

**CALCUL DES ELEMENTS
PROPRES D'UNE MATRICE
SYMETRIQUE AVEC OPENMP****RAPPORT DU PROJET**

Préparé par :
Hammoumi Rayane
Giannico Raffaele
Romdhani Hadil

Encadré par :
France Boillod-Cerneux

Remerciements

Tout d'abord, nous tenons à témoigner toute notre reconnaissance à Mme France Boillod-Cerneux pour son soutien dans la concrétisation de ce projet.

Table de Matières

04

Introduction

05

Objectif & librairie

06

Etape 1:

07

Etape 2:

08

Etape 3

09

Etape 4

10

Etape 5: Parallélisation

12

Etape 6: Performance

15

Conclusion

15

Webographie

INTRODUCTION

L'étude des valeurs propres et des vecteurs propres d'une matrice symétrique est importante dans de nombreux domaines, tels que la mécanique des structures, la théorie des systèmes, la géologie et l'analyse numérique.

Cependant, pour les matrices de grande taille, il peut être difficile de calculer exactement les valeurs propres et les vecteurs propres au vu du nombre de calculs à faire.

C'est là qu'intervient la méthode Padé-Rayleigh-Ritz (PRR), qui permet d'obtenir rapidement une approximation d'un petit nombre d'éléments propres.

La méthode PRR est une technique efficace pour résoudre des problèmes complexes dans ces domaines, en offrant une solution rapide pour les grandes matrices symétriques.

Objectifs du Projet

- Réussir à implémenter l'algorithme PRR pour déterminer les éléments propres de grandes matrices

Algorithme PRR (entrées : $y_0 = x / \|x\|$, m sorties : $\lambda_1, \dots, \lambda_m, q_1, \dots, q_m$)

1. Calculer $C_{2k-1} = (y_k, y_{k-1})$, $C_{2k} = (y_k, y_k)$ où $y_k = Ay_{k-1}$, for $k = 1, m$.
Constituer les matrices B_{m-1} , B_m et V_m .
2. Calculer $E_m = B_{m-1}^{-1}$. Constituer $F_m = E_m \times B_m$.
Calculer les valeurs et vecteurs propres de F_m : (λ_i, u_i) pour $i = 1, \dots, m$.
3. Calculer $q_i = V_m \times u_i$ pour $i = 1, \dots, m$
4. Si $\max_{i=1,m} \|(Aq_i - \lambda_i q_i)\| \leq \varepsilon$, alors avec un nouveau vecteur x aller à l'étape 1.

- Réussir à paralléliser notre code avec OpenMP pour améliorer les performances

Librairie GSL

La bibliothèque scientifique GNU (GSL) est une bibliothèque numérique pour les programmeurs en C et en C++. Elle fournit une large gamme de routines mathématiques et statistiques telles que l'optimisation, l'algèbre linéaire et les fonctions spéciales. Elle est conçue pour être portable et efficiente et est utilisée dans de nombreuses applications scientifiques et d'ingénierie.

Les structures de donnée de matrice et de vecteur utilisées par GSL sont:

```
typedef struct
{
    size_t size1;
    size_t size2;
    size_t tda;
    double * data;
    gsl_block * block;
    int owner;
} gsl_matrix;
```

```
typedef struct
{
    size_t size;
    size_t stride;
    double * data;
    gsl_block * block;
    int owner;
} gsl_vector;
```

Phase de projection

Elle consiste à projeter le problème d'une matrice A de taille n dans un sous-espace de taille m beaucoup plus petit que n.

Concrètement, il s'agit de calculer les produits matrice vecteurs A^*y_k afin de constituer les éléments des matrices B_m et B_{m-1} de taille $m*m$. On stocke les y_k dans une matrice V_m .

Voici la fonction:

```
void projection(gsl_spmatrix *A, gsl_matrix *B0, gsl_matrix *B1,  
gsl_matrix *Vm, gsl_vector *yk, size_t taille_sous_espace)
```

NB: y_k est égal à au vecteur initial x divisé par sa norme

Afin de réussir le fonctionnement de cette phase, on a crée plusieurs fonctions auxiliaires

- **gsl_spmatrix *lit_fichier_mat(char nomFichier[]):**

Lit le fichier spécifié et stocke les valeurs dans une matrice de type `gsl_spmatrix`. On fait appel à la fonction de `gsl`: **gsl_spmatrix_fscanf** qui à partir d'un fichier crée un élément de type `gsl_spmatrix`.

