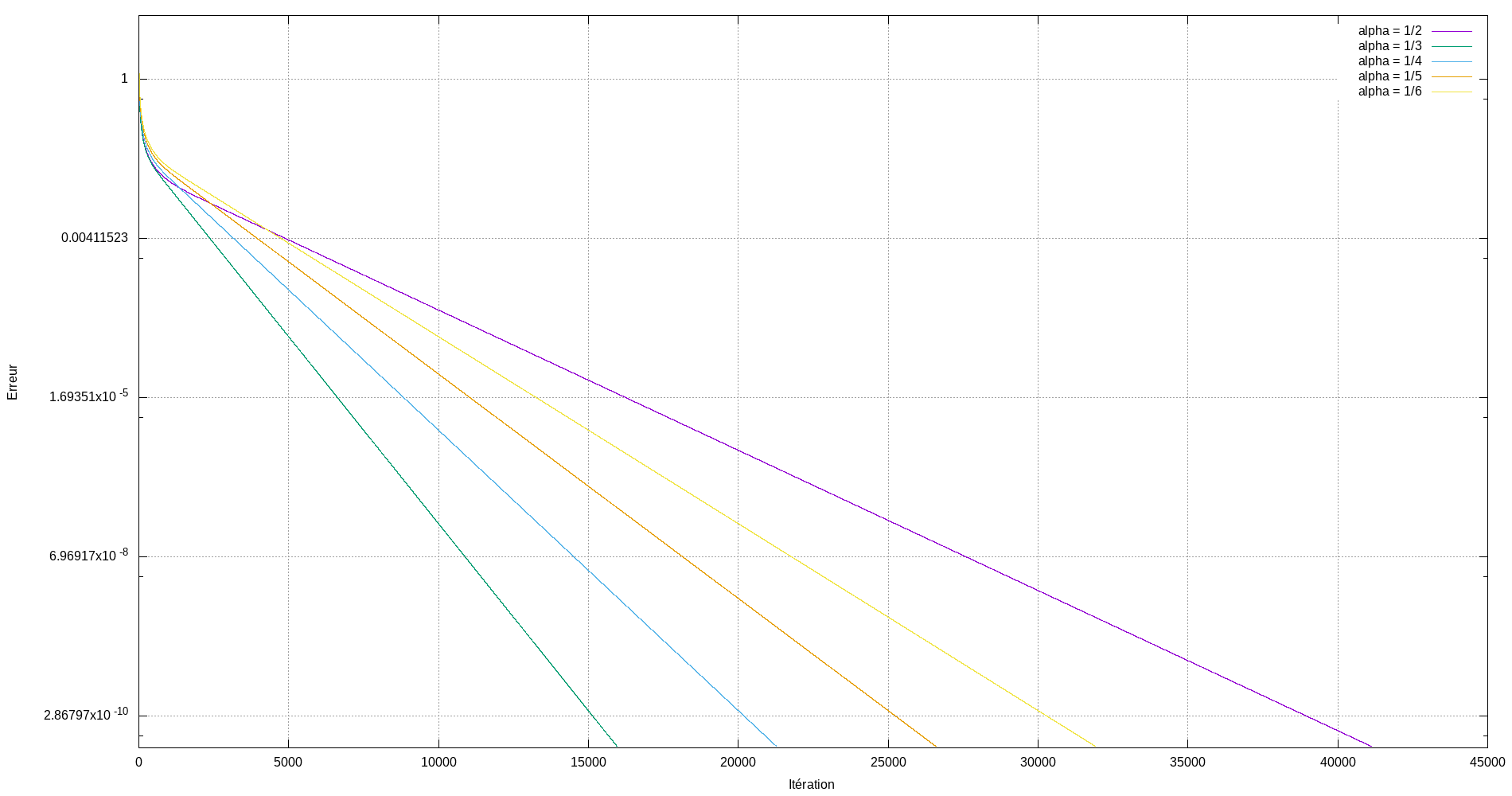


Fig8 : Convergence Richardson pour différente valeur de α

Ici on remarque que plus alpha est petit, plus la méthode converge rapidement vers la solution. On remarque aussi que si alpha est supérieur à 1, la méthode ne converge plus et ne fonctionne donc plus. En revanche plus alpha est proche de 0, plus on a de risque de dépasser la solution et donc d’obtenir une erreur plus importante.

Sources et Références :

<https://www.netlib.org/lapack/>

<https://www.ibm.com/docs/en/essl/6.2?topic=reference-lapack-lapacke>

<https://www.netlib.org/blas/>

<https://www.ibm.com/docs/en/xl-c-and-cpp-aix/16.1?topic=libraries-using-basic-linear-algebra-subprograms-blas>

1: <https://www.ibm.com/docs/en/essl/6.2?topic=blaes-sgbsv-dgbsv-cgbsv-zgbsv-general-band-matrix-factorization-multiple-right-hand-side-solve>, <http://www.netlib.org/lapack/explore-html/d3/d49/group__double_g_bsolve_gafa35ce1d7865b80563bbed6317050ad7.html>

2: <https://www.ibm.com/docs/en/essl/6.2?topic=mvs-sgbmv-dgbmv-cgbmv-zgbmv-matrix-vector-product-general-band-matrix-its-transpose-its-conjugate-transpose>, <http://www.netlib.org/lapack/explore-html/d7/d15/group__double__blas__level2_ga0dc187c15a47772440defe879d034888.html>

Annexe :

-Dêpot github: <https://github.com/Ennaox/CN-TP4-5>