Algèbre linéaire exacte efficace : le calcul du polynôme caractéristique

Clément Pernet

UNIVERSITÉ JOSEPH FOURIER

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Spécialité : Mathématiques Appliquées

préparée au **Laboratoire de Modélisation et Calcul (L.M.C.)**
dans le cadre de **l'École Doctorale**
«Mathématiques, Sciences et Technologies de l'Information, Informatique»

présentée et soutenue publiquement

par

Clément PERNET

le 27 septembre 2006

Algèbre linéaire exacte efficace : le calcul du polynôme caractéristique

---

Directeurs de thèse : Dominique DUVAL et Jean-Guillaume DUMAS

JURY

M.	Jean DELLA DORA	Président proposé
M.	Luc GIRAUD	Rapporteur
M.	Gilles VILLARD	Rapporteur
Mme.	Dominique DUVAL	Directrice de thèse
M.	Jean-Guillaume DUMAS	Responsable de thèse
M.	Luis Miguel PARDO	Examinateur
M.	Jean-Louis ROCH	Examinateur

REMERCIEMENTS

Je remercie tout d'abord Jean Della Dora pour avoir accepté de présider le jury de cette thèse. Plus généralement, c'est sa vision singulière de la recherche et la qualité des rapports humains qu'il a su insuffler à notre équipe MOSAIC, que je tiens à saluer.

Messieurs Luc Giraud et Gilles Villard ont eu la lourde tâche de rapporter ce manuscrit dans des délais particulièrement réduits. Je les remercie pour l'attention minutieuse qu'ils y ont porté, ainsi que pour leurs nombreuses corrections. Celles-ci ont participé à l'amélioration sensible de ce document.

Je tiens à exprimer ma profonde reconnaissance à Jean-Guillaume Dumas pour presque six années de collaboration au cours desquelles il m'a transmis sa passion et ouvert les portes de la recherche. Son investissement personnel constant à mes cotés a grandement contribué à l'aboutissement de cette thèse. Merci à lui et à Dominique Duval pour leur encadrement irréprochable.

Au cours de cette thèse j'ai eu le privilège de collaborer six mois avec Luis Miguel Pardo à l'université de Cantabrie. Je lui témoigne ma gratitude pour la générosité de son accueil et pour avoir pu prendre part à ce jury.

Je remercie aussi sincèrement Jean-Louis Roch pour nos nombreuses discussions et la passion qu'il a su transmettre à travers ses cours. Sa présence dans ce jury me tient particulièrement à cœur.

Je remercie chaleureusement l'ensemble des membres du projet `LinBox` pour un travail d'équipe de qualité, malgré l'éloignement, et dans une ambiance amicale. Merci tout particulièrement à David Saunders, pour son hospitalité au Delaware, notre fructueuse collaboration durant son séjour invité au laboratoire, et tout simplement pour notre amitié. Je remercie chaleureusement Pascal Giorgi, Claude-Pierre Jeannerod et Gilles Villard pour le plaisir que m'a apporté notre collaboration active dans une ambiance enthousiaste et amicale. Merci aussi à Wayne Eberly, Mark Giesbrecht, Erich Kaltofen et Arne Storjohann pour leur conseils et nos multiples discussions.

Je remercie également Claudine, Cathy et Juanna dont la bonne humeur et l'efficacité mettent l'huile nécessaire dans les rouages de la machine administrative.

Les implémentations présentées dans nos travaux s'appuient sur de nombreuses bibliothèques bénéficiant des licences GPL ou LGPL, sans lesquelles de nombreux résultats n'auraient pu voir le jour. Je remercie la communauté du logiciel libre pour cette formidable initiative à laquelle nous contribuons à notre tour modestement en distribuant les bibliothèques présentées dans ce document sous la licence LGPL.

Une thèse n'est pas une expérience solitaire et je remercie tous les doctorants du laboratoire pour la fraternité qui y règne. Un grand merci à tous ceux qui ont partagé mon quotidien pendant ces trois années, Aude et Claire pour votre présence et votre écoute dans ce fameux

bureau 16, Carlos et Zhendong qui m'ont épaulé au cours de mes séjours espagnols et américains, ainsi que mes compagnons de soirée et compagnons de cordée qui se reconnaîtront.

C'est enfin à ma famille que vont mes derniers remerciements, pour son rôle moteur tant au niveau affectif qu'intellectuel, qu'elle a joué dans l'accomplissement de ce travail.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

Introduction

CONTEXTE ET THÈSE

Calcul exact

Si à sa naissance le calcul scientifique mêlait le calcul approché au calcul exact, son essor à partir des années 1950 reste fortement lié au calcul approché, aussi appelé calcul numérique. En effet ses principales applications provenaient de problèmes physiques où une approximation de la solution par ses premiers chiffres significatifs est souvent suffisante. Les nombres flottants et leur arithmétique ont été introduits pour approcher les nombres réels, et ont rapidement occupé une place privilégiée dans les architectures des ordinateurs. Parallèlement, le calcul exact souffrait de son principal désavantage : obtenir tous les chiffres d'une solution était trop coûteux, et l'intérêt de la communauté scientifique restait limité pour cette approche.

Il a fallu attendre les avancées théoriques des années 1980, comme celle dûes à Dixon ou à Wiedemann pour relancer l'intérêt des méthodes exactes en algèbre linéaire. Parallèlement, de nouvelles applications en mathématiques (topologie algébrique, théorie des graphes, théorie de la représentation, ...) sont apparues et pour lesquelles les résultats approchés des méthodes numériques n'ont aucun sens. Des efforts important ont été fournis pour le développement de bibliothèques efficaces d'arithmétique exacte (PARI, GMP, NTL, ...) d'une part, et d'algorithmes performants en algèbre linéaire d'autre part (en particulier au niveau des complexités binaires lors des dix dernières années).

Il s'imposait donc de s'intéresser à la mise en pratique de ces algorithmes dans le cadre d'une bibliothèque d'algèbre linéaire exacte. C'est l'objectif que s'est donné le projet LINBOX, créé en 1998, avec pour objectif premier, la mise en œuvre des récents algorithmes de type *boîte noire*. Cet objectif a rapidement été élargi au projet d'une bibliothèque générale d'algèbre linéaire exacte. Ce projet servira de cadre pour les développements logiciels effectués dans nos travaux.

Algorithmique en algèbre linéaire exacte

Le domaine privilégié en calcul exact est l'anneau des nombres entiers $\mathbb{Z}$, et éventuellement le corps des nombres rationnels $\mathbb{Q}$. Dans ces domaines, les nombres manipulés ont des tailles variables et les opérations arithmétiques ont donc un coût dépendant des tailles de leurs opérandes. Ainsi la croissance des calculs intermédiaires doit être contrôlée afin de réduire la complexité des algorithmes.

Une alternative consiste à utiliser les corps finis pour effectuer les calculs dans $\mathbb{Z}$ ou $\mathbb{Q}$. En algèbre linéaire, ils peuvent intervenir de deux manières :

– soit par la remontée p-adique de Dixon [26] pour la résolution de systèmes linéaires,

– soit par le théorème des Restes Chinois pour le calcul du déterminant, du polynôme caractéristique, etc

Il s'agit dans les deux cas d'effectuer les opérations dominantes dans des corps finis dont la taille des coefficients reste constante. Les avantages de ces approches sont multiples. Elles permettent tout d'abord d'améliorer d'un ordre de grandeur les complexités de plusieurs algorithmes dans $\mathbb{Z}$. Par ailleurs, elles permettent l'utilisation d'algorithmes définis sur un corps utilisant donc une division toujours définie. Enfin, la taille fixe des coefficients des corps finis est adaptée à l'utilisation des mots machine et de l'arithmétique entière des ordinateurs.

L'algorithmique en algèbre linéaire distingue deux façons de représenter les matrices :

les représentations denses où l'on considère les $n \times m$ coefficients de la matrice et s'autorise toutes les opérations sur ses coefficients.

les représentations *boîte noire* où la matrice est uniquement vue comme une application linéaire. La seule opération possible est l'application à un vecteur. Elle permet en particulier de tirer partie de la structure des matrices.

En algèbre linéaire dense, la multiplication de matrices est à la fois une opération fondamentale et un sujet de recherche toujours très actif, tant au niveau théorique que pratique.

Sur le plan théorique, la révolution qu'a initiée Strassen [108] en introduisant en 1969 un algorithme sous-cubique de multiplication matricielle a conduit à de multiples améliorations de l'exposant dans la complexité du produit matriciel. La question de la meilleure complexité asymptotique pour cette opération reste actuellement encore ouverte. Par ailleurs ces progrès ont rapidement bénéficié aux autres problèmes d'algèbre linéaire au niveau de leur complexité théorique, grâce à des réductions. Cependant au niveau pratique, ces algorithmes ont longtemps été considérés comme inintéressants, car leur amélioration asymptotique ne pouvait prendre effet que pour des dimensions alors inaccessibles. C'est à partir des années 1980, que les progrès technologiques des architectures informatiques permirent d'aborder des dimensions suffisantes pour s'intéresser à la mise en pratique de l'algèbre linéaire rapide. Souffrant de certains problèmes d'instabilité numérique, ces algorithmes rapides n'ont jamais été massivement utilisés. En revanche, le calcul exact, qui ne connaît pas ce problème d'instabilité peut bénéficier sans réserve de ces algorithmes rapides.

Au niveau pratique enfin, la différence de vitesse grandissante entre les unités de mémoire (lentes) et les unités de calcul (rapides) ont conduit à la création d'une structure hiérarchique de la mémoire des architectures modernes. Une utilisation adaptée de cette structure, respectant la localité des données permet à certaines routines de base d'atteindre des performances extrêmes. La multiplication matricielle en calcul flottant fait partie de ces routines.

Ainsi, la multiplication matricielle doit est la brique dans l'élaboration de routines d'algèbre linéaire exacte dense. En nous appuyant sur ces deux aspects, nous développerons une routine efficace de multiplication matricielle sur un corps fini, qui servira ensuite de brique de base pour le développement de routines d'algèbre linéaire exacte dense, menant au calcul du polynôme caractéristique de matrices denses.

Avec les représentations *boîtes noires*, la brique de base en algorithmique est le calcul du polynôme minimal. Cette approche est issue des méthodes itératives en calcul numérique (de type Krylov, Lanczos). Les travaux de Wiedemann [119] ont montré que leur adaptation au

calcul exact était possible, et leur analyse par Kaltofen et Saunders [76] a jeté les bases nécessaires à la création de la bibliothèque LINBOX. C'est donc à nouveau par des réductions à cette brique de base que nous aborderons le calcul du polynôme caractéristique de matrices *boîte noire*.

Enfin, le modèle de calcul probabiliste est fréquemment utilisé dans les algorithmes d'algèbre linéaire exacte. Un résultat exact est calculé avec une probabilité d'erreur bornée. Si en théorie, leur nature diffère de celle des algorithmes déterministes, nous soutenons que leur fiabilité en pratique n'est pas moindre, pourvu que la probabilité d'échec soit réduite à la probabilité d'apparition d'autres phénomènes physiques pouvant engendrer des erreurs lors du calcul dans la machine.

Thèse

Dans ce contexte, nous pouvons formuler la thèse que nous défendons ainsi :

Le calcul exact en algèbre linéaire est à la fois utile et praticable en grandes dimensions. Les récentes avancées théoriques dans la résolution de problèmes classiques, comme le calcul du polynôme caractéristique, peuvent ainsi être confrontées à la mise en pratique par des algorithmes denses ou boîte noire, éventuellement probabilistes, et aboutir à des solutions efficaces en pratique.

A travers ces travaux, nous proposons avant tout une démarche pour l'élaboration d'algorithmes exacts efficaces en pratique. Devant la diversité des algorithmes disponibles, des domaines de calcul ($\mathbb{Z}$, $\mathbb{Z}_p$), des représentations matricielles (dense ou *boîte noire*) et des niveaux d'optimisation (du nombre d'opérations algébriques, à la gestion de la mémoire cache), il convient de structurer la conception des algorithmes.

Les calculs dans $\mathbb{Z}$ sont ramenés à des calculs dans des corps finis par des techniques de Reste Chinois. Les algorithmes dans les corps finis se réduisent ensuite aux briques de base (polynôme minimal pour les matrices *boîte noire* et produit matriciel pour les matrices denses). Lorsque différentes réductions sont envisageables, ayant chacune des domaines de prédilection. Des algorithmes adaptatifs permettent alors de combiner ces algorithmes. Enfin nous pouvons concentrer les efforts d'optimisation sur les briques de base (essentiellement le produit matriciel pour les matrices denses). Nous considérons des optimisations tant au niveau matériel (optimisation de la mémoire cache, unités arithmétiques flottantes) qu'au niveau algorithmique (réductions modulaires différées, algorithmes de multiplication rapide). L'efficacité de ces routines de base, combinées à la pertinence de cette structuration permet d'obtenir des performances expérimentales jusqu'alors inégalées.

La première étape dans ce travail consiste donc à élaborer des routines de base pour l'algèbre linéaire dense sur des petits corps finis. Ainsi nous abordons d'abord la mise au point du noyau de produit matriciel dans le chapitre 2, tirant son efficacité des routines BLAS du calcul numérique et des algorithmes de multiplication rapide. Nous étudions ensuite le problème de l'élimination de Gauss et de la résolution de systèmes linéaires dans le chapitre 3. Enfin nous appliquons ces deux briques de bases à plusieurs autres problèmes d'algèbre linéaire dans le chapitre 4.

Dans une deuxième partie, nous traitons du calcul du polynôme caractéristique de matrices denses. Nous proposons dans le chapître 6 un nouvel algorithme tirant parti de l'efficacité des routines présentées en première partie. Le chapitre 7 traite de l'algorithme par branchements de Keller-Gehrig, dont nous proposons une amélioration ainsi qu'une application à un problème de théorie du contrôle : le calcul de la forme de Kalman. Enfin nous étudions l'algorithme rapide de Keller-Gehrig dans le chapitre 8 : nous mettons en évidence son intérêt pratique et relâchons sa condition de généricité en ouvrant des perspectives de généralisation de son domaine d'application. Ces algorithmes sur des corps finis sont ensuite utilisés dans le chapitre 9 pour le calcul du polynôme caractéristique dans $\mathbb{Z}$. Nous donnons une borne améliorée sur la taille des coefficients, pour un algorithme déterministe, utilisant le théorème des restes chinois. Nous présentons aussi un algorithme probabiliste calculant les mutliplicités des facteurs du polynôme caractéristique. Cette méthode conçue ici pour les matrices denses, sera approfondie par la suite pour les matrices *boîte noire*.

La troisième partie est consacrée aux algorithmes de type *boîte noire*. Nous résumons dans le chapitre 10 les résultats asymptotiques dans le domaine, incluant une perspective d'amélioration. Nous présentons ensuite dans le chapitre 11 différentes techniques de calcul des multiplicités de facteurs irréductibles du polynôme caractéristique dans un corps fini, débouchant sur un nouvel algorithme adaptatif. À nouveau cet algorithme est utilisé dans le chapitre 12 pour le calcul dans $\mathbb{Z}$, grâce cette fois-ci à une remontée p-adique. Il est enfin appliqué à la résolution d'un problème issu de la théorie des graphes.

LA BIBLIOTHÈQUE LINBOX

Le projet LINBOX est une collaboration internationale (France, États-Unis, Canada) pour le développement d'un outil logiciel générique pour résoudre les problèmes classiques en algèbre linéaire exacte : calcul du rang, du déterminant, de formes normales matricielles, ...

Motivation originale : l'algèbre linéaire exacte en boîtes noires

Initialement, ce projet visait particulièrement à rassembler les techniques de type *boîte noire* pour l'algèbre linéaire dans des corps finis ou dans $\mathbb{Z}$. Le terme *boîte noire*, initialement introduit par Trager et Kaltofen pour les polynômes [77], désigne les matrices creuses ou structurées, dont on ne suppose qu'une unique fonctionnalité : l'application à un vecteur (et éventuellement l'application de sa transposée à un vecteur).

FIG. 1 – *Principe d'une matrice boîte noire dans un corps K*

L'intérêt de cette approche est de préserver la structure des matrices, pour conserver l'efficacité de leur produit matrice-vecteur. De plus, elle permet de manier des matrices de très grandes dimensions, descriptibles avec peu d'éléments, là où une description dense par les $n \times m$ coefficients serait impossible. Les algorithmes *boîte noire* de calcul exacts s'étant fortement développés [76, 44, 115, 46, 39, 30] etc, de paire avec les techniques de préconditionnement pour le calcul exact [12], il s'imposait de les rassembler au sein d'une même bibliothèque.

