Recherche informatique de matrices de Butson

Arnaud Mallen et Vincent Popie, sous la tutelle de **Jean-Marc Schlenker**

Semestre 2, 2010/2011

Table des matières

Introduction

Le but de ce projet est de trouver des matrices de Hadamard complexes (matrices de Butson) pour un maximum de couples (n, l) avec n la taille de la matrice et l la puissance des racines de l'unité dans la matrice.

Tandis que deux élèves de l'Université Paul Sabatier, Antoine Campi et Sofiane Saadane, étaient en charge de la partie théorique, nous avons mis nos compétences d'élèves ingénieurs en modélisation mathématique pour implémenter des programmes en **C**++ permettant

Dans un premier temps nous allons situer dans quel contexte théorique nous nous situions et énoncer les propriétés et théorèmes sur lesquels nous nous sommes appuyés. Les démonstrations des théorèmes que nous citons ne seront pas explicitées car leur paternité ne nous revient pas. Toutefois elles peuvent être trouvées dans [2]. Nous allons ensuite présenter un programme traitant le cas particulier des matrices de taille $N \times l$ avec l premier. Enfin un programme capable de traiter le cas général Les résultats obtenus seront exposés.

Chapitre 1

Contexte Mathématique

1.1 Matrices d'Hadamard

Les matrices de Hadamard réelles sont des matrices carrées dont tous les coefficients sont des 1 ou des -1 (multipliées par $\frac{1}{\sqrt{n}}$).

La principale propriété de ces matrices est qu'elles sont orthogonale, c'est à dire que le produit scalaire de deux lignes i et j est égal à $\delta_{i,j}$.

Theorem 1.1. *S'il existe une matrice d'Hadamard de taille n, $n = 2$ ou n divise 4.*

1.2 Matrices de Butson

Definition 1.2. On appelle matrice de Butson les matrices de Hadamard complexe dont les coefficients sont de la forme : $\frac{\omega}{\sqrt{n}}$, où ω est une racine de l'unité. On notera $H_n(l)$ l'ensemble des matrices de Butson de taille n contenant des racines $l-$ièmes de l'unité

Deux matrices H_1 et H_2 sont équivalentes s'il existe deux matrices diagonales et unitaires D_1 , D_2 et deux matrices de permutation P_1 et P_2 telles que

$$H_1 = D_1 P_1 H_2 P_2 D_2$$

Ce résultat signifie que l'on peut multiplier chaque ligne par un complexe de module un ou permuter à la fois les lignes et les colonnes d'une matrice de Butson sans perdre la propriété d'orthogonalité. Le fait de trouver une matrice de Butson implique donc qu'on trouve toute une classe d'équivalence de matrices. Cette donnée est cruciale lors de la construction des algorithmes de recherche de matrices. En effet, le nombre d'éléments d'une classe d'équivalence est très grand pour les tailles de matrices que nous allons considérer, il est donc nécessaire d'instaurer une relation d'ordre au sein de ces classes d'équivalence et de ne rechercher que l'élément le plus petit (ou le plus grand).

Nous avons donc utilisé l'ordre lexicographique sur les lignes et les colonnes de la matrice.

1.2.1 Notation additive

Pour simplifier les écritures des matrices, on désignera par l'indice k la racine de l'unité : $e^{\frac{2i\pi k}{l}}$ pour $k \in [0; l-1]$

1.3 Théorèmes d'obstruction

Theorem 1.3. *Si $H_n(l) \neq \emptyset$ et $l = p_1...p_s$ alors n appartient à $p_1\mathbb{N} + ... + p_s\mathbb{N}$*

Definition 1.4. Un p-cycle est la somme de toutes les racines p-ième de l'unité modulo multiplication par un nombre complexe de module 1 .

Une matrice de Butson est dite régulière si tous les produits scalaires entre deux lignes de la matrices peuvent être décomposés en cycles.

Theorem 1.5 (de De Launey)**.** *Si $H_n(l) \neq \emptyset$ alors il existe d dans $\mathbb{Z}[e^{2i\pi/l}]$ tel que $\mid d \mid^2 = n^n$.*

Theorem 1.6 (de Haagerup)**.** *1. Si $H_2(l) \neq \emptyset$ alors 2 divise l.*

2. Si $H_3(l) \neq \emptyset$ alors 3 divise l.

3. Si $H_4(l) \neq \emptyset$ alors 4 divise l.

4. Si $H_5(l) \neq \emptyset$ alors 5 divise l.

Theorem 1.7. *Supposons $H_n(l) \neq \emptyset$. 1) si $n = p+2$ avec p premier $\geqslant 3$ alors $l \neq 2p^b$ où $b \in \mathbb{N}$. 2) si $n = 2q$ avec $p > q \geqslant 3$ premiers alors $l \neq 2^a p^b$ où a,b sont des entiers naturels.*