- **double produit_scalaire(gsl_vector *yk, gsl_vector *yk_suivant):**

Cette fonction calcule le produit scalaire de deux vecteurs, y_k et y_k_{suivant} , représentés par `gsl_vector *`. La valeur finale est renvoyée comme un double.

- **produit_spmatrice_vecteur(gsl_spmatrix *m, gsl_vector *v,
gsl_vector *resultat):**

Cette fonction calcule le produit matrice-vecteur d'une matrice creuse m et d'un vecteur v , représentés par `gsl_spmatrix` et `gsl_vector` respectivement. Le résultat est stocké dans un vecteur `resultat`.

Étape 2

Résolution du problème dans le sous-espace

Cette étape consiste à calculer l'inverse de B_{m-1} et à le multiplier par B_m . Voici les fonctions utilisées:

- **`gsl_matrix *inverse_matrix(gsl_matrix *A):`**

Cette fonction calcule l'inverse d'une matrice en utilisant la décomposition LU de la bibliothèque GSL. Elle utilise ensuite la fonction `gsl_linalg_LU_decomp` pour effectuer la décomposition LU de la matrice d'entrée et la fonction `gsl_linalg_LU_invert` pour calculer son inverse.

Grâce à cette fonction nous pouvons calculer la matrice E_m

- **`gsl_matrix *multiplie_matrices(gsl_matrix *matrice1, gsl_matrix *matrice2):`**

Cette fonction multiplie deux matrices en utilisant une implémentation naïve du produit matriciel.

Nous pouvons maintenant calculer la matrice F_m .

- **`void calcule_valeurs_et_vecteurs_propre(gsl_matrix *matrix, gsl_vector *valeurs_propres, gsl_matrix *vecteurs_propres):`**

Cette fonction calcule les valeurs propres et les vecteurs propres d'une matrice donnée .

Elle calcule les valeurs propres et les vecteurs propres en utilisant la fonction `gsl_eigen_nonsymmv`.

Ensuite, la fonction affiche les valeurs propres et les vecteurs propres à l'aide d'une boucle for. Les valeurs propres complexes sont converties en valeurs propres réelles et stockées dans un vecteur de valeurs propres. Les matrices de vecteurs propres complexes sont converties en matrices de vecteurs propres réels et stockées dans une matrice de vecteurs propres.

Ces éléments propres nous serviront dans l'étape 3...

Retour dans l'espace de départ

Le but de cette étape est d'effectuer le calcul suivant:

$$\mathbf{q}_i = \mathbf{V}_m \times \mathbf{u}_i$$

où **qi** correspond au produit de la matrice vecteur. Nous avons fait le choix de stocker ce produit dans une `gsl_matrix`. Chaque colonne contiendra un résultat de ce produit.

Vm la matrice

ui le vecteur qui contient les vecteurs propres d'une valeur propre *i* de la matrice F_m .

Pour réaliser cette étape nous avons créer une fonction `calcule_qi`.

```
void calcule_qi(gsl_spmatrix *A, gsl_matrix *qi, gsl_matrix  
*vecteurs_propres, gsl_matrix *Vm, size_t taille_sous_espace)
```

Dans cette fonction, nous avons besoin de la **matrice Vm** et des **vecteurs propres**. Puisque nous allons effectuer le produit de ceux-ci. Cette boucle va nous permettre d'itérer sur la matrice de vecteurs propres et récupérer une colonne qui sera notre vecteur à multiplier.

Ce qui nous permettra ensuite d'appeler la fonction **produit_matrice_vecteur**. Elle sert à multiplier la matrice **Vm** ainsi que ce vecteur.

```
void produit_matrice_vecteur(gsl_matrix *m, gsl_vector *v,  
gsl_vector *resultat)
```

Elle prend en paramètre la matrice, et le vecteur à multiplié. Ainsi qu'un vecteur résultat qui contiendra le produit de la matrice vecteur. Dans cette fonction, il y a une double boucle qui nous permettra d'itérer sur la matrice et le vecteur. Et donc de multiplier chaque ligne de la matrice par le vecteur.

Le déroulement du programme

Dans la fonction main, le programme se lance seulement si l'on a donné en paramètre toutes les informations demandées.

Il faut en effet préciser:

- le fichier de la matrice (au format Matrix Market .mtx)
- la taille m du sous-espace
- la précision

Ensuite, on lit la matrice demandée en paramètre et on la stocke dans un type `gsl_matrix`.

Aussi, nous initialisons toutes les variables dont nous allons avoir besoin dans la suite.