La généricité d'un intergiciel[1]

Le calcul exact se distingue par la multiplicité des domaines de base manipulés : corps finis, nombres entiers, rationnels, polynômes, ... Et pour chaque domaine, de multiples implémentations sont disponibles, chacune ayant des domaines de prédilection spécifiques. Par exemple, une représentation logarithmique tabulée peut être préférable pour des calculs arithmétiques alors qu'une représentation modulaire sur des flottants permet l'utilisation de routines d'algèbre linéaire numériques. Il faut ajouter à cela, que les meilleurs outils de calcul exact sont dispersés en diverses bibliothèques : la bibliothèque `GMP` [60] pour les calculs multiprécisions, en particulier avec des nombres entiers, la bibliothèque `NTL` [100] pour l'arithmétique des polynômes, ou encore la bibliothèque de calcul algébrique `Givaro` [52]. À l'opposé de ces bibliothèques spécialisées et efficaces, on trouve des logiciels généralistes et complets de calcul, tels que `Maple` [11, 85] ou Mathematica [121]. Ils sont largement diffusés, et très simples d'emploi, mais les algorithmes qu'ils mettent en œuvre ne sont pas toujours les plus performants et ils sont donc souvent inadaptés au calcul intensif.

Le projet ambitieux de LINBOX [33] est de former un intergiciel fournissant les meilleurs algorithmes aux logiciels généralistes, tout en utilisant les bibliothèques spécialisées les plus adaptées pour les calculs de base.

C'est ce double besoin de généricité et de performance qui a conduit à l'utilisation du langage `C++`.

LINBOX-1.0

Après six années de maturation, la première version officielle 1.0 de LINBOX [2] a pu voir le jour en juillet 2005. La conception prévoit désormais quatre niveaux d'utilisation :

- à travers des serveurs de calcul en ligne, et des formulaires `JavaScript`.
- par des exécutables précompilés, ou comme paquetage de logiciels généralistes.
- en incluant les sources et en y accédant par des méthodes générales correspondant à des solutions de problèmes classiques (déterminant, rang, polynôme minimal, ...). La bibliothèque se charge de faire les choix opportuns dans les algorithmes, en fonction des données.
- en incluant directement les algorithmes à un niveau inférieur, laissant ainsi plus de choix à l'utilisateur dans les optimisations.

Les solutions disponibles au troisième niveau d'utilisation sont les fonctions suivantes : déterminant, rang, polynôme minimal et caractéristique, résolution de système linéaire, test si la matrice est définie positive. Chacune de ces solutions est utilisable sur un corps fini ou dans $\mathbb{Z}$, avec des matrices denses ou *boîte noire*.

L'introduction d'algorithmes par blocs (comme la résolution de Lanczos par blocs) a nécessité d'ajouter des algorithmes pour les matrices denses. Nous avons donc introduit les routines

[1] Traduction proposée par D. Durand du terme anglo-saxon *middleware*, utilisé dans le contexte des réseaux, désignant la *classe de logiciels qui assurent l'intermédiaire entre les applications et le transport des données par les réseaux.* Voir `www.linux-france.org/prj/jargonf/I/intergiciel.html`

[2] La bibliothèque, ainsi que sa documentation et l'accès à des serveurs de calculs sont disponibles en ligne à l'adresse `www.linalg.org`

de base d'algèbre linéaire dense, que nous décrirons dans la partie I, ainsi que nos travaux concernant le polynôme caractéristique (parties II et III).

Enfin la bibliothèque dispose d'un jeu de tests pour ses différents algorithmes, ainsi que de programmes d'exemples d'utilisation des solutions. Leur compilation a été testée sur des architectures différentes et des systèmes variés (Linux, Cygwin, Pentium 4, Itanium2-64).

Contexte des expérimentations

Pour illustrer les algorithmes que nous présentons, nous donnons des résultats expérimentaux de leur implémentation en C++. Il peut s'agir de l'implémentation directement disponible dans la bibliothèque LINBOX ou alors d'une implémentation expérimentale, mais faisant appel à des routines de LINBOX. La thèse de Pascal Giorgi [56] décrit en détail l'intégration des routines d'algèbre linéaire dense dans LINBOX, que nous avons réalisé en collaboration. Pour cette raison nous ne rentrons pas dans les détails de l'intégration de nos travaux dans LINBOX, et renvoyant le lecteur à ce document.

Comme nous le verrons dans la partie 2.1.2, ces implémentations reposent grandement sur les routines BLAS d'algèbre linéaire flottante. Nous en utilisons deux implémentations : `ATLAS`[3], version 3.6, qui fournit un code s'adaptant automatiquement lors de l'installation à l'architecture [118] et le `GOTO BLAS`[4], version 1.02, développé par Kazushige Goto [58] plus particulièrement optimisé pour les architectures PC. Par la suite nous les désignerons respectivement par ATLAS et GOTO. Nous utilisons aussi les bibliothèques `GMP`, version 4.1.4 et `NTL`, version 5.4, pour l'arithmétique des nombres entiers et des polynômes. Nous utilisons le compilateur `gcc`, versions de 3.2.3 (sur Itanium) à 4.0 (sur certains PC).

Ces expérimentations servent le plus souvent à comparer la rapidité de calcul des algorithmes. Nous donnons soit le temps CPU en secondes (temps passé effectivement par le processus dans le CPU), soit éventuellement pour les corps finis, la vitesse de calcul. Cette vitesse est obtenue en divisant une estimation du nombre d'opérations algébriques effectuées par le temps CPU utilisé. Son unité est le Mfops, pour Millions d'opérations dans le corps par seconde.

Les matrices que nous avons utilisées sont le plus souvent des matrices denses dont les coefficients sont tirés aléatoirement et uniformément dans le corps fini utilisé, ou dans un intervalle $[0 \dots n]$ de $\mathbb{Z}$. Pour les expérimentations avec les méthodes *boîte noire*, nous avons utilisé des matrices creuses provenant de diverses applications (complexes simpliciaux, robotique, matrices d'adjacence de graphes fortement réguliers, ...). Toutes ces matrices ainsi qu'une description du format utilisé pour leur représentation sont disponible sur internet[5].

[3] `www.netlib.org/atlas`
[4] `www.tacc.utexas.edu/resources/software/`
[5] `www-lmc.imag.fr/lmc-mosaic/Clement.Pernet/Matrices/`

RAPPELS THÉORIQUES ET NOTATIONS

Nous rappelons ici les notions mathématiques de base sur lesquelles nous nous appuyons dans notre étude.

Ouvrages de référence : Pour les approfondir, on pourra se reporter à l'excellent ouvrage de Gantmacher [50] pour les questions générales d'algèbre linéaire. Le livre de Gerhard et von-zur-Gathen [51] ainsi que la référence encyclopédique [59] sont les références en ce qui concerne l'algorithmique pour le calcul formel. Pour les questions de complexité algébrique, on pourra se reporter à l'ouvrage de Bürgisser, Clausen et Shokrollahi [13]. Enfin, le livre d'Abdeljaoued et Lombardi [84] illustre la théorie de la complexité algébrique par l'algèbre linéaire et particulièrement le calcul du polynôme caractéristique.

Algèbre linéaire

Soit une matrice carrée A de dimension $n \times n$ sur un anneau principal $\mathcal{A}$.

Polynôme caractéristique : Nous définissons le polynôme caractéristique comme le déterminant de sa matrice caractéristique :

$$P^A_{\text{car}}(X) = \det(XI_n - A).$$

Pour des raisons de commodité, nous avons préféré la définition anglo-saxonne rendant ce polynôme unitaire, à la définition française, $P^A_{\text{car}}(X) = \det(A - XI_n)$, ajoutant un facteur $(-1)^n$ dans le monôme dominant.

Nous rappelons également la définition du déterminant d'une matrice $A = [A_{i,j}]$:

$$\det(A) = \sum_{\sigma \in \mathfrak{S}_n} \epsilon(\sigma) \prod_{j=1}^{n} a_{\sigma(j),j} \tag{1}$$

où $\mathfrak{S}_n$ est l'ensemble des permutations de l'intervalle $[1 \dots n]$ et $\epsilon(\sigma)$ est la signature de la permutation σ.

Polynôme minimal : Considérons le $\mathcal{A}[X]$-module formé par $\mathcal{A}^n$ dont la mutliplication externe par $\mathcal{A}[X]$ est définie comme suit : $\forall(\alpha, \beta, v) \in \mathcal{A} \times \mathcal{A} \times \mathcal{A}^n \ \ (\alpha X + \beta).v = \alpha Av + \beta v$. Le polynôme minimal du vecteur v relativement à la matrice A est défini comme le polynôme unitaire de plus petit degré, annulateur de v dans ce $\mathcal{A}[X]$-module. Plus simplement, on peut

le définir comme le polynôme unitaire P de plus petit degré tel que $P(A)v = 0$ dans $\mathcal{A}^n$. Nous le noterons $P_{\text{min}}^{A,v}$.

Le polynôme minimal de la matrice A est défini comme le polynôme unitaire de plus petit degré tel que $P(A) = 0$ dans $\mathcal{A}^{n\times n}$. Nous le noterons P_{min}^{A}.

Enfin, pour être complets, nous définissons aussi le polynôme $P_{\text{min}}^{u,A,v}$, pour deux vecteurs u et v, comme le polynôme minimal générateur de la séquence des scalaires

$$ {}^tuv,\ {}^tuAv, \ldots,\ {}^tuA^iv, \ldots $$

Il est est utilisé en particulier dans l'algorithme *boîte noire* de Wiedemann [119].

Le polynôme minimal de A étant annulateur de A, il vérifie aussi $P(A)v = 0$. Il est donc divisible par $P_{\text{min}}^{A,v}$. Par ailleurs, on sait par le théorème de Cayley-Hamilton que le polynôme caractéristique est annulateur de A. Enfin $P_{\text{min}}^{A,v}$ annule la séquence ${}^tuv,\ {}^tuAv, \ldots,\ {}^tuA^iv, \ldots$ et est donc multiple de $P_{\text{min}}^{u,A,v}$. Nous avons donc les relations de divisibilité suivantes dans $\mathcal{A}[X]$:

$$ P_{\text{min}}^{u,A,v} \mid P_{\text{min}}^{A,v} \mid P_{\text{min}}^{A} \mid P_{\text{car}}^{A}. $$

Matrice compagnon : Soit $P = X^d + \sum_{i=0}^{d-1} c_i X^i$ un polynôme unitaire de degré d. La matrice compagnon associée au polynôme P, notée C_P, est la matrice de dimensions $d \times d$

$$ C_P = \begin{bmatrix} 0 & & & -c_0 \\ 1 & 0 & & -c_1 \\ & \ddots & \ddots & \vdots \\ & & 1 & -c_{n-1} \end{bmatrix} $$

Son polynôme caractéristique égale son polynôme minimal et vaut P : $P_{\text{car}}^{C_P} = P_{\text{min}}^{C_P} = P$.

Forme normale de Frobenius : Si la matrice A est maintenant définie sur un corps K, on définit la transformation de similitude de la matrice A avec la matrice inversible U comme étant la matrice $U^{-1}AU$. On dit alors que deux matrices A et B sont équivalentes par transformation de similitude (ou semblables), si il existe $U \in \text{GL}_n(K)$ tel que $B = U^{-1}AU$. On peut alors décomposer $\mathcal{M}_n(K)$ en classes d'équivalence pour cette relation. La forme normale de Frobenius définit alors un unique représentant pour chacune de ces classes :

Définition. Toute matrice $A \in K^{n\times n}$ est semblable à une unique matrice diagonale par blocs,

$$ F = P^{-1}AP = \text{diag}(C_{f_1}, C_{f_2}, \ldots, C_{f_\phi}) $$

où les blocs C_{f_i} sont des blocs compagnons définis par les polynômes unitaires $f_1, \ldots, f_\phi$ vérifiant $f_{i+1}|f_i$ pour $1 \leq i < \phi$ et appelés les facteurs invariants de A. Cette matrice est appelée forme normale de Frobenius de A.

De manière similaire, sur un anneau principal $\mathcal{A}$ on considère les transformations d'équivalence, de la forme UAV où U et V sont inversibles, et on peut définir des classes d'équivalence de la même façon que pour les transformations de similitude. La forme normale servant à caractériser chaque classe d'équivalence est la forme normale de Smith :

Définition. Toute matrice A $n \times m$ définie sur un anneau principal est équivalent à une matrice diagonale

$$S = UAV = \begin{bmatrix} s_1 & & & \\ & s_2 & & \\ & & \ddots & \\ & & & s_n \end{bmatrix}$$

où U et V sont des matrices $n \times n$ et $m \times m$ unimodulaires (de déterminant inversible dans $\mathcal{A}$), où les $r = \text{rang}(A)$ premiers s_i sont non nuls et vérifient $s_{i+1}|s_i$ pour $i \in [1 \dots r-1]$. Cette matrice est appelée forme de Smith. Elle est unique à la multiplication près des s_i par des éléments inversibles de $\mathcal{A}$.

A noter qu'il s'agit ici de la définition donnée par Villard dans [112] facilitant la mise en relation de la forme de Smith avec la forme de Frobenius. Dans la définition classique [101, 50], les facteurs invariants sont ordonnés dans le sens inverse : $\forall i \in [1 \dots r-1] \;\; s_i|s_{i+1}$.

Les polynômes $f_1, \dots f_\phi$, comme les éléments $s_1, \dots s_n$ sont appelés les facteurs invariants de la matrice. Cette identification provient de la propriété suivante :

Théorème ([50, Chap 6])**.** *La forme normale de Frobenius de A est diag$(C_{f_1}, \dots, C_{f_\phi})$ si et seulement si la forme normale de Smith de la matrice $XI_n - A$ est diag$(f_1, \dots, f_\phi, 1, \dots, 1)$.*

La forme normale de Frobenius résume tous les invariants algébriques et géométriques de la matrice A. En particulier, on montre que $f_1 = P^A_{\text{min}}$ et que $\prod_{i=1}^{\phi} f_i = P^A_{\text{car}}$; le déterminant se lit sur le coefficient constant de ce polynôme et le rang est le nombre de facteurs invariants ayant un coefficient constant nul.

Modèle de calcul et complexité

Pour les algorithmes déterministes, nous utilisons un modèle de calcul de type RAM (Random Access Machine) binaire ou arithmétique, dont on trouvera une description dans [2, §1.2].

Pour les algorithmes probabilistes, nous supposons disposer aussi d'un générateur aléatoire : il s'agit d'une séquence, de taille polynomiale dans la taille des entrées, de bits aléatoires, ou d'éléments d'un sous-ensemble fini du domaine. Un algorithme est probabiliste s'il retourne un résultat juste pour une proportion constante des entrées aléatoires. Le comportement pour les autres entrées définit deux classes d'algorithmes probabilistes :

les algorithmes *Monte Carlo* : dont le résultat n'est pas spécifié dans ce cas,

les algorithmes *Las Vegas* : qui retournent `Echec`.

Ainsi l'utilisateur d'un algorithme *Las Vegas* peut l'exécuter tant qu'il retourne `Echec` pour en faire un algorithme *toujours correct, probablement rapide*. A l'inverse, les algorithmes *Monte Carlo* ne permettent pas de certifier le résultat et sont donc *toujours rapides, probablement corrects*.

Pour les calculs de complexité, nous utiliserons le modèle de complexité algébrique (comptant le nombre d'opérations algébriques effectuées sur un corps ou un anneau) et le modèle binaire (comptant le nombre d'opérations binaires effectuées). Plus précisément nous utilisons essentiellement la complexité algébrique (pour les corps finis) sauf dans les chapitres 5, 9 et 12 où nous utilisons la complexité binaire pour les algorithmes dans $\mathbb{Z}$.

Nous rappelons la signification de la notation $\mathcal{O}(n)$:

$$f(n) = \mathcal{O}(g(n)) \text{ si et seulement si } \exists n_0, C, \forall n \geq n_0 \; f(n) \leq Cg(n)$$

De plus, la notation $\tilde{\mathcal{O}}()$ (*soft O*) inclut les facteurs logarithmiques et polylogarithmiques :

$$\tilde{\mathcal{O}}(f(n)) = \mathcal{O}(f(n)(\log n)^{\alpha})$$

une certaine constante α.

Si les notations $\mathcal{O}()$ et $\tilde{\mathcal{O}}()$ conviennent à l'étude asymptotique, elles dissimulent des constantes pouvant rendre un algorithme impraticable, quoique asymptotiquement le meilleur. Par exemple, la dimension n à partir de laquelle un algorithme en $6n^{2,807}$ est moins coûteux qu'un algorithme en $2n^3$ est $n = 297$. C'est ainsi que les algorithmes de multiplication matricielle rapide n'ont longtemps été considérés que pour leur intérêt théorique.