1.4 Tableau synthèse

Voici le tableau obtenu par nos collègues de M1 Sofiane Saadane et Antoine Campi à partir des résultat précédent et de leurs recherches personnelles :

n\ l	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	F_2	ll	F_2	ll	F_2	ll	F_2	ll	F_2	ll	F_2	ll	F_2	ll	F_2	ll	F_2	ll	F_2
3	ll	F_3	ll	ll	F_3	ll	ll	F_3	ll	ll	F_3	ll	ll	F_3	ll	ll	F_3	ll	ll
4	F_{22}	ll	F_{22}	ll	F_{22}	ll	F_{22}	ll	F_{22}	ll	F_{22}	ll	F_{22}	ll	F_{22}	ll	F_{22}	ll	F_{22}
5	ll	ll	ll	F_5	ll	ll	ll	ll	F_5	ll	ll	ll	ll	F_5	ll	ll	ll	ll	F_5
6	s_1	T	H	ll	T	ll	H	T	s_3	ll	T	ll	s_3	T	H	ll	T	ll	H
7	ll	ll	ll	ll	P	F_7	ll	ll	s_2	ll	P	ll	F_7	ll	ll	ll	P	ll	?
8	F_{222}	ll	F_{222}	ll	F_{222}	ll	F_{222}	ll	F_{222}	ll	F_{222}	ll	F_{222}	?	F_{222}	ll	F_{222}	ll	F_{222}
9	ll	F_{33}	ll	ll	F_{33}	ll	ll	F_{33}	X_9^{10}	ll	F_{33}	ll	ll	F_{33}	ll	ll	F_{33}	ll	X_9^{10}
10	s_1	ll	X_{10}^4	X_{10}^5	X_{10}^6	ll	X_{10}^4	ll	X_{10}^5	ll	X_{10}^4	ll	s_3	X_{10}^5	X_{10}^4	ll	X_{10}^6	ll	X_{10}^4
11	ll	ll	ll	ll	dl	ll	ll	ll	?	F_{11}	?	ll	?	?	ll	ll	?	ll	?
12	X_{12}^2	X_{12}^3	X_{12}^2	ll	X_{12}^2	ll	X_{12}^2	X_{12}^3	X_{12}^2	ll	X_{12}^2	ll	X_{12}^2	X_{12}^3	X_{12}^2	ll	X_{12}^2	ll	X_{12}^2
13	ll	ll	ll	ll	?	ll	ll	ll	?	ll	ll	F_{13}	?	?	ll	ll	?	ll	?
14	s_1	ll	s_3	ll	?	X_{14}^7	s_3	ll	?	ll	?	ll	X_{14}^7	?	s_3	ll	?	ll	?
15	ll	dl	ll	dl	dl	ll	ll	dl	dl	ll	dl	ll	dl	F_{15}	ll	ll	dl	ll	dl

Légende n : taille de la matrice
l : matrice contenant des racines l-ièmes de l'unité
ll : obstruction de Lam and Lung
$s_{1,2,3}$: obstructions de Sylvester
dl : obstruction de De Launey
? : pas d'obstruction mais pas de matrice connue

Les matrices citées dans ce tableau ne sont pas explicitées dans ce rapport mais sont disponibles dans le mémoire d'Antoine Campi et

Chapitre 2

Premier Programme

Le premier programme que nous avons implémenté est simplement une adaptation d'un programme précédemment réalisé par Jean Marc Schlenker.

2.1 Présentation

2.1.1 Complétion de la matrice

Le point de départ de ce premier algorithme de recherche de matrice de Butson est la propriété de régularité. On considère le cas des matrices de taille $n = N \times l$ avec l premier. On a les propriétés suivantes :

- La première ligne des matrices est remplie de zéros
- Chaque ligne contient N $l-$cycles

La première propriété découle du caractère minimal des matrices. Si on obtient une matrice dont la première ligne ne contient pas de zéros, on peut

La condition de minimalité assure que la première ligne de la matrice est remplie de zéros.. De plus, nous recherchons des matrices régulières, dont le produit scalaire de deux lignes se décompose en une somme de $l-$cycles.

Il ne sera donc pas possible pour ce programme de trouver des matrices de Butson non régulières. Son but est de compléter une matrice récursivement case par case, en insérant chaque valeur possible dans chaque case. Cette complétion est bien sur effectuée de manière intelligente car pour une matrice de taille N contenant des racines $l-$ièmes de l'unité il y a l^{N^2} matrices carrées différentes. En ne considérant que les matrices vérifiant les propriétés énoncées précédemment et la propriété de minimalité, le nombre de cas considérés est considérablement plus faible.