La boucle de l'algorithme est un `while` qui s'arrête lorsque le booleen `precision_atteinte` est à `true`.

C'est dans cette boucle que nous allons exécuter l'algorithme PRR.

Nous allons donc appeler nos fonction pour:

- la projection,
- l'inversion,
- les produits des matrices
- et pour le calcul des éléments propres.

A chaque itération, on vérifie si la précision est atteinte grâce à la fonction `verifie_si_precision_atteinte()`. C'est à ce moment que nous allons changer ou pas le booleen. Si la précision n'est pas atteinte, le vecteur y_k devient égal au dernier vecteur propre calculé divisé par sa norme et la boucle continue.

Enfin, nous libérons la mémoire de toutes les matrices et vecteurs utilisés.

Parallélisation

Les parties les plus chers de cet algorithme sont les étapes 1 et 3. L'effort de la parallélisation devra se faire sur ces deux étapes.

Pour cela, on a décidé de paralléliser les fonctions de bases :

1) Produit matrice-vecteur:

Fonctions :

```
void produit_spmatrice_vecteur(gsl_spmatrix *m, gsl_vector *v, gsl_vector *resultat)
```

et

```
void produit_matrice_vecteur(gsl_matrix *m, gsl_vector *v, gsl_vector *resultat)
```

La parallélisation de ce code se trouve dans la boucle for qui calcule le produit matrice-vecteur. Elle est marquée avec la directive:

```
#pragma omp parallel for schedule(static),
```

indiquant qu'elle sera exécutée de manière parallèle en utilisant le mécanisme de parallélisation OpenMP.

La clause "schedule(static)" signifie que les itérations seront divisées équitablement entre les threads vu que les variables sont indépendantes.

Nous avons choisis d'utiliser static par rapport au autres directive car elle peut donner des résultats plus rapides pour les boucles qui ont un nombre de tours connu à l'avance.

De plus, en testant avec les autres directives comme dynamic ou guided nous pouvions observer une dégradation du temps d'exécution.

2) Produit scalaire :

Fonction :

```
double produit_scalaire(gsl_vector *yk, gsl_vector *yk_suivant)
```

La parallélisation dans ce code est réalisée en utilisant l'instruction "parallel for reduction".

Cela permet d'exécuter la boucle "for" en parallèle sur plusieurs threads et de réduire les résultats intermédiaires des threads en utilisant une opération de réduction : (*"reduction (+:res)"*).

Cela signifie que chaque thread effectue une partie de la boucle "for" et que le résultat de cette partie est combiné avec le résultat total en utilisant l'opération d'addition ("+").

Le résultat final est stocké dans la variable "res".

Étape 5

3) Exécution totale :

Fonctions:

- **void soustrait_vecteur2_au_vecteur1(gsl_vector *vecteur1, gsl_vector *vecteur2, gsl_vector *resultat)**
- **void produit_constante_vecteur(double constante, gsl_vector *vecteur, gsl_vector *resultat)**
-
- **double calcule_norme(gsl_vector *vecteur)**

Dans ces fonctions, la parallélisation est implémentée en utilisant l'instruction OpenMP :

#pragma omp parallel for

pour paralléliser les boucles for. Cela permet à chaque itération de la boucle d'être exécutée par un thread différent, ce qui accélère le temps d'exécution.

Les fonctions `soustrait_vecteur2_au_vecteur1`, `produit_constante_vecteur` et `verifie_si_precision_atteinte` peuvent être parallélisées, car chaque itération est indépendante les unes des autres et peut donc être exécutée de manière concurrente sans interférer avec les autres itérations.

Lors de l'exécution, OpenMP gère les threads et effectue la répartition des tâches entre eux. La réduction `reduction(+:res)` dans la fonction `produit_scalaire` indique à OpenMP que le résultat final doit être accumulé en utilisant l'opérateur d'addition.

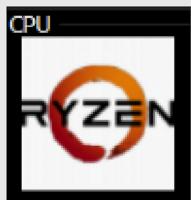
Étape 6

Etude de performance

On a essayé de faire le plus possible de tests pour évaluer la performance de notre code. Nous avons effectuer les test en utilisant le terminal **WSL(Windows Subsystem Linux)**.

C'est une fonctionnalité de Windows 10 qui permet d'exécuter des applications Linux nativement sur un système Windows sans utiliser de machine virtuelle ou de système d'exploitation parallèle. WSL fournit également un environnement de développement Linux intégré à Windows, ce qui permet aux développeurs de travailler sur des projets Linux sans quitter leur environnement Windows.