Notre point de vue étant ici l'efficacité en pratique des algorithmes, nous nous attachons à détailler le plus souvent leur complexité en donnant systématiquement la constante dissimulée par la notation $\mathcal{O}()$. Ainsi, nous donnons pour un algorithme X sa complexité $T_{\mathrm{X}}(n)$ par le terme dominant de sa complexité, incluant la constante multiplicative. Il s'agit d'un abus de notation qui nous fait écrire $T_{\mathrm{X}}(n) = f(n)$ quand la complexité de l'algorithme X est $f(n) + o(f(n))$.

Matériel de récurrence pour la complexité algébrique

Dans les chapitres 3 et 4, nous étudierons les complexités algébriques de différents problèmes réduits au produit matriciel. En particulier, nous donnons une expression de la constante du terme dominant de ces complexités. Pour ce faire nous aurons besoin des deux lemmes suivants.

Lemme 0.1. *Soit m un entier positif, $\omega > 2$ et C, a, g, e des constantes. Si*

1. $T(m) = CT(\frac{m}{2}) + am^{\omega} + \epsilon(m)$, *avec* $\epsilon(m) \leq gm^2$, *et* $T(1) = e$
2. $\log_2(C) < \omega$.

Alors

$$T(m) = \frac{a2^{\omega}}{2^{\omega} - C} m^{\omega} + o(m^{\omega}) = \mathcal{O}(m^{\omega}).$$

Démonstration. On pose $t = \log_2(m)$. En déroulant la formule de récurrence, on obtient

$$\begin{aligned} T(m) &= C^t T(1) + am^{\omega} \sum_{i=0}^{t-1} \left(\frac{C}{2^{\omega}}\right)^i + \sum_{i=0}^{t-1} C^i \epsilon(\frac{m}{2^i}) \\ &= C^t T(1) + am^{\omega} \frac{1 - \left(\frac{C}{2^{\omega}}\right)^t}{1 - \frac{C}{2^{\omega}}} + \sum_{i=0}^{t-1} C^i \epsilon(\frac{m}{2^i}). \end{aligned}$$

Si $C \neq 4$, on a

$$T(m) = \frac{a2^\omega}{2^\omega - C} m^\omega + kC^t + g'm^2$$

où $g' < \frac{4g}{4-C}$ et $k < T(1) - \frac{a2^\omega}{2^\omega - C} - g'$. Par ailleurs si $C = 4$, on a

$$T(m) = \frac{a2^\omega}{2^\omega - C} m^\omega + k'C^t + gm^2 \log_2(m),$$

où $k' < T(1) - \frac{a2^\omega}{2^\omega - C}$. Dans les deux cas, on obtient $T(m) = \frac{a2^\omega}{2^\omega - C} m^\omega + o(m^\omega)$, en utilisant le fait que $C^t = m^{\log_2(C)}$. □

Voici maintenant une généralisation pour le cas où le problème fait intervenir deux dimensions indépendantes, m et n.

Lemme 0.2. *Soit m et n deux entiers positifs, $\omega > 2$ et C, a, b, g, h, e des constantes. Si*

1. $T(m,n) = C(T(\frac{m}{2}, n) + T(\frac{m}{2}, n - \frac{m}{2})) + am^\omega + bm^{\omega-1}n + \epsilon(m,n)$*, avec* $\epsilon(m,n) \leq gm^2 + hmn$ *et* $T(1,n) \leq en$*.*
2. $\log_2(C) < \omega - 2$

Alors $T(m,n) = \mathcal{O}(m^\omega) + \mathcal{O}(m^{\omega-1}n)$.

Démonstration. De la même façon, on déroule la récurrence pour trouver

$$\begin{aligned}
T(m,n) &= C^t \sum_{i=1}^{m} T(1, n-i+1) + am^\omega \frac{1 - \left(\frac{C}{2^\omega}\right)^t}{1 - \frac{C}{2^\omega}} \\
&\quad + \sum_{i=0}^{t-1} \sum_{j=1}^{2^i} C^i \left[\left(\frac{m}{2^i}\right)^{\omega-1} (n - (j-1)\frac{m}{2^i}) + \epsilon\left(\frac{m}{2^i}, n - (j-1)\frac{m}{2^i}\right) \right] \\
&= C^t \left(mn - \frac{m(m-1)}{2} - \frac{a2^\omega}{2^\omega - C} \right) + m^\omega \left(\frac{a2^\omega}{2^\omega - C} - \sum_{i=0}^{t-1} \sum_{j=1}^{2^i} \frac{j-1}{2^{i\omega}} \right) \\
&\quad + m^{\omega-1}n \sum_{i=0}^{t-1} \sum_{j=1}^{2^i} \left(\frac{C}{2^{\omega-1}}\right)^i + m^2 \sum_{i=0}^{t-1} \sum_{j=1}^{2^i} \left(\frac{C}{4}\right)^i (g - h(j-1)) \\
&\quad + hmn \sum_{i=0}^{t-1} \sum_{j=1}^{2^i} \left(\frac{C}{2}\right)^i \\
&= C^t k(m,n) + \alpha m^\omega + \beta m^{\omega-1}n + o(m^\omega) + o\left(m^{\omega-1}n\right)
\end{aligned}$$

pour certaines constantes α et β, et comme $k(m,n) = \mathcal{O}(mn) + \mathcal{O}(m^2)$. □

Le lemme 0.1 permet d'obtenir le terme dominant dans sa totalité (constante et exposant) à partir des paramètres de la formule de récurrence. Pour le lemme 0.2, nous avons préféré ne laisser que l'information sur l'exposant par souci de simplicité. Les constantes α et β pourront ensuite être déterminées facilement en injectant l'expression $T(m,n) = \alpha m^\omega + \beta m^{\omega-1}n$ dans la formule de récurrence.

Première partie

Algèbre linéaire dense efficace sur un petit corps fini

INTRODUCTION

Motivation

En calcul exact, les corps finis jouent un rôle central : soit par leurs applications en cryptographie ou en théorie des codes, soit comme brique de base pour des calculs dans $\mathbb{Z}$ ou $\mathbb{Q}$. En effet, les techniques de restes chinois ou de remontée p-adique permettent de réduire des calculs tels que le déterminant d'une matrice entière ou la résolution d'un système entier dans $\mathbb{Q}$ à des calculs sur des corps finis. Contrairement aux nombres entiers, tous les coefficients d'un corps fini peuvent être représentés dans le même espace mémoire, ce qui permet de développer des arithmétiques efficaces, en utilisant les unités élémentaires de mémoire des ordinateurs : les mots machine.

Par ailleurs, l'expérience acquise en algèbre linéaire pour le calcul numérique montre qu'un ensemble restreint de routines de base particulièrement optimisées suffisent à assurer des performances excellentes pour la plupart des problèmes faisant appel à l'algèbre linéaire. Il s'agit de la démarche qui a conduit à l'élaboration des BLAS (Basic Linear Algebra Subroutines) [83, 28, 27].

De telles bibliothèques n'existaient pas pour le calcul exact avec des corps finis. Nous avons donc cherché à combler, partiellement, ce manque en développant des routines de base, sur lesquelles nous allons pouvoir par la suite nous appuyer pour le calcul du polynôme caractéristique.

Démarche adoptée

Opération fondamentale en algèbre linéaire, le produit matriciel tient toujours une place de choix parmi les problèmes d'algorithmique en algèbre linéaire et son importance va croissante. Jusqu'à la fin des années 1960, les algorithmes en algèbre linéaire étaient décrits par des boucles d'opérations arithmétiques sur un domaine de base. Et les complexités algébriques des problèmes canoniques (déterminant, rang, résolution de systèmes, etc) étaient cubiques en la dimension des matrices considérées, tout comme celle du produit matriciel, mais sans lien apparent. Cette complexité était alors considérée comme optimale.

Or en 1969, Strassen a ouvert une brèche en proposant dans [108] un algorithme sous-cubique (en $\mathcal{O}\left(n^{2,8074}\right)$) pour effectuer le produit de deux matrices carrées d'ordre n. Depuis, de nombreux autres algorithmes ont été proposés, réduisant l'exposant à $2,7799$ (Bini, Capovani, Lotti et Romani dans [7]), $2,55$ (Schönage dans [99]), $2,39$ (Coppersmith et Winograd dans [17]) et enfin $2,376$ (Coppersmith et Winograd dans [18]). Enfin, notons aussi les récents travaux de Cohn et Umans [14] utilisant une approche différente, basée sur la théorie

des groupes, pour tenter d'approcher l'exposant $\omega = 2 + o(1)$, mais n'obtiennent à l'heure actuelle que l'exposant $\omega = 2,41$.

Par la suite, nous désignerons par ω, un exposant *réalisable* pour le produit matriciel, en reprenant les notations de [51, §12.1]. On appelle *l'exposant du produit matriciel* μ, la borne inférieure sur tous les exposants réalisables ω. La valeur de μ reste à l'heure actuelle inconnue, et l'hypothèse $\mu = 2$ qui semble la plus réaliste n'est qu'une conjecture.

La question s'est immédiatement posée de savoir si les autres problèmes canoniques verraient leur complexité algébrique réduite de ce fait (Strassen, dans son article [108], appliquait déjà son résultat à l'élimination de Gauss). La réponse affirmative a été rapidement établie pour la plupart d'entre eux (voir [10, 13] pour une revue détaillée sur le sujet), dont la complexité algébrique est devenue $\mathcal{O}(n^\omega)$ par des *réductions* au produit matriciel. A notre connaissance, seul le calcul du polynôme caractéristique ne répond pas à la règle et nous aurons l'occasion de revenir sur le sujet dans la partie II.

Dès lors, le produit matriciel est devenu un constituant de base de la plupart des algorithmes d'algèbre linéaire, pour permettre les réductions de complexité. Le phénomène se poursuit encore actuellement pour améliorer des complexités binaires, comme par exemple celle du déterminant dans [78].

Parallèlement, cette réduction s'est aussi imposée au niveau des implémentations en calcul numérique, indépendamment de l'existence d'un exposant sous-cubique. En effet, comme nous le verrons dans la partie 2.1.2, c'est sur le produit matriciel que les efforts d'optimisation de code ont le plus porté leurs fruits et ces réductions permettent donc d'en tirer parti pour les autres algorithmes.

Ainsi, pour élaborer des implémentations efficaces sur des corps finis, nous procédons de manière similaire. Les efforts d'optimisation porteront essentiellement sur deux routines centrales : le produit matriciel d'une part et la résolution de systèmes triangulaires multiples d'autre part. Les autres routines de bases se réduiront à ces deux routines grâce à des algorithmes récursifs par blocs.

Pour garantir l'efficacité des routines centrales, nous considérons autant le niveau de l'implémentation (réduction modulaire différée, utilisation de l'arithmétique flottante, optimisation de la gestion du cache) que de l'algorithmique (utilisation de la multiplication matricielle rapide de Winograd). Ces considérations conduisent à des algorithmes en cascades reposant sur des paramètres seuils dont nous proposons des méthodes de détermination fine.

1

PRÉLIMINAIRES : ARITHMÉTIQUE DES CORPS FINIS ET PRODUIT SCALAIRE

La première tâche pour implémenter des routines d'algèbre linéaire exacte est de développer l'arithmétique sous-jacente. En effet les objets mathématiques impliqués, tels que les corps finis, n'existent pas au niveau des unités arithmétiques des ordinateurs actuels. Une émulation logicielle est donc nécessaire. L'implémentation d'arithmétiques efficaces pour les corps finis est une partie importante du développement de routines d'algèbre linéaire exacte.

Ce sujet a déjà été largement étudié et nous nous reportons par exemple à [31] et aux références qui y sont incluses pour un revue détaillée sur le sujet. Nous rappelons ici les différentes façons d'implémenter une telle arithmétique et argumentons notre préférence pour l'une d'elles dans le cadre des routines d'algèbre linéaire efficaces.

1.1 Implémentation des corps finis

Par la suite, nous distinguons les corps finis dits *premiers*, dont la cardinalité est un nombre premier, parmis les corps finis en général, appelées corps de Galois, dont la cardinalité est du type $q = p^k$, avec p premier et $k \geq 1$. Pour $k > 1$, ils sont obtenus comme une extension algébrique de degré k à partir d'un corps premier. Nous nous concentrons principalement sur les corps premiers. Afin de faciliter l'utilisation de l'arithmétique des mots machine, nous considérons les corps de cardinalité inférieure à 2^γ, où γ est le nombre de bits de la représentation entière utilisée. Par exemple γ peut valeur 32 ou 64 pour des types entiers ou encore 53 pour des flottants double précision dont on n'utilise que la mantisse. C'est en ce sens que nous appelons ces petits corps des *petits corps*.

Une implémentation consiste à choisir une représentation pour les éléments du corps, ainsi qu'une arithmétique pour les manipuler. Nous décrivons dans la partie 1.1.1 les principaux types d'implémentations existantes pour les corps finis, puis nous décrirons plus précisément dans la partie 1.1.2 celle que nous avons privilégiée pour la plupart de nos expérimentations utilisant des corps premiers.

1.1.1 Les principaux types existants

Représentation classique avec divisions entières : Les éléments sont représentés par les entiers d'un intervalle de longueur p. On distingue en particulier deux cas : les *représentations positives*, utilisant l'intervalle $[0, \ldots, p-1]$ et les *représentations centrées* utilisant l'intervalle

$$[-\frac{p-1}{2}, \ldots, \frac{p-1}{2}]$$

pour $p > 2$. Les opérations du groupe additif sont faites par l'unité arithmétique de la machine, suivies d'un test et d'une éventuelle correction. La multiplication est suivie d'une réduction modulaire alors que la division est calculée par l'algorithme d'Euclide étendu.

Représentation de Montgomery : Cette représentation, proposée dans [86], permet d'éviter la réduction modulaire coûteuse pour les multiplications. Une représentation décalée est utilisée et la réduction modulaire est remplacée par des multiplications. Les autres opérations restent inchangées, mise à part la division.

Inverse en virgule flottante : Une autre idée est de réduire le coût de la réduction modulaire en précalculant l'inverse de la caractéristique p sur un nombre flottant. Ainsi seulement deux multiplications sur des nombres flottants et des arrondis sont nécessaires. Cependant, les arrondis peuvent introduire des erreurs de ± 1 et un ajustement doit donc être fait, comme c'est le cas dans la bibliothèque NTL [100].

Logarithmes discrets (ou logarithmes de Zech) : Ici, les éléments sont représentés par la puissance d'un générateur du groupe multiplicatif. Ainsi, les opérations du groupe multiplicatif sont effectuées par des additions et soustractions modulo $p - 1$. Néanmoins les additions et soustractions sont rendues plus complexes et sont généralement tabulées [31, §2.4].

Concernant les corps de Galois non-premiers, une première approche repose sur la propriété suivante : GF$(p^k) \equiv \mathbb{Z}_p/(P)$ où P est un polynôme irréductible de degré k sur Z_p. Ainsi, les éléments sont représentés par des polynômes à coefficients dans un corps premier. L'arithmétique est donc celle de polynômes modulo P dont les coefficients utilisent une des arithmétiques décrites ci-dessus. Alternativement, on peut utiliser la représentation logarithmique Zech, en représentant chaque élément par la puissance correspondante d'un polynôme générateur, de la même façon que pour les les corps premiers.

1.1.2 `Modular<double>` : une implémentation flottante pour l'algèbre linéaire exacte

Comme nous l'avons mentionné précédemment, les architectures informatiques modernes (Pentium III et IV, Athlons, Itaniums, Optérons, ...) privilégient les calculs en virgule flottante, en particulier concernant l'opération `AXPY` ou *fused-mac*. Par ailleurs la réduction modulaire par inverse flottant est parmi les plus efficaces (voir [31] sur le sujet). C'est pour ces raisons que nous avons développé une implémentation de corps fini premier, basée sur les flottants double précision (type `double` en C). Les éléments sont donc représentés par des entiers stockés dans la mantisse d'un `double`. L'exposant est inutilisé dans cette représentation. Les opérations arithmétiques du groupe additif ainsi que la multiplication et le `AXPY` sont effectués

par les unités arithmétiques machine alors que la division est toujours faite par un algorithme d'Euclide étendu. Enfin la réduction modulaire utilise le principe de l'inverse flottant décrit dans la partie 1.1.1. Nous verrons dans la partie 2.1.2 que cette implémentation permet surtout d'utiliser les routines numériques directement sur les éléments, sans effectuer de copies ni de conversions, ce qui est un avantage majeur. Plus de détails sur cette implémentation sont donnés dans [56]. La caractéristique principale de cette implémentation est qu'elle utilise un type flottant, dédié aux calculs approchés sur les réels, pour des calculs **exacts** sur des corps finis.