Voici une version simplifiée de l'algorithme utilisé :

Algorithm 2.1 *Complétion d'une matrice de Hadamard*

```
procedure compléter(Matrice M, integer LigneI, integer ColonneJ)
MM=copie de M;
SI position==premièreColonne ALORS

    % Minimale teste si la matrice M est minimale selon l'ordre lexicographique
    SI minimale(MM) ALORS
        Insérer un zéro dans la Colonne 1;
        compléter(MM,LigneI,1);
    FIN SI
SINON
    % valeurMinimum permet d'eviter de tester des valeurs incohérentes
    min=valeurMinimum(MM);
    POUR valeur DE min A l-1 FAIRE
        % On calcule les produits scalaires avec les lignes précédentes
        produitsScalaires=calculerProduitsScalaires(MM,valeur);
        SI produitsScalaires==deltaDeKronecker ALORS
            MM(i,j)=valeur;
            SI MM est pleine ALORS
              SI minimale(MM) ALORS
                afficher(MM);
              FIN SI
            SINON SI j==N*l-1
              % En bout de ligne on passe à la ligne suivante
              compléter(MM,Lignei+1,0);
            SINON
              % Au milieu d'une ligne on passe à la colonne suivante
              compléter(MM,Lignei,Colonnej+1);
            FIN SI
        FIN SI
    FIN POUR
FIN SI
FIN
```

La version de l'algorithme présentée ci-dessus est quelque peu différente de la version réelle implémentée en **C++**. Par souci de rapidité, les produits scalaires sont calculés termes à termes. Les occurences de chaque résultat sont comptées et lorsqu'un résultat identique apparait plus de l fois on est certain que la matrice n'est pas régulière, on peut alors considérer la valeur insérée erronnée. Par construction récursive, toutes les valeurs sont testées.

2.1.2 Test de minimalité

Pour tester la minimalité de la matrice nous avons utilisé l'algorithme suivant :

Algorithm 2.2 *Test de la minimalité d'une matrice M*

```
function minimale(Matrice M)
    résultat=true;
    compteur=0;
    POUR i DE 0 A nombreLignesM FAIRE
        matTemp=M;
        LigneI=i-ème ligne de M;
        % On échange la i-ème ligne de matTemp avec sa dernière
        echangerLignes(matTemp,i,end);
        OrdonnerColonnes(matTemp);
        SI  i-ème ligne de matTemp < LigneI ALORS
            résultat=false;
            break;
        SINON
            SI i-ème ligne de matTemp==LigneI ALORS

              SI i-ème ligne de matTemp > dernière ligne de matTmp ALORS
                résultat=false;
                break;
              FIN SI
            FIN SI
        FIN SI
    FIN POUR
    RETOURNER résultat;
FIN
```

2.2 Adaptation en C++

Le choix du langage est une décision difficile lorsqu'il s'agit d'obtenir des programmes rapides. Nous avons fait le choix du **C++** pour plusieurs raisons. Tout d'abord c'est un langage très rapide, que nous avons appréhendé récemment. Nous connaissons le **C**, réputé pour être plus rapide, mais l'implémentation aurait été beaucoup plus complexe et moins modulable. En effet, nous nous sommes beaucoup réposés sur la librairie standard du **C++,** qui fournit une grande quantité d'algorithmes et de mécanismes performants non disponibles en **C**, langage qui en outre n'est pas orienté objet.

2.2.1 Améliorations de l'algorithme

Notre premier travail a été de comprendre l'algorithme proposé en python et d'y apporter quelques améliorations. Ces améliorations ont été mineures et ont surtout consisté à rajouter des commentaires au programme pour le rendre plus facilement compréhensible. Deux boucles **for** renvoyant un booléen ont été remplacées par des boucles **while** qui s'arrêtent lorsque le booléen change de valeur et évitent ainsi des calculs inutiles.

2.2.2 La classe Matrice

Le programme que nous devions adapter est un programme codé en **Python**, langage orienté objet de plus haut niveau que le **C++** et disposant d'une gestion automatique de la mémoire. Il fait usage des listes et un

certain nombre de mécanismes n'existant pas en **C++**. Pour pallier ces problèmes, nous avons recherché une classe de matrices déjà existante sur internet et possédant toutes les opérations nécessaires à l'implémentation de ce programme. Cette recherche a été infructueuse, la majeure partie des classes de matrices proposant une grande quantité d'opérations ayant attrait au calcul numérique, optimisées pour des matrices de grande taille.