Voici les caractéristiques de notre bécane:



**AMD Ryzen 5 5600G (6 coeurs, 12 threads,
4,4GHz de fréquence en boost)**



16Go RAM DDR4 3200MHz



SSD NVME PCIE Gen3

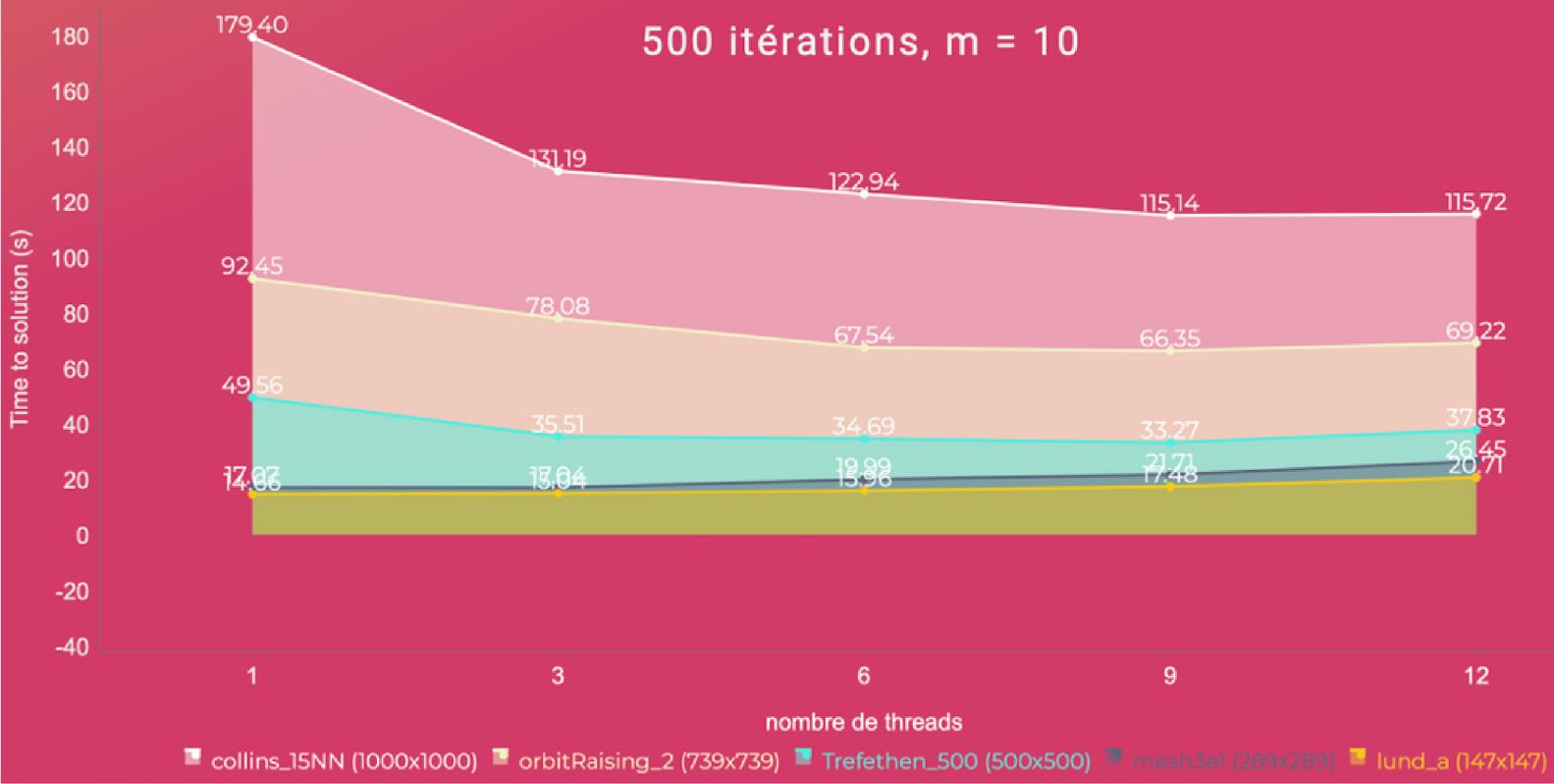
Afin d'évaluer le temps d'exécution, nous avons utilisé **omp_get_wtime()**. C'est une fonction OpenMP qui retourne le temps écoulé en secondes depuis le début de l'application jusqu'au moment où la fonction est appelée.