1.2 Le produit scalaire

L'opération essentielle en algèbre linéaire est sans doute la succession d'une multiplication et d'une addition (ou accumulation à un résultat temporaire) : $y \leftarrow ax + y$. Souvent appelée `AXPY`, cette opération peut être vue comme atomique, surtout depuis son introduction en virgule flottante dans les processeurs sous le nom de *fused-mac*. Ainsi, les arithmétiques de corps premiers définissent aussi cette opération, ce qui permet de n'appliquer qu'une réduction modulaire pour deux opérations arithmétiques.

Le produit scalaire consiste ensuite à enchaîner ces opérations sur les éléments de deux vecteurs, ce qui permet en particulier d'utiliser les structures de *pipeline* des architectures dans le cas des calculs flottants. En revanche, sur des corps finis, la réduction modulaire après chaque `AXPY` est pénalisante (ajout d'opérations et éventuellement de branchements conditionnels) et empêche une mise en *pipeline* efficace des calculs.

De ce constat est née l'idée de différer la réduction modulaire autant que possible. En effet, pour toutes les implémentations décrites précédemment, sauf celle des logarithmes de Zech, les éléments sont plongés canoniquement dans $\mathbb{Z}$ et peuvent donc être maniés par les unités arithmétiques machine. Une réduction modulaire finale permet ensuite de retrouver le résultat dans le corps fini. La représentation de $\mathbb{Z}$ par les mots machine est bien sûr limitée à un intervalle borné. L'approche différée n'est donc valide que lorsque les résultats restent dans cet intervalle.

Par exemple si on utilise une représentation positive sur des mots de γ bits avec le modulo p, on peut effectuer λ accumulations sans réduction modulaire si

$$\lambda(p-1)^2 < 2^\gamma. \tag{1.1}$$

Si la représentation est centrée, (i.e. $-\frac{p-1}{2} \leq x \leq \frac{p-1}{2}$ pour la représentation d'un élément x du corps premier impair $\mathbb{Z}_p$), l'équation 1.1 devient :

$$\lambda \left(\frac{p-1}{2}\right)^2 < 2^{\gamma-1} \tag{1.2}$$

ce qui améliore λ d'un facteur 2. A noter qu'un bit est pris à la mantisse pour exprimer le signe (d'où le $\gamma - 1$).

Ainsi, le surcoût des réductions modulaires peut être amorti puisque seulement $\left\lceil \frac{n}{\lambda} \right\rceil$ réductions sont nécessaires pour un produit scalaire de dimension n.

Par ailleurs, on note que la représentation centrée est mieux adaptée aux calculs avec réduction modulaire différée. Dans la mesure où on a le choix d'utiliser cette représentation, il faut

la préférer. Cependant, de nombreuse implémentations existantes pour les corps finis utilisent des représentations positives. Pour cette raison, nous déclinons par la suite chaque résultat concernant les réductions modulaires différées pour les représentations positives et centrées.

2

LE PRODUIT MATRICIEL

Nous présentons ici des travaux sur le produit matriciel que nous avons initiés dans [91], publié dans [34] et prolongés depuis.

2.1 Par l'algorithme classique

Nous considérons l'opération $C \leftarrow A \times B$ où la matrice A a pour dimensions $m \times k$, B, $k \times n$ et C, $m \times n$. Par algorithme classique, nous désignons la méthode qui consiste à cacluler chacun des $m \times n$ coefficients du résultat comme un produit scalaire de dimension k. Les opérations arithmétiques sont donc l'addition et la multiplication sur le corps fini. Nous étudions ici comment les ordonnancer de la manière la plus efficace.

2.1.1 Réduction modulaire différée

Le produit matriciel étant vu comme un ensemble de produits scalaires, on peut donc lui appliquer la remarque de la partie 1.2 : la réduction modulaire peut être différée. Soit λ le nombre maximal de `AXPY` pouvant être effectués sans réduction modulaire. Nous distinguerons essentiellement deux cas :

– celui des représentations positives sur des mots de γ bits non signés, pour lesquelles λ vérifie
$$\lambda(p-1)^2 < 2^\gamma \leq (\lambda+1)(p-1)^2. \tag{2.1}$$
– celui des représentations centrées sur des mots de γ bits signés, pour lesquelles λ vérifie
$$\lambda\left(\frac{p-1}{2}\right)^2 < 2^{\gamma-1} \leq (1+\lambda)\left(\frac{p-1}{2}\right)^2. \tag{2.2}$$

Cette contrainte induit un découpage en blocs des matrices à multiplier : A est découpée en k/λ blocs de dimension $m \times \lambda$ et B en k/λ blocs de dimension $\lambda \times n$ (sauf les derniers blocs de A et B qui sont de dimension plus petite).

Ainsi, l'algorithme consiste maintenant à effectuer les produits par blocs par une arithmétique de $\mathbb{Z}$, en effectuant les réductions modulaires entre chaque opération de blocs. Cette organisation laisse maintenant entrevoir la possibilité d'utiliser les routines numériques spécialisées, pour effectuer les opérations de blocs. C'est ce que nous allons maintenant étudier.

2.1.2 Utilisation de l'arithmétique flottante

Une approche naïve pour le produit matrice-vecteur et la multiplication matricielle serait de les considérer comme respectivement n ou n^2 produits scalaires. De la sorte l'efficacité serait celle du produit scalaire, traitée dans [31] : en général autour d'une opération de corps pour deux cycles d'horloge.

Or des efforts considérables ont été menés pour améliorer ces opérations linéaires en tirant parti de la structure hiérarchique de la mémoire des ordinateurs actuels ainsi que des unités arithmétiques spécialisées [118]. Dans la plupart des architectures des ordinateurs modernes, un accès à la mémoire vive est plus de cent fois plus coûteux qu'une opération arithmétique. Pour contourner ce ralentissement, la mémoire est structurée en deux ou trois niveaux de mémoire cache, jouant le rôle de tampon pour réduire le nombre d'accès à la mémoire vive et réutiliser au maximum les données chargées. Cette technique n'est valable que si l'algorithme permet la réutilisation de certaines données. C'est la principale raison pour laquelle le produit matriciel est particulièrement bien adapté à ce type d'optimisation : c'est la première opération basique en algèbre linéaire dont la complexité arithmétique $\mathcal{O}(n^3)$ est d'un ordre de grandeur supérieur à la complexité en mémoire $\mathcal{O}(n^2)$.

Ces considérations ont mené les numériciens à l'élaboration des BLAS [83, 28, 27], dont l'efficacité est principalement basée sur un noyau optimisé pour le produit matriciel. L'un de ses principes constitutifs est d'effectuer les produits matriciels par blocs de petite dimension, de sorte que les arguments pour un produit de blocs tiennent dans la mémoire cache, limitant ainsi les surcoûts d'accès mémoire. Initialement, ces routines étaient fournies par le constructeur d'une architecture donnée. Mais les récents travaux de [118] ont permis le développement de BLAS génériques s'adaptant aux spécificités de la plupart des architectures existantes grâce à un système d'optimisations basées sur des expérimentations lors de l'installation. L'efficacité de telles routines est sans comparaison : par exemple la puissance de crête du produit matriciel sur un Pentium 4 cadencé à 3,4 Ghz est de $5,276$ milliards d'opérations par seconde, ce qui correspond à $1,55$ opérations arithmétiques par cycle d'horloge. Cette efficacité provient d'une utilisation pertinente des niveaux de cache, des structures de *pipeline* et du parallélisme interne à ce type de processeur.

Il est donc tout naturel d'utiliser ces routines pour effectuer les produits matriciels sur $\mathbb{Z}$ des blocs décrits dans la partie 2.1.1. Pour ce faire on doit pouvoir convertir les éléments du corps fini vers leur image dans $\mathbb{Z}$ représentée sur un `double`. A l'inverse on doit aussi disposer du mécanisme réciproque de conversion d'un entier représenté sur un `double` vers un élément du corps fini, impliquant une réduction modulaire. Ces conversions sont toujours réalisables, mais elles demandent de dupliquer les blocs de matrices dans le cas d'une implémentation quelconque. En revanche, l'avantage de l'implémentation `Modular<double>` tient dans le fait que les éléments sont directement représentés sur des `double`, et les BLAS peuvent donc les utiliser sans recopie. Après le calcul sur $\mathbb{Z}$, une simple réduction modulaire suffit pour retrouver le résultat sur le corps fini.

2.2 L'algorithme rapide de Winograd

En calcul numérique, les travaux de [63] puis de [69, 70] ont mis en évidence le gain en pratique des algorithmes de multiplication matricielle rapide. L'instabilité numérique de ces

algorithmes a été mise en évidence et analysée par Bini et Lotti dans [8] puis par Higham dans [63, 64]. Cette instabilité dépend fortement des matrices utilisées (parfois l'algorithme classique est même moins stable). Même si leur intégration au sein des BLAS a été envisagée dans [29, 70], les numériciens considèrent ces algorithmes comme instables et des BLAS de référence, comme ATLAS, refusent de considérer leur implémentation.

Les techniques d'agrégation et cancellation de Laderman [82] sont aussi comparées dans [79]. Elles permettent d'obtenir une meilleure stabilité que la variante de Winograd mais avec des temps de calcul plus longs.

Mais pour le calcul exact, la stabilité numérique ne pose pas de problème, et nous allons étudier comment intégrer l'algorithme rapide de Strassen [108] pour une routine sur les corps finis. Plus particulièrement, nous considérerons la variante de Winograd décrite dans [51, algorithme 12.1], qui en réduit le nombre d'additions.

2.2.1 Mise en cascade avec l'algorithme classique

Asymptotiquement, l'algorithme de Winograd améliore le nombre d'opérations arithmétiques requises pour le produit matriciel de $\mathcal{O}(n^3)$ à $\mathcal{O}(n^{2,8074})$. Mais pour une dimension n donnée, le nombre minimal d'opérations n'est pas atteint en appliquant tous les appels récursifs de l'algorithme, mais en remplaçant ceux d'un certain niveau seuil par des produits matriciels classiques. Pour justifier cette remarque, nous considérons l'algorithme en cascade effectuant k niveaux récursifs de l'algorithme de Winograd puis des multiplications classiques. Nous allons étudier $T(n,k)$, le nombre d'opérations arithmétiques (additions, soustractions et multiplications) en fonction de n et k, d'abord pour $n = 2^i$ avec $i \geq k$ puis pour n quelconque.

Pour $n = 2^i$: Soit A le nombre d'additions de bloc dans un niveau récursif de l'algorithme (respectivement $A = 18$ et $A = 15$ pour les algorithmes de Strassen et Winograd).

$$\begin{aligned} T(n,k) &= 7T(\frac{n}{2}, k-1) + A\left(\frac{n}{2}\right)^2 \\ &= 7^k T(\frac{n}{2^k}, 0) + \frac{A}{4}n^2 \sum_{i=0}^{k-1} \left(\frac{7}{4}\right)^i \\ &= 7^k \left(2\left(\frac{n}{2^k}\right)^3 - \left(\frac{n}{2^k}\right)^2\right) + \frac{A}{3}n^2\left(\left(\frac{7}{4}\right)^k - 1\right) \\ &= 2n^3\left(\frac{7}{8}\right)^k + \left(\frac{A}{3} - 1\right)n^2\left(\frac{7}{4}\right)^k - \frac{A}{3}n^2. \end{aligned}$$

Cette fonction est minimale en

$$k_0 = \log_2 n + 1 + \left(\log_2 3 - \log_2 \ln \frac{7}{4} + \log_2 \ln \frac{8}{7} - \log_2(A-3)\right). \tag{2.3}$$

Pour $A = 15$, on obtient $k_0 \approx \log_2 n - 3,067257711$. Donc le niveau récursif seuil optimal, k, est nul pour $n = 2, 4, 8$, puis $k = i$ pour $n = 2^{i+3}$.

Le tableau 2.1 compare le nombre d'opérations arithmétiques en fonction de l'ordre des matrices et du niveau récursif où le changement intervient.

		Nombre de niveaux récursifs de l'algorithme de Winograd					
n	Classique	1	2	3	4	5	6
4	**112**	144	214				
8	**960**	1024	1248	1738			
16	7936	**7680**	8128	9696	13126		
32	64512	59392	**57600**	60736	71712	95722	
64	520192	466944	431104	**418560**	440512	517344	685414

TAB. 2.1 – *Nombre d'opérations arithmétiques dans le produit de deux matrices d'ordre* n

Pour n **quelconque :** Plusieurs techniques, décrites dans [69, 70] peuvent être appliquées pour obtenir des dimensions paires afin d'effectuer le découpage de l'algorithme de Winograd. Suite à la discussion [70, §4.1], nous utilisons le pelage dynamique (*dynamic peeling*), qui consiste à découper des bandes de rang 1 pour rendre les dimensions paires. Après le calcul de l'algorithme de Winograd, des mises à jour de rang 1 sont nécessaires. Il s'agit de produits matrice-vecteur et de produits tensoriels.

Nous ne donnons pas de formule explicite pour le nombre d'opérations arithmétiques dans le cas général, mais si $n = 2^i + p$ avec $p < 2^i$, alors le seuil k_0 calculé pour $n = 2^i$ avec la formule 2.3 sera toujours meilleur qu'un seuil supérieur.

Ainsi, cet algorithme en cascade est préférable pour le calcul en pratique. Ce phénomène est d'ailleurs amplifié par le fait que les opérations arithmétiques effectuées dans le produit classique sont plus rapides lorsqu'elles sont groupées en opérations des BLAS (grâce à la meilleure gestion des accès mémoire). Par conséquent, le nombre optimal de niveaux récursifs à effectuer dépend de paramètres liés aux architectures et doit donc être déterminé expérimentalement. On peut le décrire par le paramètre suivant : la dimension k_{Winograd} pour laquelle un niveau récursif de l'algorithme de Winograd est aussi rapide que l'algorithme classique. Le nombre optimal de niveaux l pour des matrices d'ordre n est alors donné par la formule :

$$l = \left\lfloor \log_2 \frac{n}{k_{\text{Winograd}}} \right\rfloor + 1. \tag{2.4}$$

Au passage, l'équation (2.3) permet de calculer la constante $C_{2,8074}$ pour les algorithmes de Winograd et Strassen en prenant $k = \log_2 n$. Quand n est une puissance de 2, on obtient

$$T(n, \log_2 n) = 7^{\log_2 n}\left(1 + \frac{A}{3}\right) - \frac{A}{3}n^2 = \left(1 + \frac{A}{3}\right) n^{2,8074}$$

Ainsi, pour l'algorithme de Winograd, $C_{2,8074} = 6$ et pour celui de Strassen, $C_{2,8074} = 7$.

2.2.2 Étude de l'allocation mémoire

Nous rappelons d'abord le principe de l'algorithme pour le calcul du produit $C \leftarrow A \times B$. Un niveau récursif de l'algorithme consiste à effectuer les découpages suivants sur A et B :

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \text{ et } B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

Puis les 22 opérations suivantes sont effectuées :

– 8 additions :

$$\begin{aligned} S_1 &\leftarrow A_{21} + A_{22} & T_1 &\leftarrow B_{12} - B_{11} \\ S_2 &\leftarrow S_1 - A_{11} & T_2 &\leftarrow B_{22} - T_1 \\ S_3 &\leftarrow A_{11} - A_{21} & T_3 &\leftarrow B_{22} - B_{12} \\ S_4 &\leftarrow A_{12} - S_2 & T_4 &\leftarrow T_2 - B_{21} \end{aligned}$$

– 7 multiplications, par appels récursifs :

$$\begin{aligned} P_1 &\leftarrow A_{11} \times B_{11} & P_5 &\leftarrow S_1 \times T_1 \\ P_2 &\leftarrow A_{12} \times B_{21} & P_6 &\leftarrow S_2 \times T_2 \\ P_3 &\leftarrow S_4 \times B_{22} & P_7 &\leftarrow S_3 \times T_3 \\ P_4 &\leftarrow A_{22} \times T_4 \end{aligned}$$

– 7 additions finales :

$$\begin{aligned} U_1 &\leftarrow P_1 + P_2 & U_5 &\leftarrow U_4 + P_3 \\ U_2 &\leftarrow P_1 + P_6 & U_6 &\leftarrow U_3 - P_4 \\ U_3 &\leftarrow U_2 + P_7 & U_7 &\leftarrow U_3 + P_5 \\ U_4 &\leftarrow U_2 + P_5 \end{aligned}$$

– Le résultat est le suivant : $C = \begin{bmatrix} U_1 & U_5 \\ U_6 & U_7 \end{bmatrix}$

La figure 2.1 donne le graphe de dépendance des tâches de l'algorithme de Winograd. A partir de ce graphe, on peut retrouver l'ordonnancement de ces tâches donné dans [29], qui minimise le nombre d'allocations temporaires pour l'opération $C \leftarrow A \times B$. Nous le rappelons dans le tableau 2.2. Cet ordonnancement n'exige que l'allocation de deux matrices temporaires X_1 et X_2 de dimensions $m/2 \times \max(n,k)/2$ et $k/2 \times n/2$. Dans le cas de matrices carrées, ces allocations temporaires additionnées pour tous les niveaux récursifs sont bornées par

$$2\sum_{i=1}^{\log n} \left(\frac{n}{2^i}\right)^2 < \frac{2}{3}n^2.$$

Considérons maintenant l'opération $C \leftarrow A \times B + \beta C$. Une première méthode naïve est d'appliquer l'ordonnancement précédent pour calculer $A \times B$ dans une matrice temporaire.