Nous avons donc décidé d'implémenter notre propre classe de manipulation de matrices. Cette classe, disponible en annexe, propose entre autres :

- la construction de matrice carrée, rectangulaire, avec des valeurs entières par défaut ou à partir d'un tableau d'entiers
- le produit scalaire, la transposition
- l'ajout de lignes, de colonnes
- la lecture et écriture dans un fichier
- la comparaison entre matrices
- le test de minimalité selon l'ordre lexicographique
- l'utilisation d'itérateurs pour parcourir la matrice ligne par ligne (dans sa version non améliorée)
- les mécanismes de base de la programmation orientée objet (destructeurs, constructeur de copie, getters et setters

2.2.3 Savoir bien programmer

Lors du débuggage de notre programme, nous avons buté pendant un mois sur des erreurs mémoire. Les messages obtenus nous indiquaient que certaines variables étaient libérées de la mémoire plus d'une fois. Nous avons cherché en vain l'origine de ces erreurs, créé nos propres constructeurs de copie et géré autant que possible les manipulations de la mémoire qui s'effectuent lors de la création et libération des variables concernées. Toutes les fonctions de la classe **Matrice** disposant d'un programme de test concluant, nous ne savions plus quoi faire et avons demandé l'aide d'Arnaud Chéritat par l'intermédiaire de M. Schlenker, chercheur à l'UPS et possédant une bonne expérience du **C++**.

A l'aide d'une commande spécifique de compilation, M. Chéritat a détecté la source de cette insidieuse erreur. La fonction **diffModuloL**, calculant la différence entre deux entiers, renvoyait des entiers négatifs. Ces entiers étant passés en indices d'une matrice, notre programme écrivait dans des mauvais endroits de la mémoire.

Cette erreur aurait été détectée facilement si l'indice négatif avait été passé en argument d'un objet de type **vector< vector <int> >**. Or ces indices étaient passés en argument d'un objet de type Matrice, définit par nous-mêmes. La variable de type **vector< vector <int> >** étant encapsulé [1] dans l'objet de type **Matrice** (impératif en programmation orientée objet !), l'erreur ne pouvait pas remonter.

Cela aurait pu être évité si nous avions créé des mécanismes de gestion des erreurs, que nous n'avons malheureusement pas traité lors de nos cours de **C++**. Ces mécanismes sont longs à mettre en place et il est très difficile d'envisager toutes les erreurs possibles et de les traiter correctement.

En outre, nous avons fait l'erreur de ne pas tester la fonction **diffModuloL**, longue d'une seul ligne et ne faisant pas partie de la classe **Matrice**. Cette fonction fait en effet partie du programme principal et ne manipule pas d'objet de type **Matrice**.

2.3 Résultats obtenus

2.3.1 Tableau récapitulatif

Le tableau suivant contient le nombre de matrices régulières obtenues pour l'éxécution de notre programme pour les couples (n, l), $2 \leq n \leq 18$ et $2 \leq l \leq 14$. Les tirets représentent les cas qui peuvent en théorie être

1. Encapsulation : idée de protéger l'information contenue dans un objet et de ne proposer que des méthodes (fonctions) de manipulation de cet objet.

traités par notre programme mais dont le temps d'exécution est trop long. Nous avons en effet fait tourner le programmes pour ces cas là pendant plus de 50 jours sans obtenir de résultats. Pour les couples (n, l) avec $n > 18$ et $l > 14$ le programme nous pensons que le programme n'est plus utilisable car les temps de calculs deviennent astronomiques. Ce résultat n'est pas vraiment étonnant. Si l'on regarde l'algorithme (2.1), on remarque que l'appel récursif de la fonction de complétion est inclu dans une boucle **for** qui est itérée un nombre de fois compris entre 1 et l.

n\l	2	3	4	5	6	7	8	9	10	11	12	13	14
2													
3		1											
4	1												
5				4									
6	0	1											
7						1							
8	1												
9		21											
10	0			20									
11										1			
12	2	32											
13												1	
14	0					42							
15		-		-									
16	74												
17													
18	0	-											

TABLE 2.1 – *Nombre de classes d'équivalence obtenues en fonction de n et l*

2.3.2 Exemple et validation

Le nombre de matrices obtenues à l'aide du programme concordent en tout point avec le tableau étendu de nos collègues de M1. Notre programme ne trouve pas de matrices lorsqu'une obstruction est présente et retrouve les matrices exemples données dans le tableau précédent. Joint à ce rapport vous pourrez trouver un dossier zip contenant toutes les solutions obtenues par notre programme.