Donc nous avons mis un `omp_get_wtime()` juste avant la boucle `while` et un juste après

Et en faisant la soustraction des 2 valeurs obtenue nous avons le temps d'exécution de la boucle.

Résultats (parallélisation)

TTS selon le nombre de threads



Observations:

- 147x147 et 289x289: le TTS augmente très légèrement à chaque fois qu'on augmente le nombre de threads
- 500x500: le TTS diminue significativement (-14s) lorsqu'on passe de 1 à 3 threads puis stagne 3 à 9 threads (-1,42s) et augmente (+4,6s) de 9 à 12 threads
- 739x739: le TTS diminue significativement de 1 à 6 threads (-24,91s), puis stagne 6 à 9 threads pour enfin augmenter de 9 à 12 threads (+3s)
- 1000x1000: grande diminution du TTS (- 64,26s !) de 1 à 9 threads, puis stagne lorsqu'on passe de 9 à 12 threads

On remarque que plus le nombre de calculs à effectuer est grand, plus il est efficace d'augmenter le nombre de threads.

Cependant, s'il y a peu de calculs à effectuer, alors il est contreproductif d'augmenter le nombre de threads. On peut supposer que cela engendre des threads qui restent idle.

En résumé, il y a un nombre de threads optimal différent à chaque taille de matrice dans le contexte de l'exécution de notre programme.

Étape 6

Résultats (valeurs propres)

Mise en application: exemple d'un calcul de 10 valeurs propres + vecteurs propres associés: pour la matrice Trefethen (500x500):

```
----[Derniere iteration]----  
[Projection] Temps d'execution : 37.682430 ms  
[Produit matrice-vecteur] Temps d'execution : 0.210390 ms  
  
taille sous espace: 10  
  
Precision: 1e-05  
Nombre de Threads: 2  
Nombre d'itérations: 246  
[Temps d'execution TOTAL] = 17.469782 s
```

```
Valeur propre = 19.3699  
Vecteur propre associé :  
0.999248  
0.0290334  
0.000168691  
-3.30755e-05  
-2.12882e-06  
5.88233e-08  
-3.71024e-10  
1.82376e-13  
-7.68197e-17  
9.39762e-35  
  
Valeur propre = -21.6816  
Vecteur propre associé :  
-0.998961  
0.0455287  
-0.00209977  
9.68454e-05  
-4.46977e-06  
7.60394e-08  
-4.17937e-10  
2.0503e-13  
-8.54571e-17  
1.0526e-34  
  
Valeur propre = -8.89593  
Vecteur propre associé :  
-0.946281  
0.0289236  
0.000840565  
-8.56481e-05  
1.33078e-06  
-2.94116e-09  
-3.15655e-11  
1.58292e-14  
-7.41103e-18  
8.48302e-36
```

```
Valeur propre = -50.0937  
Vecteur propre associé :  
1  
-0.000105697  
1.19598e-08  
-6.30502e-12  
2.51411e-15  
-1.92524e-19  
6.46707e-22  
-3.54452e-25  
1.62469e-28  
-2.33435e-34  
  
Valeur propre = 103.856  
Vecteur propre associé :  
-0.999967  
-0.00809989  
0.000729065  
3.8266e-06  
1.06355e-05  
-2.55408e-07  
1.57802e-09  
-7.75412e-13  
3.26155e-16  
-5.0085e-34
```

```
Valeur propre = 1102.56  
Vecteur propre associé :  
-0.407439  
0.000293899  
-8.1331e-06  
9.86126e-09  
-1.1124e-07  
2.64842e-09  
-1.60053e-11  
4.8876e-15  
-4.52624e-18  
-9.94085e-33
```

NB: les 4 vecteurs manquant sont tous égaux à un des vecteurs déjà affichés

CONCLUSION

Avec la méthode PRR, nous avons réussi à calculer les valeurs et vecteurs propres correspondants d'une matrice A de grande taille n.

Cette méthodes peut être utilisée dans différents domaines afin de résoudre des problèmes similaires.

Ainsi que la parallélisation, nous a permis de distinguer la différence de performances entre différents processeurs et l'impact des petits changements telles que le nombre de threads et la tailles des matrices sur le temps d'exécution.

Ce projet a représenté une excellente opportunité pour appliquer nos connaissances théorique apprises dans l'UE de programmation parallèle et distribuée sur un modèle pratique .

WEBOGRAPHIE

<https://matrixmarket.xyz/home>

<https://stackoverflow.com/>

<https://www.gnu.org/software/gsl/doc/html/>