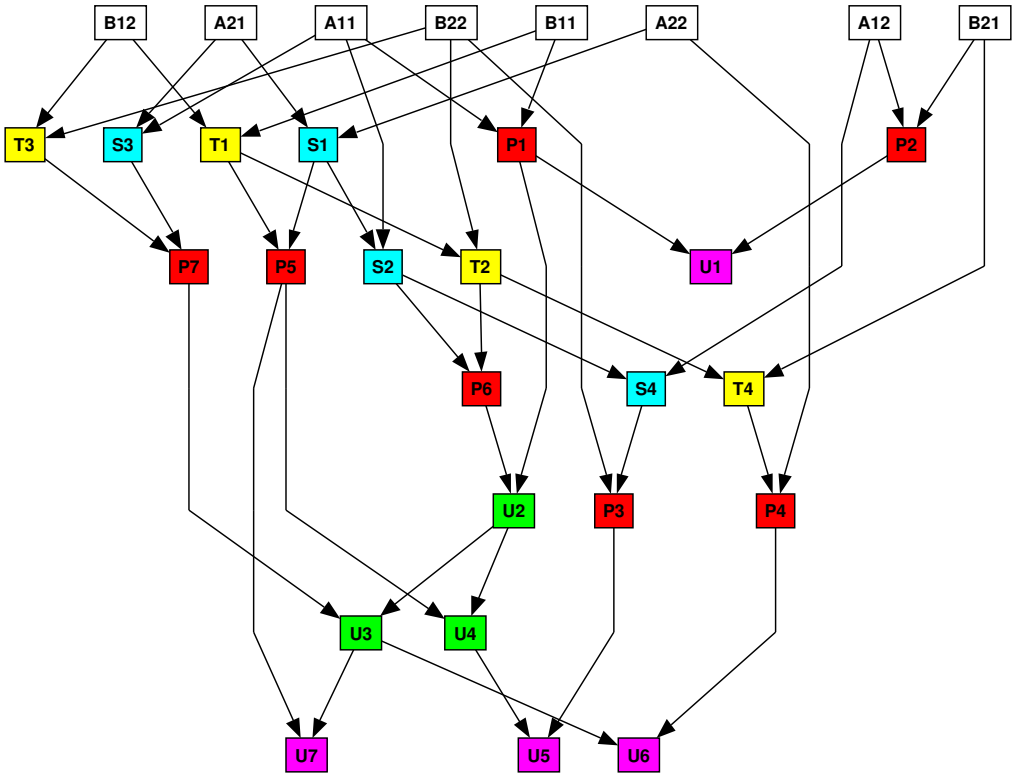


FIG. 2.1 – *Graphe de dépendance des tâches dans un appel récursif de l'algorithme de Winograd*

#	operation	loc.	#	operation	loc.	#	operation	loc.
1	$S_3 = A_{11} - A_{21}$	X_1	9	$P_6 = S_2T_2$	C_{12}	17	$U_5 = U_4 + P_3$	C_{12}
2	$T_3 = B_{22} - B_{12}$	X_2	10	$S_4 = A_{12} - S_2$	X_1	18	$T_4 = T_2 - B_{21}$	X_2
3	$P_7 = S_3T_3$	C_{21}	11	$P_3 = S_4B_{22}$	C_{11}	19	$P_4 = A_{22}T_4$	C_{11}
4	$S_1 = A_{21} + A_{22}$	X_1	12	$P_1 = A_{11}B_{11}$	X_1	20	$U_6 = U_3 - P_4$	C_{21}
5	$T_1 = B_{12} - B_{11}$	X_2	13	$U_2 = P_1 + P_6$	C_{12}	21	$P_2 = A_{12}B_{21}$	C_{11}
6	$P_5 = S_1T_1$	C_{22}	14	$U_3 = U_2 + P_7$	C_{21}	22	$U_1 = P_1 + P_2$	C_{11}
7	$S_2 = S_1 - A_{11}$	X_1	15	$U_4 = U_2 + P_5$	C_{12}			
8	$T_2 = B_{22} - T_1$	X_2	16	$U_7 = U_3 + P_5$	C_{22}			

TAB. 2.2 – *Ordonnancement de l'algorithme de Winograd pour l'opération* $C \leftarrow A \times B$

Puis le résultat est additionné à βC. Dans le cas de matrices carrées, ceci impliquerait $(1 + 2/3)n^2$ allocations temporaires.

On peut cependant réduire le nombre d'allocations temporaires en incorporant l'accumulation $+\beta C$ au sein de l'algorithme. Nous rappelons dans le tableau 2.3, l'ordonnancement de [70] n'utilisant que trois blocs temporaires X_1, X_2 et X_3, de dimensions $m/2 \times n/2$, $m/2 \times k/2$ et $k/2 \times n/2$. On se référera à cet article pour la preuve de l'optimalité de cet ordonnancement, grâce au jeu des galets (*pebble game*).

#	operation	loc.	#	operation	loc.
1	$S_1 = A_{21} + A_{22}$	X_2	12	$P_2 = \alpha A_{12}B_{21} + C_{11}$	C_{11}
2	$T_1 = B_{12} - B_{11}$	X_3	13	$S_4 = A_{12} - S_2$	X_2
3	$P_5 = \alpha S_1 T_1$	X_1	14	$T_4 = T_2 - B_{21}$	X_3
4	$C_{22} = P_5 + \beta C_{22}$	C_{22}	15	$U_5 = \alpha S_4 B_{22} + U_4$	C_{12}
5	$C_{12} = P_5 + \beta C_{12}$	C_{12}	16	$P_4 = \alpha A_{22} T_4 + \beta C_{21}$	C_{21}
6	$S_2 = S_1 - A_{11}$	X_2	17	$S_3 = A_{11} - A_{21}$	X_2
7	$T_2 = B_{22} - T_1$	X_3	18	$T_3 = B_{22} - B_{12}$	X_3
8	$P_1 = \alpha A_{11} B_{11}$	X_1	19	$U_3 = \alpha S_3 T_3 + U_2$	X_1
9	$C_{11} = P_1 + \beta C_{11}$	C_{11}	20	$U_7 = U_7 + U_3$	C_{22}
10	$U_2 = S_2 T_2 + P_1$	X_1	21	$U_6 = U_6 + U_3$	C_{21}
11	$U_4 = C_{12} + U_2$	C_{12}			

TAB. 2.3 – *Ordonnancement pour* $C \leftarrow \alpha AB + \beta C$

Le total des allocations temporaires de cet ordonnancement est borné par

$$3 \sum_{i=1}^{\log n} \left(\frac{n}{2^i} \right)^2 < n^2.$$

Nous résumons dans le tableau 2.4 les propriétés de différents algorithmes rapides, concernant le nombre d'allocations temporaires et le nombre d'opérations de blocs. Il montre que l'ordonnancement du tableau 2.3 est le meilleur en ce qui concerne l'allocation mémoire, sans surcoût dans le nombre d'opérations effectuées.

2.2.3 Traitement des dimensions impaires

L'algorithme de Winograd repose sur un découpage récursif des opérandes. Par conséquent, il n'est défini que pour des matrices carrées dont l'ordre est une puissance de deux. En se limitant à l niveaux récursif de l'algorithme, il est alors bien défini, pour le produit de matrices $m \times k$ par $k \times n$ si 2^l divise à la fois m, k et n. Dans le cas général un traitement spécifique doit être appliqué pour gérer les dimensions impaires. On trouvera dans [70, §4] une étude détaillée

Algorithme	Operation	# d'Add.	# de blocs. temp. alloués	Total de mémoire temp. allouée
Strassen	$C \leftarrow A \times B$	18	2	$2/3n^2$
Winograd	$C \leftarrow A \times B$	15	2	$2/3n^2$
Strassen	$C \leftarrow \alpha A \times B + \beta C$	22	2+4	$(2/3+1)n^2$
Winograd	$C \leftarrow \alpha A \times B + \beta C$	19	2+4	$(2/3+1)n^2$
Huss-Ledermann & Al.	$C \leftarrow \alpha A \times B + \beta C$	19	3+0	n^2

TAB. 2.4 – *Comparaison de différents ordonnancements et algorithmes de multiplication rapides*

sur le sujet : le pavage (*padding*) consistant à compléter les dimensions impaires par des zéros, et le pelage (*peeling*) consistant à découper les dimensions impaires. Chacune de ces méthodes pouvant être statique si elle est appliquée avant le découpage récursif, ou dynamique si elle est appliquée au cours de chacun des appels.

En suivant les conclusions de cet article, nous avons opté pour la technique du pelage dynamique : à chaque niveau récursif, chaque opérande est découpée en un bloc de dimensions paires et un ou deux éventuels vecteurs formant la dernière colonne ou la dernière ligne impaire. Les blocs pairs sont traités par l'algorithme récursif de Winograd, et une mise à jour est ensuite appliquée avec les éventuels vecteurs. Les opérations de mise à jours sont des produits matrice-vecteur, des produits scalaires, ou des produits tensoriels.

2.2.4 Contrôle du débordement

Afin de l'insérer au sein de la structure en cascade décrite dans la partie 2.1.1, il faut envisager d'effectuer l'algorithme de Winograd aussi bien sur le corps fini que sur $\mathbb{Z}$, en employant la réduction modulaire différée. Il faut donc pouvoir contrôler la croissance des valeurs entières manipulées au cours des appels récursifs pour éviter les débordements. En effet, les bornes données en partie 2.1.1 pour le produit scalaire ne s'appliquent plus pour l'algorithme de Winograd. Nous donnons ici la borne optimale pour ce calcul en montrant que la plus grande valeur intermédiaire calculée intervient dans le pire des cas dans les calculs récursifs de P_6. Nous avions donné ce résultat pour l'opération $C \leftarrow A \times B$ dans [34, theorème 3.1], et nous le généralisons maintenant pour le calcul de $C \leftarrow A \times B + \beta C$.

Theorème 2.1. *Soit $A \in \mathbb{Z}^{m\times k}$, $B \in \mathbb{Z}^{k\times n}$ $C \in \mathbb{Z}^{m\times n}$, trois matrices et $\beta \in \mathbb{Z}$ avec $m_A \leq a_{i,j} < M_A$, $m_B \leq b_{i,j} < M_B$ et $m_C \leq c_{i,j} < M_C$. Et de plus $0 \leq -m_A \leq M_A$, $0 \leq -m_B \leq M_B$, $0 \leq -m_C \leq M_C$, $M_C \leq M_B$ et $|\beta| \leq M_A, M_B$. Alors toutes les valeurs intermédiaires z impliquées dans le calcul de $A \times B + \beta C$ par $l \geq 1$ niveaux récursifs de l'algorithme de Winograd vérifient :*

$$|z| \leq \left(\frac{1+3^l}{2}M_A + \frac{1-3^l}{2}m_A\right)\left(\frac{1+3^l}{2}M_B + \frac{1-3^l}{2}m_B\right)\left\lfloor\frac{k}{2^l}\right\rfloor.$$

De plus, cette borne est optimale.

Démonstration. Nous allons montrer que le pire cas a lieu dans le calcul récursif de P_6.

Nous commençons par le cas $k = 2^l q$, où l est le nombre de niveaux récursifs, avant de le généraliser pour k quelconque. Nous terminons la preuve en donnant une instance de calcul pour laquelle la borne est atteinte, ce qui prouve son optimalité.

Quelques propriétés sur les suites du type $2u - v$

Considérons les suites (u_n) et (v_n), définies récursivement par :

$$\begin{cases} u_{l+1} = 2u_l - v_l \\ v_{l+1} = 2v_l - u_l \\ u_0 \leq 0 \\ v_0 \geq 0 \end{cases}$$

Comme

$$\begin{cases} u_{l+1} + v_{l+1} = u_l + v_l = \cdots = u_0 + v_0 \\ v_{l+1} - u_{l+1} = 3(v_l - u_l) = \cdots = 3^{l+1}(v_0 - u_0) \end{cases},$$

on en déduit

$$\begin{cases} u_l = u_0 \frac{(1+3^l)}{2} + v_0 \frac{(1-3^l)}{2} \\ v_l = v_0 \frac{(1+3^l)}{2} + u_0 \frac{(1-3^l)}{2} \end{cases}.$$

Ainsi, nous obtenons les propriétés suivantes :

$$u_l \leq 0 \text{ and } v_l \geq 0 \tag{2.5}$$

$$u_l \text{ est décroissante et } v_l \text{ est croissante} \tag{2.6}$$

$$v_l > -u_l \text{ si } v_0 > -u_0 \tag{2.7}$$

Nous définissons maintenant v^A et v^B, deux suites du type v en posant $u_0^A = m_A$, $v_0^A = M_A$, $u_0^B = m_B$ et $v_0^B = M_B$. Nous définissons également $t_j = \frac{1+3^j}{2}$ et $s_j = \frac{1-3^j}{2}$. Ainsi $t_j + s_j = 1$ et $t_j - s_j = 3^j$. La propiété suivante est vérifiée :

$$(2M_A - m_A)t_j + (2m_A - M_A)s_j = M_A t_{j+1} + m_A s_{j+1} = v_{j+1}^A. \tag{2.8}$$

Notations

Soit

$$b_l = \left(\frac{1+3^l}{2}M_A + \frac{1-3^l}{2}m_A\right)\left(\frac{1+3^l}{2}M_B + \frac{1-3^l}{2}m_B\right)\left\lfloor \frac{k}{2^l} \right\rfloor.$$

La suite $(b_l)_{l>0}$ est croissante d'après (2.6).

Nous voulons borner les valeurs intervenant dans le calcul de chacune des opérations de l'algorithme de Winograd. Tout d'abord, remarquons que le résultat du produit matriciel est indépendant de la façon dont il est calculé. Il est donc toujours borné par

$$k\text{max}(|m_A|, |M_A|)\text{max}(|m_B|, |M_B|) + \beta\text{max}(|m_C|, |M_C|) \leq (k+1)M_A M_B.$$

Or cette valeur est toujours inférieure à b_1 pour $k \geq 1$ et donc inférieure à b_l $\forall l \geq 1$. Par conséquent, les coefficients des blocs U_1, U_5, U_6 et U_7 vérifient toujours la borne.

Il reste maintenant à montrer que chacun des neuf résultats intermédiaires suivants est borné par b_l.

$$\begin{aligned}
P_1 &= A_{11} \times B_{11} \\
P_2 &= A_{12} \times B_{21} + \beta C_{11} \\
P_3 &= (A_{12} + A_{11} - A_{21} - A_{22}) \times B_{22} \\
P_4 &= A_{22} \times (B_{22} + B_{11} - B_{21} - B_{12}) + \beta(C_{22} - C_{12} - C_{21}) \\
P_5 &= (A_{21} + A_{22}) \times (B_{12} - B_{11}) + \beta C_{12} \\
P_6 &= (A_{21} + A_{22} - A_{11}) \times (B_{22} + B_{11} - B_{12}) \\
P_7 &= (A_{11} - A_{21}) \times (B_{22} - B_{12}) + \beta(C_{22} - C_{12}) \\
U_2 &= (A_{21} + A_{22} - A_{11}) \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} \\
U_3 &= A_{22} \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} + \beta(C_{22} - C_{12}) \\
U_4 &= (A_{21} + A_{22}) \times B_{22} + A_{11} \times (B_{12} - B_{22}) + \beta C_{12}
\end{aligned}$$

Nous allons montrer que le plus grand calcul intermédiaire possible est toujours lors du calcul de P_6. Considérons l niveaux récursifs indexés par j : $j = l$ est le premier découpage des matrices en quatre blocs et $j = 0$ correspond au dernier niveau, pour lequel le produit matriciel est effectué par la multiplication classique.

Nous définissons aussi les notations suivantes :

- $M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(X)$ est une borne supérieure sur les résultats intermédiaires du calcul de $X = A \times B + \beta C$ avec j niveaux récursifs et $m_A \leq a_{i,j} \leq M_A$, $m_B \leq b_{i,j} \leq M_B$ et $m_C \leq c_{i,j} \leq M_C$. k est la dimension commune de A et B
- $M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = \max_X M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(X)$.
- $M(X)\frac{k}{2^j}$ pour $M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(X)$.