Il est important de noter que le programme retrouve les matrices explicitées dans le tableau présent en 1.4 pour les cas correspondants. Par exemple pour le cas $l = 3$ et $n = 6$, on trouve une solution :

$$X_6^3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 0 & 2 & 1 & 2 \\ 0 & 1 & 2 & 0 & 2 & 1 \\ 0 & 2 & 1 & 2 & 0 & 1 \\ 0 & 2 & 2 & 1 & 1 & 0 \end{pmatrix}$$

qui n'est autre que la matrice de Tao ([1, 2])ayant subit une permutation des deux dernières lignes puis une permutation des deux dernières colonnes :

$$T = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 0 & 2 & 2 & 1 \\ 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 2 & 2 & 1 & 0 & 1 \\ 0 & 2 & 1 & 2 & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 0 & 2 & 2 & 1 \\ 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 2 & 1 & 2 & 1 & 0 \\ 0 & 2 & 2 & 1 & 0 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 0 & 2 & 1 & 2 \\ 0 & 1 & 2 & 0 & 2 & 1 \\ 0 & 2 & 1 & 2 & 0 & 1 \\ 0 & 2 & 2 & 1 & 1 & 0 \end{pmatrix}$$

Pour le cas $l = 3$ et $n = 3$ la solution trouvée est

$$X_3^3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 2 & 1 \end{pmatrix} = F_3$$

De même, le programme ne trouve pas de matrice lorsqu'une obstruction est démontrée.

Pour être certains de la validité des solutions, nous avons utilisé un sous-programme testant la propriété d'orthogonalité des matrices et un second sous-programme vérifie que toutes les solutions obtenues sont différentes. Ces sous-programmes ont bien sûr été testés eux-mêmes.

2.4 Amélioration de la structure de données

La structure de données **vector< vector <int> >** présente l'avantage d'être très intuitive à la fois d'un point de vue théorique et d'un point de vue écriture du code. Toutefois, le fait d'imbriquer des vecteurs dans un vecteur coûte cher en temps de calcul, en particulier lors de la création et destruction des variables de ce type. En effet ces deux étapes ne sont pas entièrement sous notre contrôle, de nombreux appels aux constructeurs et destructeurs sont effectués implicitement. Par exemple, lors de la destruction d'un **vector< vector <int> >,** le destructeur de la classe **vector** appele le destructeur du type de données qu'il contient pour effacer séquentiellement chaque élément qu'il contient, c'est à dire un **vector**, qui lui-même appelle le destructeur de la classe **int**.

Example 2.1. Pour une matrice carrée de taille 15, le destructeur de la classe **vector** est ainsi appelé 16 fois, celui de la classe **int** 225 fois.

Nous avons donc décidé de changer de structure de donnée en utilisant simplement un **vector< int >** et d'utiliser en interne un indicage particulier. Etant donné une matrice A de taille $n \times m$, le terme $A_{i,j}$ est situé dans la case indicée $(i-1) \times m + j - 1$, l'indicage d'un tableau en **C++** débutant à l'indice 0.

$$\begin{pmatrix} (1 & 2 & 3) \\ (4 & 5 & 6) \\ (7 & 8 & 9) \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

FIGURE 2.1 – *Modification de la structure de donnée*

Ce changement a été rapide à implémenter étant donné que nous avons utilisé la méthode de programmation orienté objet standard.

Example 2.2. Pour une matrice carrée de taille 15, le destructeur de la classe **vector** est dans ce cas appelé 1 fois, celui de la classe **int** 225. On gagne ainsi 15 appels du destructeur de la classe **vector** à chaque destruction de variable.

Pour appuyer ce résultat nous avons conçu un programme simple construisant puis détruisant un grand nombre de **vector< vector<int> >**, le même nombre de **vector<int>** et affichant les temps d'exécution correspondant. Pour 100000 constructions/destructions, on obtient la sortie console suivante :

```
VectorVector : 701374016
Vector : 101489018
Ratio tempsVectorVector/ tempsVector : 6.91084
```

Le résultat est donc plutôt convaincant. Ce résultat dépend bien sûr de la machine sur laquelle le programme tourne. Sur le serveur **gmmtrans1** du département GMM on obtient un ratio assez variable de l'ordre de 4, ce qui est aussi convaincant.

Il reste toutefois des voies d'amélioration pour notre programme. La principale partie à améliorer est l'implémentation d'itérateurs spécifiques à notre classe de matrice. En **C++**, les itérateurs sont des pointeurs permettant de parcourir des tableaux. Lors du passage de **vector< vector <int> >** à **vector<int>**, nous n'avons pas pu continuer à nous reposer sur les itérateurs créés par défaut par la librairie standard.

En effet, quand l'incrémentation d'un **vector< vector <int> > : :iterator** nous permettait de sauter d'une ligne dans la matrice, l'incrémentation d'un **vector <int> : :iterator** ne fait qu'avancer d'une case dans le vecteur, sans tenir compte de notre indiçage spécifique. Ne sachant pas comment implémenter des itérateurs personnalisés, nous avons donc utilisé une conversion en **vector< vector <int> > : :iterator** a chaque appel de la fonction **minimale**, seule partie de l'algorithme nécessitant ces itérateurs.

Chapitre 3

Deuxième Algorithme

Une fois l'algorithme pour le cas particulier de l premier et de n multiples de l, nous avons donc décidé d'aborder le cas général c'est-à-dire l et n quelconques. Nous voulions néanmoins obtenir des résultats intéressants, donc chercher des matrices pour les cas où il n'y a pas d'obstruction, mais où il n'y a pas encore de matrices connues. Nous avons donc commencé par le cas $l = 15$ et $n = 8$, puisque les lignes sont des combinaisons d'un cinq cycle et d'un trois cycles.

3.1 Présentation

Nous allons dans cette partie présenter l'algorithme pour ce cas précis, sachant qu'une fois implémenté, il n'y a que quelques paramètres à changer dans le programme pour pouvoir exécuter le programme dans un autre cas. La principale idée de notre programme est de rechercher toutes les lignes qu'il est possible de mettre dans la matrice, et de les tester une à une de manière intelligente.

Pour le cas l=15 et n=8, il y a 15 combinaisons différentes, car il y a 3 cinq cycles et 5 trois cycles. Pour chaque combinaison, il y a 8! possibilités de lignes différentes (en changeant l'ordre des éléments). Le nombre de lignes possibles est donc $8! \times 15 = 604800$.

Il est évident que le temps d'exécution du programme aurait été trop long si nous en étions restés là. Tout d'abord, nous avons imposé à chaque ligne de contenir au moins un 0 (en notation indicielle), et de commencer par celui-ci. En effet si on se réfère à la propriété numéro , une ligne ne commence pas par un zéro peut être multipliée par une racine de l'unité pour que cela soit le cas.

Le nombre de lignes possibles est donc de$7 \times 7! = 35280$. Si on étudie les 7 combinaisons possibles contenant un zéro :

0	0	3	5	6	9	10	12
0	1	3	6	6	9	11	12
0	2	3	6	7	9	12	12
0	3	3	6	8	9	12	13
0	3	4	6	9	9	12	14
0	1	4	5	7	10	10	13
0	2	5	5	8	10	11	14

On remarque que chaque ligne contient un élément en double, on peut donc diviser par deux le nombre de lignes sauf pour la première combinaison, puisque le 0 est imposé en première position. On a donc un nombre de lignes égal à

$$nbLignes = 7! + \frac{6 \times 7!}{2} = 20160$$

Sous les conditions de minimalité, on peut imposer à la première ligne de ne contenir que des 0, et à la seconde de contenir des éléments croissants. En effet si elle 'nest pas croissante on peut effectuer des permutations successives sur les colonnes jusqu'à obtenir une suite croissante d'éléments. Pour diminuer

encore le temps d'exécution, nous avons ordonné les 20160 lignes par ordre lexicographique et les avons rassemblé par groupe.

Definition 3.1. Deux lignes appartiennent au même groupe si les 4 premiers éléments de celles-ci sont identiques.

Lorsque l'on veut ajouter une ligne dans la matrice, on vérifie que ses 4 premiers éléments sont compatibles avec ceux des lignes précédentes (en utilisant la propriété ...), s'ils ne le sont pas, on peut donc éliminer toutes les lignes appartenant au même groupe que la ligne testée. S'ils sont compatibles, on teste cette fois la ligne entière jusqu'à en trouver une compatible.

3.2 Implémentation en C++

Nous avons utilisé la classe matrice que nous avions créé précédemment car elle est adaptée à cette résolution. Nous avons trouvé sur internet l'implémentation de l'algorithme qui crée toutes les permutations possible d'un tableau d'entier en s'appuyant sur la librairie standard. Nous avons ensuite créé une fonction permettant de ranger toutes les lignes créées par ordre croissant en éliminant celles ne commençant pas par un 0. Ensuite nous avons implémenté la fonction permettant de créer des groupes, et enfin une fonction qui est appelée récursivement pour compléter la matrice.

3.3 Résultats

Le nombre de lignes créé par notre algorithme est bien égal à $nbLignes$.