Les formules suivantes correspondent aux sept appels récursifs :

$$M^{j+1,k}_{m_A,M_A,m_B,M_B,m_C,M_C} =$$

$$\max \left(\begin{array}{l}
M(P_1)\frac{k}{2^j} = M^{j,\frac{k}{2}}_{m_A,M_A,m_B,M_B,0,0} \\
M(P_2)\frac{k}{2^j} = M^{j,\frac{k}{2}}_{m_A,M_A,m_B,M_B,m_C,M_C} \\
M(P_3)\frac{k}{2^j} = M^{j,\frac{k}{2}}_{2m_A-2M_A,2M_A-2m_A,m_B,M_B,0,0} \\
M(P_4)\frac{k}{2^j} = M^{j,\frac{k}{2}}_{m_A,M_A,2m_B-2M_B,2M_B-2m_B,m_C-2M_C,M_C-2m_C} \\
M(P_5)\frac{k}{2^j} = M^{j,\frac{k}{2}}_{2m_A,2M_A,m_B-M_B,M_B-m_B,m_C,M_C} \\
M(P_6)\frac{k}{2^j} = M^{j,\frac{k}{2}}_{2m_A-M_A,2M_A-m_A,2m_B-M_B,2M_B-m_B,0,0} \\
M(P_7)\frac{k}{2^j} = M^{j,\frac{k}{2}}_{m_A-M_A,M_A-m_A,m_B-M_B,M_B-m_B,m_C-M_C,M_C-m_C} \\
M(U_2)\frac{k}{2^j} \\
M(U_3)\frac{k}{2^j} \\
M(U_4)\frac{k}{2^j}
\end{array} \right) \tag{2.9}$$

Par ailleurs, l'algorithme classique est utilisé pour $j = 0$:

$$M^{0,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = \max \begin{pmatrix} M_A M_B k + \beta M_C \\ -m_A M_B k - \beta m_C \\ -M_A m_B k - \beta m_C \end{pmatrix} \tag{2.10}$$

Quelques invariants

Lemme 2.1. *Les invariants suivants sont vérifiés au cours des appels récursifs :*

$$0 \leq -m_A \leq M_A,\ 0 \leq -m_B \leq M_B,\ 0 \leq -m_C \leq M_C \tag{2.11}$$
$$m_C \geq m_B \text{ et } M_C \leq M_B \tag{2.12}$$
$$M_C - m_C \leq M_B - m_B \tag{2.13}$$

Démonstration. D'après l'équation (2.9), on obtient les invariants (2.11) et (2.12). Puis l'invariant (2.13) est une conséquence de (2.11) et (2.12). □

Récurrence pour $k = 2^l q$

Soit HR_j l'hypothèse de récurrence suivante :
Si les invariants (2.11,2.12, 2.13) sont vérifiés, alors

$$M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = [v^A_j][v^B_j]\frac{k}{2^j}. \tag{2.14}$$

Supposons que les invariants précédents sont vérifiés et que HR_j est vraie. Nous allons montrer que le maximum de (2.9) est atteint pour le calcul de P_6, ce qui permet de vérifier HR_{j+1}.

Les conditions sur m_A, M_A, m_B, M_B, m_C et M_C sont vérifiées pour chacun des sept appels récursifs. Nous pouvons donc appliquer HR_j à chacun d'eux.

– Pour $P_1 = A_{11} \times B_{11}$:

$$\begin{aligned} M(P_6) - M(P_1) &= [(2M_A - m_A)t_j + (2m_A - M_A)s_j] \times \\ &\quad [(2M_B - m_B)t_j + (2m_B - M_B)s_j] - v^{A_{11}}_j v^{B_{11}}_j \\ &= v^A_{j+1} v^B_{j+1} - v^{A_{11}}_j v^{B_{11}}_j \\ &\geq v^A_{j+1} v^B_{j+1} - v^A_j v^B_j \end{aligned}$$

Et comme v^A et v^B sont croissantes et positives, il vient $M(P_6) \geq M(P_1)$.

– Pour $P_2 = A_{12} \times B_{21} + \beta C_{11}$: de la même manière, $M(P_6) \geq M(P_2)$.

– Pour $P_3 = (A_{12} + A_{11} - A_{21} - A_{22}) \times B_{22}$:

$$\begin{aligned} M(P_6) - M(P_3) &= v_{j+1}^A v_{j+1}^B - v_j^{A_{11}+A_{12}-A_{21}-A_{22}} v_j^{B_{22}} \\ &= v_{j+1}^A v_{j+1}^B - [(2M_A - 2m_A)t_j + (2m_A - 2M_A)s_j]v_j^B \\ &= v_{j+1}^A v_{j+1}^B - (v_{j+1}^A - m_A t_j - M_A s_j)v_j^B \text{ d'après (2.8)} \\ &= v_{j+1}^A[v_{j+1}^B - v_j^B] - u_j^A v_j^B \\ &\geq v_{j+1}^A[v_{j+1}^B - v_j^B] - v_{j+1}^A v_j^B \text{ d'après (2.7)} \\ &\geq v_{j+1}^A[v_{j+1}^B - 2v_j^B] \\ &\geq v_{j+1}^A 3^j[M_B - m_B] \geq 0 \end{aligned}$$

– Pour $P_4 = A_{22} \times (B_{22} + B_{11} - B_{21} - B_{12}) + \beta(C_{22} - C_{12} - C_{21})$: de la même manière,

$$M(P_6) - M(P_4) = v_{j+1}^A v_{j+1}^B - v_j^{A_{22}} v_j^{B_{22}+B_{11}-B_{12}-B_{21}} \geq 0$$

– Pour $P_5 = (A_{21} + A_{22}) \times (B_{12} - B_{11}) + \beta C_{12}$:

$$\begin{aligned} M(P_6) - M(P_5) &= v_{j+1}^A v_{j+1}^B - v_j^{A_{21}+A_{22}} v_j^{B_{12}-B_{11}} \\ &= v_{j+1}^A v_{j+1}^B - 2v_j^A \left[v_j^B - u_j^B\right] \\ &= \left[2v_j^A - u_j^A\right] v_{j+1}^B - v_j^A \left[v_{j+1}^B - u_j^B\right] \\ &= v_j^A v_{j+1}^B - u_j^A v_{j+1}^B + v_j^A u_j^B \\ &= v_j^A \left[v_{j+1}^B + u_j^B\right] - u_j^A v_{j+1}^B \\ &= v_j^A \left[2v_j^B\right] - u_j^A v_{j+1}^B \end{aligned}$$

et comme $u_j^A \leq 0 \leq v_j^A, v_j^B, v_{j+1}^B$, il vient $M(P_6) - M(P_5) \geq 0$.

– Pour $P_7 = (A_{11} - A_{21}) \times (B_{22} - B_{12}) + \beta(C_{22} - C_{12})$: en utilisant P_5,

$$\begin{aligned} M(P_5) - M(P_7) &= v_j^{A_{21}+A_{22}} v_j^{B_{12}-B_{11}} - v_j^{A_{11}-A_{21}} v_j^{B_{22}-B_{12}} \\ &= [2M_A t_j + 2m_A s_j - (M_A - m_A)t_j - (m_A - M_A)s_j] \times \\ &\quad [(M_B - m_B)t_j + (m_B - M_B)s_j] \\ &= [(M_A + m_A)(t_j + s_j)]\,[(M_B - m_B)(t_j - s_j)] \\ &\geq 0. \end{aligned}$$

– Les coefficients des blocs U_1, U_5, U_6 et U_7 sont bornés par $kM_AM_B + \beta M_C$ et sont donc inférieurs en valeur absolue à ceux de P_6.

– Pour $U_2 = (A_{21} + A_{22} - A_{11}) \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11}$:

$$\forall x \text{ coefficient de } U_2, |x| \leq \max \begin{pmatrix} (2M_A - m_A)(M_B - m_B) + 2M_AM_B \\ (-2m_A + M_A)(M_B - m_B) - 2m_AM_B \\ (-2m_A + M_A)(M_B - m_B) - 2M_Am_B \end{pmatrix} k/2^j. \tag{2.15}$$

Or $2M_A - m_A - (-2m_A + M_A) = M_A + m_A \geq 0$ et $0 \leq -m_A \leq M_A$, donc l'équation 2.15 se réduit à $|x| \leq (2M_A - m_A)(M_B - m_B) + 2M_AM_B$.

$$\begin{aligned} M(P_6) - M(U_2) &\geq (2M_A - m_A)(2M_B - m_B) - (2M_A - m_A)(M_B - m_B) \\ &\quad -2M_A M_B \\ &= (2M_A - m_A)M_B - 2M_A M_B \\ &= -m_A M_B \geq 0 \end{aligned}$$

– Pour $U_3 = A_{22} \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} + \beta(C_{22} - C_{12})$: de la même manière,

$$\forall x \text{ coeff. de } U_3, |x| \leq \max \begin{pmatrix} (M_A(M_B - m_B) + 2M_A M_B)k/2^j + |\beta|(M_C - m_C) \\ (M_A(M_B - m_B) - 2m_A M_B)k/2^j + |\beta|(M_C - m_C) \\ (M_A(M_B - m_B) - 2M_A m_B)k/2^j + |\beta|(M_C - m_C) \end{pmatrix}.$$

Le maximum est toujours atteint par le premier argument et comme $k/2^j \geq 1$, $\beta \leq M_A - m_A$ et $M_C - m_C \leq M_B - m_B$, il vient :

$$\begin{aligned} |x| &\leq (M_A(M_B - m_B) + 2M_A M_B)k/2^j + \beta(M_C - m_C) \\ &\leq (2M_A - m_A)(M_B - m_B) + 2M_A M_B)k/2^j \\ &\leq M(U2) \leq M(P6). \end{aligned}$$

– Pour $U_4 = (A_{21} + A_{22}) \times B_{22} + A_{11} \times (B_{12} - B_{22}) + \beta C_{12}$: de la même manière,

$$\forall x \text{ coefficient de } U_4, |x| \leq (M_A(M_B - m_B) + 2M_A M_B)k/2^j + |\beta| M_C.$$

Comme $M_C \leq M_B - m_B$, $-m_A \leq M_A$ et $-m_B \leq M_B$, il vient

$$M(U_4) \leq M(U_3) \leq M(P_6).$$

Ainsi

$$M^{j+1,k}_{m_A,M_A,m_B,M_B} = M(P_6)\frac{k}{2^{j+1}} = v^A_{j+1} v^B_{j+1} \frac{k}{2^{j+1}},$$

et HR_{j+1} est vérifiée.

Pour l'initialisation de la récurrence ($j = 1$), les produits des blocs sont effectués par l'algorithme classique. D'après (2.9) et (2.10), on obtient :

$$\begin{aligned} M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_1) &= M_A M_B k/2 \\ M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_2) &= M_A M_B k/2 + |\beta| M_C \\ M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_3) &= 2(M_A - m_A)M_B k/2 \\ M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_4) &= 2M_A(M_B - m_B)k/2 + |\beta|(2M_C - m_C) \\ M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_5) &= 2M_A(M_B - m_B)k/2 + |\beta| M_C \\ M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_6) &= (2M_A - m_A)(2M_B - m_B)k/2 \\ M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_7) &= (M_A - m_A)(M_B - m_B)k/2 + |\beta|(M_C - m_C) \\ M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(U_2) &= (2M_A - m_A)(M_B - m_B)k/2 + 2M_A M_B k/2 \\ M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(U_3) &= M_A(M_B - m_B)k/2 + 2M_A M_B k/2 + |\beta|(M_C - m_C) \\ M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(U_4) &= 2M_A M_B k/2 + M_A(M_B - m_B)k/2 + |\beta| M_C \end{aligned}$$

A nouveau nous allons montrer que $M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_6)$ atteint la valeur la plus grande, en utilisant les invariants (2.11), (2.12) et (2.13) et le fait que $|\beta| \leq M_A, M_B$.

C'est évident pour P_1 et P_2.

– Pour P_3 :

$$\begin{aligned} M^{1,k}_{m_A,\ldots}(P_6) - M^{1,k}_{m_A,\ldots}(P_3) &= ((2M_A - m_A)(2M_B - m_B) - 2(M_A - m_A)M_B)k/2 \\ &= (2M_A M_B k - (2M_A - m_A)m_B)k/2 \geq 0 \end{aligned}$$

– Pour P_4 : Comme $-|\beta|(2M_C - m_C) \geq -M_A(2M_B - m_B)$, on obtient

$$\begin{aligned} M^{1,k}_{m_A,\ldots}(P_6) - M^{1,k}_{m_A,\ldots}(P_4) &= ((2M_A - m_A)(2M_B - m_B) - 2M_A(M_B - m_B))k/2 \\ &\quad -|\beta|(M_C - 2m_C) \\ &\geq (M_A - m_A)(2M_B - m_B) - 2M_A(M_B - m_B) \\ &= m_A(m_B - 2M_B) \geq 0 \end{aligned}$$

– Pour P_5 : $M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_5) \leq M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_4)$

– Pour P_7 :

$$\begin{aligned} M^{1,k}_{m_A,\ldots}(P_6) - M^{1,k}_{m_A,\ldots}(P_7) &= ((2M_A - m_A)(2M_B - m_B) \\ &\quad -(M_A - m_A)(M_B - m_B)k/2 - |\beta|(M_C - m_C) \\ &\geq M_A(2M_B - m_B) + (M_A - m_A)M_B \\ &\quad -M_A(M_B - m_B) \\ &\geq (2M_A - m_A)M_B \geq 0 \end{aligned}$$

– Pour U_2, U_3 et U_4 : avec le même argument que pour le cas où j est quelconque.

HR_1 est donc vérifiée.

Avec k quelconque

Soit l tel que $2^l d \leq k < 2^l(d+1)$ ($d = \left\lfloor \frac{k}{2^l} \right\rfloor$). Les opérations de mise à jour, de pelage dynamique, sont des produits matrice-vecteur, des produits scalaires et des produits tensoriels. Ces calculs n'impliquent donc pas de valeurs intermédiaires supérieures en valeur absolue à $kM_AM_B + |\beta|M_C \leq (k+1)M_AM_B$.

Nous allons montrer que cette borne est toujours inférieure à celle de l'algorithme de Winograd utilisée pour les blocs pairs :

$$\forall l \geq 1 \;\; 2^l(d+1)M_AM_B \leq v_l^A v_l^B \left\lfloor \frac{k}{2^l} \right\rfloor$$

(car $(k+1)M_AM_B \leq 2^l(d+1)M_AM_B$).

– Pour $l = 1$, l'inéquation est vérifiée : $2M_AM_B(d+1) \leq (2M_A - m_A)(2M_B - m_B)d$ (car $d \geq 1$).

– Supposons qu'elle est vérifiée pour $l \geq 1$ et montrons la pour $l+1$:

$$\begin{aligned} v_{l+1}^A v_{l+1}^B \left\lfloor \frac{k}{2^{l+1}} \right\rfloor &= [(2M_A - m_A)t_l + (2m_A - M_A)s_l] \\ &\times [(2M_B - m_B)t_l + (2m_B - M_B)s_l]d \\ &\geq 2[M_A t_l + m_A s_l][M_B t_l + m_B s_l]2d \\ &\geq v_l^A v_l^B \left\lfloor \frac{k}{2^l} \right\rfloor \\ &\geq 2(2^l M_A M_B(2d+1)) \\ &\geq 2^{l+1} M_A M_B(d+1)). \end{aligned}$$

Par récurrence, la borne de l'équation (2.14) est vérifiée pour k quelconque.

Optimalité de la borne

Nous construisons simplement une suite de paires de matrices carrées A_l et B_l d'ordre 2^l pour lesquelles l niveaux récursifs de l'algorithme de Winograd impliquent des calculs intermédiaires égaux en valeur absolue à la borne (2.14).

Soit J_l la matrice carrée d'ordre 2^l dont les coefficients sont égaux à 1. Soit $(A_l)_{l\in\mathbb{N}^*}$ et $(B_l)_{l\in\mathbb{N}^*}$ définies récursivement comme suit :

$$\begin{cases} A_1 = \begin{bmatrix} m_A & 0 \\ M_A & M_A \end{bmatrix}, \quad B_1 = \begin{bmatrix} M_B & m_B \\ 0 & M_B \end{bmatrix} \\ \\ A_{l+1} = \begin{bmatrix} \overline{A_l} & 0 \\ A_l & A_l \end{bmatrix}, \quad B_{l+1} = \begin{bmatrix} B_l & \overline{B_l} \\ 0 & B_l \end{bmatrix} \end{cases}$$

où $\overline{A_l} = (M_A + m_A)J_l - A_l$ et $\overline{B_l} = (M_B + m_B)J_l - B_l$.