3.4 Améliorations / Généralisation

Dans le cas présenté, la taille de la matrice est assez faible, et le fait qu'il n'y ai que deux cycles limite le nombre de possibilité que les groupes appartiennent à une combinaison de cycles possibles. Dans des cas plus généraux, il serait peut-être judicieux d'ajouter les éléments d'une ligne un par un, l'ajout d'un élément non compatible entraînant la suppression du groupe de lignes correspondant.

Les cas possibles à traiter sont :

Conclusion

- Faire face à des problèmes. Compétence d'un matheux en info etc
- Résultats intéressants
- Pas posé le problème de la remise en question de la régularité puisqe pas de propriété intéressante permettant d'avoir des temps de calcul faibles...
- apprendre à utiliser la commande ssh avec screen, utiliser des machines blabla

Bibliography

[1] T. Banica, J. Bichon, J-M Schlenker
Representations of quantum permutation algebras,
Journal of Functional Analysis 257 (2009) 2864-2910

[2] A. Campi, S. Saadane,
Matrices de Hadamard
Université Paul Sabatier

Annexe

Cette annexe regroupe toutes les productions **C++** utilisées pour ce projet sauf les programmes de test (le rapport aurait été trop lourd).

matrice.h

```
/* testMatrice.cpp

Projet sur les matrices de Hadamar

Arnaud Mallen
Vincent Popie
4GMM option MMN

Programme principal
*/

#include <iostream>
#include "matrice.h"
#include "testMatrice.h"
#include <sstream>
#include <mach/mach_time.h>
#include <stdint.h>


int N;
int l;

// Calcule la difference modulo (les racines tournent)
int diffModuloL(int p,int q);
int nombreSolutions=0;
void complete(Matrice M, int i, int j, Matrice L);
string int2str(int i);
void verifierSolutionsDifferentes(int petitl,int grandN, int nombreDeSolutions);


int main ()
{

    TestMatr();

    cout <<"Valeur de l? : ";
  cin>>::l;
  cout <<endl<<"Valeur de n? : ";
  cin>>::N;
  cout<<endl;



  // Cree une matrice-ligne de taille N*l, la premiere ligne contient des zeros, la seconde
      1:l
```

```
45    Matrice MN(1,(::l)*(::N),0);

47
    MN.ajouterLigneZeros();
49    for(int valeurs=0;valeurs<(::l);valeurs++){
        for( int position=0;position<(::N);position++)
51        MN(1,valeurs*::N+position)=valeurs;

53    }

55    Matrice L(2,(::l),0);

57    uint64_t tempsInitial = mach_absolute_time();
      complete(MN,2,0,L);
59    uint64_t tempsVector=mach_absolute_time()-tempsInitial;
      cout<<endl<<"Temps d'execution : "<<tempsVector<<endl;
61    // On verifie la coherence des reponses.
      verifierSolutionsDifferentes(::l, ::N, ::nombreSolutions);
63    return 0;

65 }

67 int diffModuloL(int p,int q){
      int r=p-q;
69    return ((p-q) < 0) ? ::l+r : r;
   }
71
   // Verifie que les solutions sont toutes differentes
73 void verifierSolutionsDifferentes(int petitl,int grandN, int nombreDeSolutions){
      // deux matrices temporaires;
75    Matrice SOL_I(::N*::l);
      Matrice SOL_J(::N*::l);
77    bool differentes=true;
      for(int i=1;i<nombreDeSolutions;i++){
79        for (int j=i+1; j<nombreDeSolutions; j++) {

81          SOL_I.lireFichier("sol_"+int2str(::N)+"_"+int2str(::l)+"_"+int2str(i));
            SOL_J.lireFichier("sol_"+int2str(::N)+"_"+int2str(::l)+"_"+int2str(j));
83          if (SOL_I==SOL_J) {
              cout<<endl<<"Les solutions "<<i<<" et "<<j<<" sont egales!!!";
85            differentes=false;
            }
87
          }
89    }
      if(differentes)
91      cout<<endl<<"Toutes les solutions sont bien differentes";

93 }

95
   // Conversion de int a string
97 string int2str(int i){
      ostringstream oss;
99    // ecrire un nombre dans le flux
      oss << i;
101   // recuperer une chaine de caracteres
        return oss.str();
103 }

105



107
   // Procedure de completion de la matrice de depart (on suppose que les matrices obtenues
       sont regulières pour les trouver...)
109 void complete(Matrice M, int i, int j, Matrice L){
```

```
    int minimum;
111 // K est surdimensionnee
    Matrice K=L;
113 if (j==0){

115   // Si on est sur la premiere colonne, on ajoute une ligne de zeros lorsque la matrice
          est minimale, on
      // initialise L a la bonne taille
117   if(M.minimale()){
        M.ajouterLigneZeros();
119     Matrice TMP(i,::l,0);
        L=TMP;
121
        // On note les zeros
123     for (int lignes=0;lignes<i;lignes++)
          L(lignes,0)=1;
125
        // Appel recursif, on complete la seconde case de la ligne i
127     complete(M,i,1,L);
      }
129 }else {
      // On trouve le minimum, on evite de parcourir trop de branches de l'arbre
131   // En remplissant avec des valeurs intelligentes -> on utilise la regularite et le
          caractere ordonne
      bool egaux=true;
133   for(int ii=0;ii<i;ii++){
        if (M(ii,j-1)!=M(ii,j)){
135       egaux=false;
        }
137   }
      if (egaux){
139     minimum=M(i,j-1);
      }else {
141     minimum=0;
      }
143   // On ne considere que les valeurs possibles
      for (int valeur=minimum;valeur<l;valeur++){
145
        K=L;
147     bool cok=true;
        // on remplit la matrice K
149     for (int k=0;k<i;k++){
          int ind=diffModuloL(valeur,M(k,j));
151       K(k,ind)++;
          if (K(k,ind)==(::N)+1){
153         cok=false;
          }
155     }

157     // Si les valeurs dans K son correctes (regularite) on insere la valeur
        if (cok){
159       M(i,j)=valeur;

161       // En fin de matrice, on affiche le resultat si la matrice est minimale
          if (i==(::N)*(::l)-1 && j==(::N)*(::l)-1){
163         if (M.minimale()){
              cout<<endl<<"On a une solution : "<<endl;
165           M.afficher();
              ::nombreSolutions++;
167           if (M.estOrthogonale(::l)) {
                cout <<endl<< "La solution est bien orthogonale!"<<endl;
169           }
              else {
171             cout<<endl<<"ERREUR, La solution n'est pas orthogonale!!";
              }
173
```

```
            M.ecrireFichier("sol_"+int2str(::N)+"_"+int2str(::l)+"_"+int2str(::
                nombreSolutions));
175         cout<<endl<<endl<<"Sol ecrite"<<endl;
          }
177     }else {

179     // Lorsque l'on n'est pas en bout de matrice, on appelle sur la case suivante

181       if (j==(::N)*(::l)-1) {
            // Bout de ligne, on passe a la ligne suivante
183         complete(M,i+1,0,K);
          }else {
185         // Autre cas : on avance d'une case sur la ligne consideree
            complete(M,i,j+1,K);
187       }

189     }
      }
191   }
  }
193

195 }
```