Comme à chaque niveau récursif, le calcul de $P_6 = (A_{21} + A_{22} - A_{11}) \times (B_{22} + B_{11} - B_{12})$ implique la plus grande valeur intermédiaire possible, définissons :

$$S(A_l) = (A_l)_{2,1} + (A_l)_{2,2} - (A_l)_{1,1} = 2A_{l-1} - \overline{A_{l-1}} = 3A_{l-1} - (M_A + m_A)J_{l-1}.$$

Par ailleurs, $S(J_l) = J_{l-1}$ et S est linéaire. Ainsi, en appliquant P_6 récursivement l fois :

$$S(S(\ldots(S(A_l)))) = S^l(A_l) = 3^{l-1}S(A_1) - (M_A + m_A)\left(\sum_{k=0}^{l-2} 3^k\right) J_1.$$

$S(A_1) = [2M_A - m_A]$ et $J_1 = [1]$ impliquent :

$$S^l(A_l) = 3^{l-1}(2M_A - m_A) - \frac{3^{l-1}-1}{3-1}(M_A + m_A) = \frac{1+3^l}{2}M_A + \frac{1-3^l}{2}m_A.$$

De même pour B_l :

$$S^l(B_l) = \frac{1+3^l}{2}M_B + \frac{1-3^l}{2}m_B.$$

L'ordre de A_l et B_l est $k = 2^l$, donc $\left\lfloor \frac{k}{2^l} \right\rfloor = 1$. Ainsi, le calcul de $A_l \times B_l$ avec l niveaux récursifs de l'algorithme de Winograd implique des valeurs intermédiaires égales à $v_l^{A_l} v_l^{B_l} \left\lfloor \frac{k}{2^l} \right\rfloor$, ce qui prouve l'optimalité de la borne. □

Remarque 2.1. Cette borne ne dépend pas de l'éventuel terme supplémentaire $+\beta C$ car c'est le calcul de P_6 qui domine la croissance des coefficients.

Si les éléments du corps fini sont représentés par des entiers compris entre 0 et $p-1$ (représentation positive), la borne devient :

Corollaire 2.2 (Représentation modulaire positive)**.** *Avec $a_{i,j}, b_{i,j}, c_{i,j}, \beta \in [0 \dots p-1]$, on a*

$$|z| \leq \left(\frac{1+3^l}{2}\right)^2 \left\lfloor \frac{k}{2^l} \right\rfloor (p-1)^2.$$

Et dans le cas d'une représentation centrée, la borne est améliorée :

Corollaire 2.3 (Représentation modulaire centrée)**.** *Avec $a_{i,j}, b_{i,j}, c_{i,j}, \beta \in [-\frac{p-1}{2} \dots \frac{p-1}{2}]$, on a*

$$|z| \leq \left(\frac{3^l}{2}\right)^2 \left\lfloor \frac{k}{2^l} \right\rfloor (p-1)^2.$$

Corollaire 2.4. *On peut calculer l niveaux récursifs de l'algorithme de Winograd sans réduction modulaire sur des entiers représentés sur γ bits dès lors que $k < k_{max}$ où*

$$k_{max} = \left(\frac{2^{\gamma+2}}{\left((1+3^l)(p-1)\right)^2} + 1\right) 2^l,$$

pour une représentation modulaire positive et

$$k_{max} = \left(\frac{2^{\gamma+2}}{\left(3^l(p-1)\right)^2} + 1\right) 2^l,$$

pour une représentation modulaire centrée.

Démonstration. Les bornes des corollaires 2.2 et 2.3 doivent être inférieures à 2^γ. On pose $d = \left\lfloor \frac{k}{2^l} \right\rfloor$, et on résout l'inéquation pour d. Enfin l'encadrement $2^l d \leq k < 2^l(d+1)$ donne le résultat. □

Corollaire 2.5 (Optimalité de la représentation modulaire centrée)**.** *Parmi toutes les représentations modulaires représentant $\mathbb{Z}_p$ par un intervalle $[m \dots M]$, avec $M - m = p - 1$, c'est la représentation centrée ($m = -M = (p-1)/2$) qui minimise la borne du théorème 2.1.*

Démonstration. On supposera sans perte de généralité que $0 \leq -m \leq M$. Pour deux matrices A, B représentées par des coefficients dans l'intervalle $[m \dots M]$, la borne devient

$$\left(\frac{1+3^l}{2}M + \frac{1-3^l}{2}m\right)^2 \left\lceil \frac{k}{2^l} \right\rceil = \left(\frac{1+3^l}{2}(p-1) + m\right)^2 \left\lceil \frac{k}{2^l} \right\rceil$$

qui est minimal quand $m = -M$. On en déduit $M = -m = (p-1)/2$. □

2.3 Mise en œuvre

Dans la nomenclature des BLAS, l'opération de multiplication matricielle s'appelle `gemm`, pour *GEneral Matrix Multiplication*. Le nom de la routine BLAS s'obtient en y apposant le préfixe correspondant au type utilisé pour la représentation flottante : `dgemm` pour des flottants double précision, sgemm pour des flottants simple précision, ... Par analogie, nous nommons la routine de multiplication matricielle dans un corps fini `fgemm`, pour Finite field GEneral Matrix Multiplication.

2.3.1 Une structure en cascade

Pour résumer, nous avons proposé trois idées permettant d'accélérer le produit matriciel sur un corps fini :

1. différer la réduction modulaire en effectuant le plus longtemps possible le calcul sur $\mathbb{Z}$,
2. découper les matrices en petits blocs pour tirer parti des niveaux de mémoire cache,
3. utiliser l'algorithme de multiplication rapide de Winograd.

Chacun de ces points définit un découpage en blocs des matrices et donc un seuil de granularité, séparant deux traitements différents :

1. les bornes sur le débordement ((2.1), (2.2) ou théorème 2.1) définissent une taille maximale k_{max}, de bloc pouvant être traité sur $\mathbb{Z}$. Au delà de cette taille, les matrices sont découpées en blocs de dimensions $m \times k_{\text{max}}$ et $k_{\text{max}} \times n$, chaque bloc est traité sur $\mathbb{Z}$ et les réductions modulaires sont effectuées entre chaque produit de blocs.
2. Pour profiter des niveaux de mémoire cache, les matrices doivent être découpées en blocs contigus (une copie est donc nécessaire). La taille de ces blocs dépend de l'architecture utilisée. Cette idée est développée dans [61].
3. Comme nous l'avons vu dans la partie 2.2.1, l'algorithme de Winograd doit être combiné en pratique avec l'algorithme classique qui effectue le produit matriciel pour des tailles de blocs inférieures à un seuil. Ce dernier doit être déterminé expérimentalement.

Pour le deuxième point (optimiser l'utilisation du cache), cette tâche peut être effectuée *à la main* ou bien confiée aux BLAS qui l'incluent parmi d'autres optimisations. Le graphique de la figure 2.2 compare les vitesses de calcul (en Million d'opérations arithmétiques par seconde) pour l'algorithme classique d'une implémentation standard, de l'implémentation *à la main* du découpage en blocs de taille 33 (expérimentalement le meilleur pour la machine utilisée) et de notre implémentation utilisant les BLAS (`fgemm`). A titre de comparaison, la vitesse de la même opération par les BLAS en calcul flottant est aussi donnée (`dgemm`). On constate premièrement l'intérêt du découpage par blocs (gain de facteur 10). Par ailleurs, l'utilisation des BLAS améliore encore cette vitesse de près du double, grâce à d'autres optimisations de l'arithmétique en virgule flottante (fused-mac, SSE, pipeline...). Ainsi il est préférable d'utiliser directement les BLAS, plutôt que s'efforcer à les imiter. Leur fréquente mise à jour et la diversité des architectures qu'ils recouvrent sont aussi un gage de pérennité.

Il reste donc deux critères de basculement : celui entre l'algorithme de Winograd et l'algorithme classique et celui entre les opérations sur $\mathbb{Z}$ et les réductions modulaires. Nous avons représenté leur agencement dans notre implémentation sur la figure 2.3.

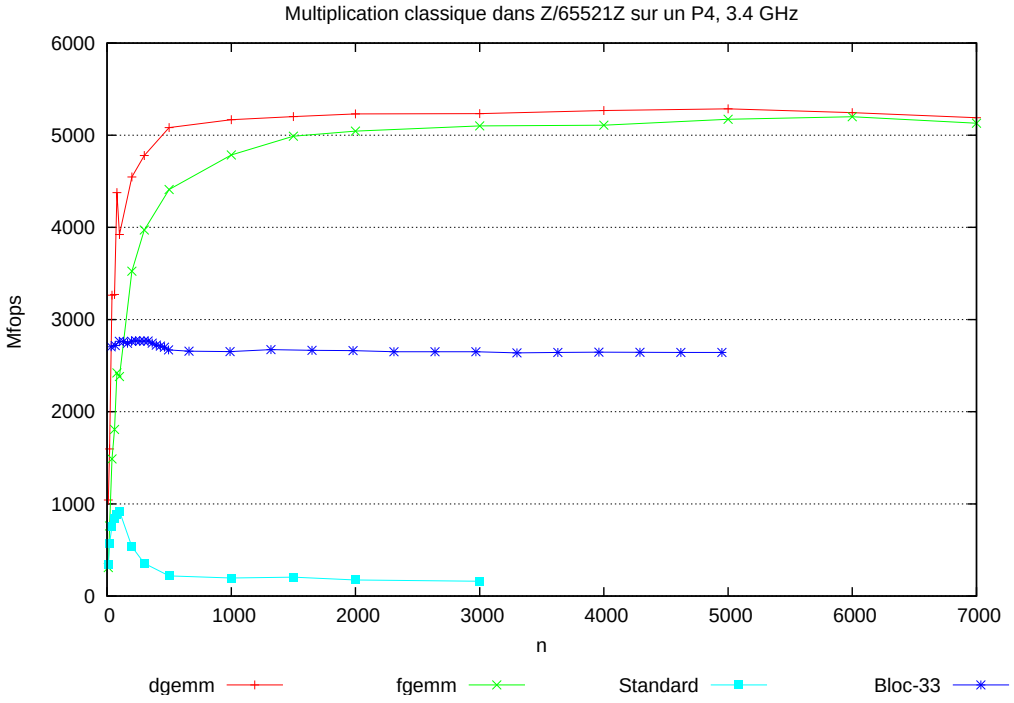


FIG. 2.2 – *Optimisation de cache : multiplication classique par blocs.*

La routine visible, `fgemm`, consiste à calculer les deux seuils : $k_{\max}$ pour le calcul différé et l, le nombre de niveaux récursifs de l'algorithme de Winograd à effectuer. La procédure `WinoMain` effectue les tests relatifs aux seuils. Si $l > 0$, le calcul est délégué à `WinoCalc` sur $\mathbb{Z}_p$ ou à `WinoMain` sur $\mathbb{Z}$, selon le seuil $k_{\max}$. `WinoCalc` effectue les 22 opérations par blocs de l'algorithme de Winograd ainsi que les mises à jour de rang 1 pour les dimensions impaires. Si $l = 0$, le calcul est effectué par `ClassicMatmul` qui découpe les matrices en blocs de taille $k_{\max}$ (éventuellement un seul bloc) et fait effectuer les produits de blocs sur $\mathbb{Z}$ par `ClassicMatmul` qui appelle directement la routine `dgemm` des BLAS, en effectuant les éventuelles conversions et réductions modulaires.

Le calcul des seuils est un point délicat de cette routine. D'une part le nombre de niveaux récursifs l est calculé à partir de k_{Winograd}, la dimension à partir de laquelle un niveau récursif devient plus intéressant que l'algorithme classique. Or ce k_{Winograd}, déterminé expérimentalement, varie, selon que le premier niveau récursif est exécuté dans $\mathbb{Z}$ ($k_{\max} > k$) ou dans $\mathbb{Z}_p$ ($k_{\max} \leq k$). Ainsi l dépend de $k_{\max}$. Or d'autre part, $k_{\max}$ est calculé d'après les formules du corollaire 2.4 et dépend donc de l et de p.

Afin de résoudre cette dépendance réciproque nous ignorons simplement le fait que l dépende de $k_{\max}$ en utilisant la relation (2.4) pour le même seuil k_{Winograd} quelque soit la valeur de $k_{\max}$.

Une perspective d'amélioration serait de d'automatiser le calcul de l en fonction de k et $k_{\max}$ (en utilisant quelques données expérimentales), afin de calculer la valeur optimale l par le schéma itératif suivant :

– $l = 0$

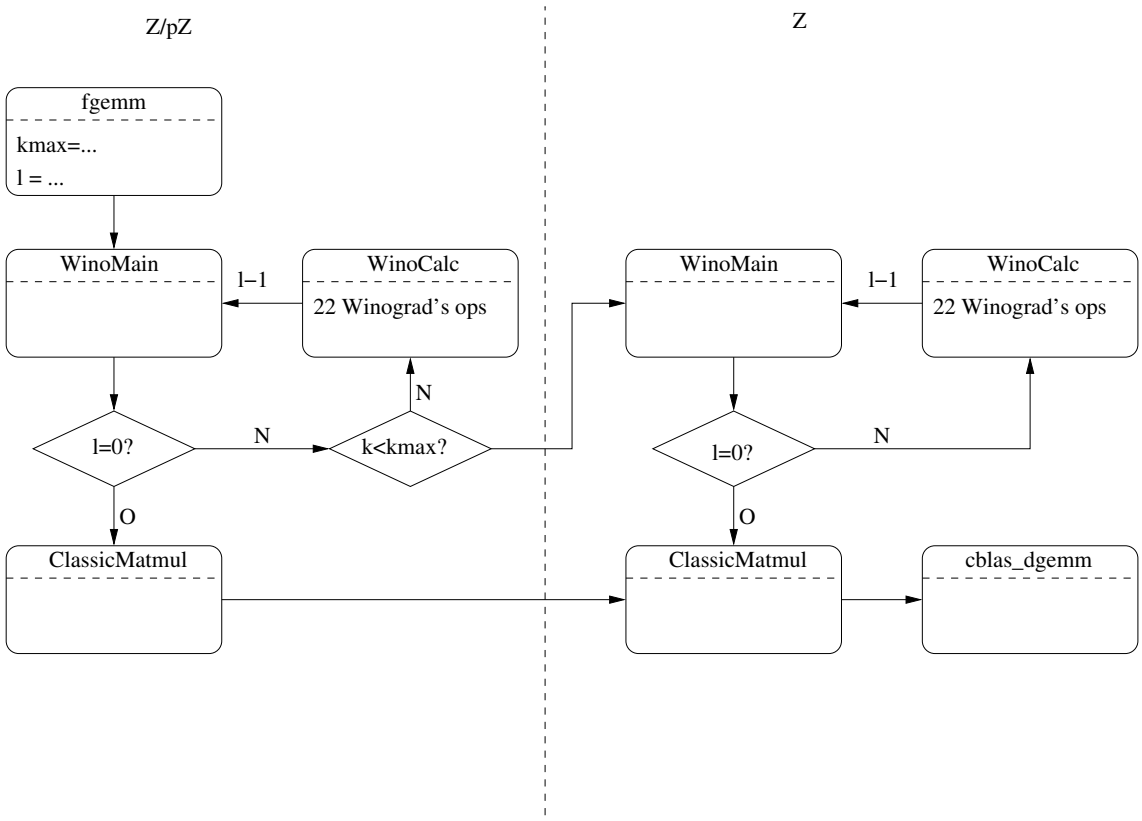


FIG. 2.3 – *Structure en cascade de l'implémentation de fgemm*

Architecture		PIII - 2Mo de Cache		P4- 512Ko de Cache		Itanium2	
k_{Winograd}		400		550		2200	
m		1000	5000	1000	5000	1000	5000
$p =$	l	2	4	1	4	0	2
5	$k_{\max}$	$1,1E14$	$5,49E12$	$5E14$	$5,49E12$	$2,25E15$	$1,1E15$
$p =$	l	2	4	1	4	0	2
1048583	$k_{\max}$	1622	95	7283	95	32768	1622
$p =$	l	2	4	1	4	0	2
8388617	$k_{\max}$	29	17	115	17	512	29

TAB. 2.5 – *Exemple de valeurs des seuils l et k_{max} pour le produit matriciel avec une représentation modulaire centrée*

– Calculer $k_{\max}(l, p, k)$
– répéter
– Calculer $l(k, k_{\max})$
– Calculer $k_{\max}(l, p, k)$
– tant que l augmente

Pour illustrer cette discussion, nous donnons dans le tableau 2.5 différentes configurations des seuils $k_{\max}$ et l selon les architectures, les nombres premiers p et la dimension k considérés.

2.3.2 Expériences et comparaison avec les routines numériques

Nous montrons dans cette partie, que les temps de calcul de cette implémentation sont comparables à ceux du calcul numérique en virgule flottante, pour le produit classique. De plus l'utilisation de l'algorithme de Winograd permet même d'être plus rapide que ces derniers pour des matrices suffisamment grandes.