./main.cpp

matrice.cpp

vectorVsVectorVectorLinux.cpp

```
/* Programme de comparaison des classes vector et vector de vector.
2 Ce programme "construit" un nombre donnÃ© de vecteurs de vecteurs
  et le meme nombre de vecteurs (mÃªme nombre d'Ã©lÃ©ments). Il permet
4                 de comparer les temps d'Ã©xecution associÃ© Ã  chaque type de donnÃ©es

6                 */

8
#include <iostream>
10 #include <vector>
#include <time.h>
12
using namespace std;
14

16 int main (int argc, char * const argv[]) {

18    // Nombre de constructions/Destructions
    const int MAX=100000;
20
    // Taille matrice
22  // Taille matrice
    const int taille=15;
24
    // On initialise au temps machine
26
    clock_t tempsInitial;
28  double tempsVectorVector, tempsVector;
    tempsInitial=clock();
30
    // On crÃ©e et dÃ©truit  MAX fois un vecteur de vecteur int
32
    for (int i=0;i<MAX;i++){
34    vector< vector <int> > vecDouble(taille,vector<int> (taille,0))\
```

```
      ;
36    }
    tempsVectorVector = ((double)clock() - tempsInitial);
38
    // On remet Ã  zÃ©ro
40    tempsInitial = clock();

42    // On crÃ©e et dÃ©truit MAX fois un vecteur d'int
    for (int i=0;i<MAX;i++){
44      vector<int> vecSimple(taille*taille,0);
    }
46
    tempsVector = ((double)clock() - tempsInitial);
48    cout<< endl<<" VectorVector : "<<tempsVectorVector<<endl;
    cout<< endl<<" Vector : "<<tempsVector<<endl;
50    cout<<endl<<"Ratio tempsVectorVector/ tempsVector : "<<tempsVectorVector/tempsVector;

52    return 0;
}
```

./vectorVsVectorVectorLinux.cpp