Pour le produit classique, nous avons vu sur la figure 2.2 que la vitesse sur un corps fini premier reste très proche de la vitesse d'un calcul similaire en virgule flottante. Le gain par rapport à une implémentation naïve du produit est considérable (un facteur 20). Concernant les corps de Galois de type GF(p^k), cette technique peut aussi être appliquée, en utilisant une représentation q-adique des éléments sur $\mathbb{Z}$, où q est un nombre premier différent de p : chaque élément étant un polynôme $k(X)$ à coefficient dans $\mathbb{Z}_p$, on le représente sur $\mathbb{Z}$ par sa valeur en q, $k(q)$. On se reportera à [34, §2.5, Annexe B] pour plus de détails à ce sujet. La figure

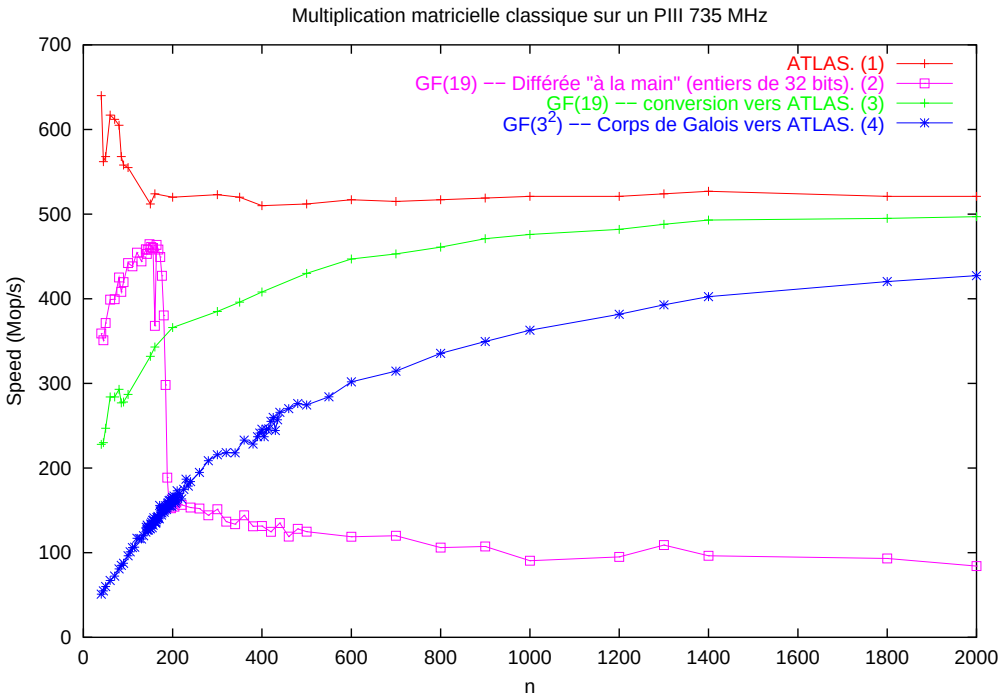


FIG. 2.4 – *Comparaison de vitesses pour le produit matriciel classique*

2.4 donne les temps de calcul pour ce type de corps (courbe (4)). Les opérations de conversion

étant plus coûteuses, la vitesse est inférieure à celle des corps premiers, mais reste toujours quatre fois supérieure à celle d'une implémentation naïve. On remarque aussi sur cette figure les bonnes performances de l'approche naïve sur des petites tailles de matrices. Il y a plusieurs raisons à cela : l'architecture considérée est un Pentium III qui ne dispose pas des opérations de type fused-mac, donc l'avantage des BLAS est essentiellement celui de la gestion des blocs pour la mémoire cache ; or pour de telles tailles, les défauts de ligne de cache n'ont pas lieu (toutes les données tiennent dans les caches), et l'utilisation des BLAS pour ce critère est donc inutile ; enfin, la représentation de corps finis utilisée n'est pas `Modular<double>`, et les conversions d'un type entier vers un type flottant pénalisent donc l'utilisation des BLAS. Ces résultats auraient pu plaider en défaveur de l'utilisation des BLAS, mais leurs progrès avec les architectures plus récentes (Pentium 4, Itanium...) et le recours à la représentation `Modular<double>` rendent cette discussion obsolète, comme le montre la figure 2.2.

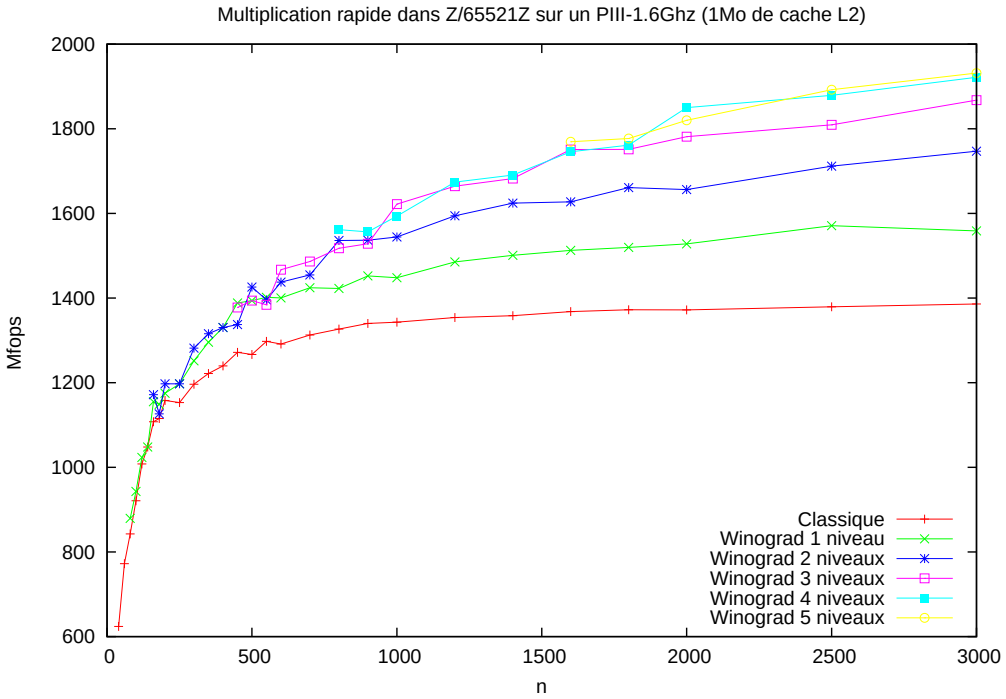


FIG. 2.5 – *Gain dû à l'utilisation de l'algorithme de Winograd*

Nous étudions maintenant l'intérêt de l'algorithme de multiplication rapide de Winograd. La figure 2.5 montre la vitesse du produit matriciel pour différents niveaux récursifs de l'algorithme de Winograd. On constate que sur cette architecture, un seul niveau récursif accélère l'opération pour des tailles supérieures à 80, 2 niveaux pour 140, 3 pour 400, etc. On obtient un gain de 38% pour $n = 3000$. Ces paramètres sont très variables selon les architectures, comme nous allons le voir.

Nous comparons maintenant les temps de calculs entre l'algorithme classique et notre implémentation, avec un choix automatique du nombre de niveaux récursifs de l'algorithme de Winograd, dans les tableaux 2.6 pour un Pentium 4 et dans le tableau 2.7 pour un Itanium2 64 bits.

Ici, le paramètre k_{Winograd} défini pour l'équation (2.4) vaut environ 1000 pour le tableau 2.6 et 2500 pour le tableau 2.7. Les gains atteignent jusqu'à 29% pour des matrices d'ordre 8000. En revanche l'utilisation de mémoire temporaire dans l'algorithme de Winograd peut devenir pénalisante lorsque les ressources en mémoire sont critiques, comme c'est le cas pour les matrices d'ordre 10000.

On remarque aussi l'intérêt de faire reposer notre implémentation sur l'interface générique des BLAS : elle bénéficie du coup directement de leur amélioration, comme c'est ici le cas avec les BLAS GOTO, optimisés pour ce type d'architectures.

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	fgemm	$0,41$s	$2,86$s	$8,81$s	$36,82$s	$101,27$s	$144,66$s	$213,69$s	DM
	dgemm	$0,39$s	$3,06$s	$10,28$s	$47,19$s	$129,20$s	$192,61$s	$276,43$s	$379,05$s
	$\frac{fgemm}{dgemm}$	**1,05**	**0,93**	**0,85**	**0,78**	**0,78**	**0,75**	**0,77**	**-**
GOTO	fgemm	$0,37$s	$2,60$s	$8,32$s	$34,80$s	$90,54$s	$128,18$s	$181,98$s	DM
	dgemm	$0,36$s	$2,79$s	$9,35$s	$43,36$s	$118,07$s	$178,23$s	$251,11$s	$344,73$s
	$\frac{fgemm}{dgemm}$	**1,02**	**0,93**	**0,88**	**0,80**	**0,76**	**0,71**	**0,72**	**-**

DM : *Débordement Mémoire*

TAB. 2.6 – *Comparaison* `fgemm`*/* `dgemm` *sur un P4-3,4GHz-1Go*

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	fgemm	$0,50$s	$3,66$s	$11,87$s	$51,21$s	$131,78$s	$188,14$s	$273,39$s	$363,41$s
	dgemm	$0,46$s	$3,47$s	$11,74$s	$53,90$s	$147,06$s	$217,20$s	$311,57$s	$422,85$s
	$\frac{fgemm}{dgemm}$	**1,08**	**1,05**	**1,01**	**0,95**	**0,89**	**0,86**	**0,87**	**0,85**
GOTO	fgemm	$0,44$s	$3,35$s	$10,68$s	$46,08$s	$119,33$s	$173,24$s	$245,50$s	$328,22$s
	dgemm	$0,40$s	$3,17$s	$10,61$s	$48,90$s	$133,90$s	$200,26$s	$284,47$s	$391,01$s
	$\frac{fgemm}{dgemm}$	**1,1**	**1,05**	**1,00**	**0,94**	**0,89**	**0,86**	**0,86**	**0,83**

TAB. 2.7 – *Comparaison* `fgemm`*/* `dgemm` *sur Itanium2-1,3GHz-16Go*

3

RÉSOLUTION DE SYSTÈMES

La résolution de systèmes pour des matrices denses est essentiellement basée sur la méthode du pivot de Gauss. Celle-ci peut être décomposée en deux phases distinctes : la mise sous forme triangulaire de la matrice, puis la résolution des systèmes triangulaires obtenus. Depuis [10], la description en opérations par blocs de ces algorithmes fait reposer la mise sous forme triangulaire sur deux opérations principales : la résolutions de systèmes triangulaires multiples et le produit matriciel. Dans un ordre constructif, nous commençons donc par l'étude de la résolution de systèmes triangulaires multiples avant d'étudier les triangularisations. Enfin nous appliquerons ces méthodes à l'inversion matricielle.

Ces travaux ont été réalisés en étroite collaboration avec Pascal Giorgi. Son mémoire de thèse [56] développe également le sujet en mettant davantage l'accent sur leur intégration au sein de la bibliothèque LINBOX. La plupart de ces résultats ont aussi été présentés dans l'article [35].

Les complexités algébriques des algorithmes considérés sont toutes en $\mathcal{O}(n^\omega)$. En revanche, nous nous concentrerons sur la constante du terme dominant dans cette complexité, que nous exprimerons en fonction de la constante du produit matriciel. Nous utiliserons ainsi les notations suivantes : pour un algorithme X, T_{X} désignera le terme dominant de la complexité algébrique de cet algorithme. Par exemple, pour la multiplication de deux matrices de dimensions $m \times k$ et $k \times n$, on définit $T_{\text{MM}}(m,k,n)$. En particulier $T_{\text{MM}}(n,n,n) = C_\omega n^\omega$, où C_ω est la constante du produit matriciel ($C_3 = 2$ pour l'algorithme classique, $C_{\log_2 7} = 6$ pour l'algorithme de Winograd).

Pour ces calculs, nous aurons souvent recours au produit matriciel de dimensions quelconques $T_{\text{MM}}(m,n,k)$. Les meilleurs résultats de complexité selon m, n et k sont donnés dans [68]. En particulier, une complexité proche de quadratique, $T(n^\kappa, n, n) = \mathcal{O}\left(n^{2+o(1)}\right)$, est atteinte pour des matrices fortement rectangulaires ($0 \leq \kappa \leq 0,2946$). La complexité finale des algorithmes que nous étudions étant $\mathcal{O}(n^\omega)$, nous n'utiliserons pas ces résultats. Il nous suffira de considérer le découpage des opérandes A et B en blocs carrés d'ordre $z = \min(m,k,n)$ pour obtenir l'expression suivante :

$$T_{\text{MM}}(m,k,n) = C_\omega \left\lceil \frac{m}{z} \right\rceil \left\lceil \frac{n}{z} \right\rceil \left\lceil \frac{k}{z} \right\rceil z^\omega.$$

3.1 Résolution multiple de systèmes triangulaires

La résolution de systèmes triangulaires se décline sous plusieurs variantes : résolution à gauche, à droite d'une matrice triangulaire supérieure ou inférieure. Nous détaillerons ici, sans perte de généralité, le cas de la résolution à gauche avec une matrice triangulaire supérieure. De plus nous considérons la résolution avec un membre de droite matriciel, ce qui peut être vu comme une résolution simultanée de plusieurs systèmes utilisant la même matrice triangulaire. Cette opération est en effet très utilisée dans les algorithmes d'algèbre linéaire par blocs, à commencer par l'élimination de Gauss, comme nous le verrons dans la partie 3.2. Nous étudions donc l'opération

$$X = U^{-1}B,$$

où X et B sont de dimensions $m \times n$ et U est triangulaire supérieure inversible et d'ordre m. Dans la nomenclature BLAS, cette opération s'appelle `trsm` pour *TRiangular System solving with Matrix right/left handside*. De la même façon que pour le produit matriciel, le préfixe `d` indique que la routine `dtrsm` correspond à l'imlpémentation spécialisée pour les flottants double précision.

Cette opération peut être effectuée par un algorithme récursif par blocs, reposant essentiellement sur le produit matriciel. Nous rappellerons cet algorithme dans la partie 3.1.1 en détaillant le calcul de sa complexité algébrique. Nous montrons ensuite comment améliorer les performances en pratique de cet algorithme dans deux implémentations en cascade : l'une basée sur la routine correspondante des BLAS `dtrsm` (partie 3.1.2) et l'autre utilisant le produit scalaire avec réduction modulaire différée, de la partie 1.2 (partie 3.1.3).

3.1.1 L'algorithme récursif par blocs

Une approche *diviser pour régner* permet de construire un algorithme par bloc pour ce problème, et de le réduire ainsi à des produits matriciels, en utilisant le découpage

$$\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A}\overbrace{\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}}^{B}.$$

pour $m \leq n$. Nous en rappelons le principe dans l'algorithme 3.1.

Cet algorithme maintient au cours de ses appels récursifs l'invariant $m \leq n$, ce qui permet de calculer la complexité en déroulant la relation de récurrence. En revanche on ne peut procéder de même pour le cas $m > n$. Nous proposons dans ce cas, d'ajouter au découpage précédent, le découpage des matrices X et B suivant sa dimension n :

$$\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A}\overbrace{\begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}^{B}$$

Ainsi, les blocs X_{11}, X_{12}, X_{21} et X_{22} sont calculés par quatre appels récursifs pour lesquels l'invariant $m > n$ est respecté.

Algorithme 3.1 : `ftrsm` : algorithme récursif

Données : $A \in \mathbb{Z}_p^{m\times m}$, $B \in \mathbb{Z}_p^{m\times n}$, tel que $m \leq n$
Résultat : $X \in \mathbb{Z}_p^{m\times n}$ tel que $AX = B$
début
  si *m=1* **alors**
    $X := A_{1,1}^{-1} \times B$
  sinon
    $X_2 :=$`trsm`(A_3, B_2)
    $B_1 := B_1 - A_2X_2$
    $X_1 :=$`trsm`(A_1, B_1)
  retourner X
fin

Lemme 3.1. *L'algorithme 3.1 est correct. Le terme dominant de sa complexité arithmétique dans $\mathbb{Z}_p$ est*

$$T_{trsm}(m,n) \leq \begin{cases} \frac{C_\omega}{2(2^{\omega-2}-1)} \left\lceil \frac{n}{m} \right\rceil m^\omega & \textit{si } m \leq n, \\ \frac{C_\omega}{2(2^{\omega-2}-1)} \left\lceil \frac{m^2}{n^2} \right\rceil n^\omega & \textit{si } m \geq n, \end{cases}$$

où $C\omega n^\omega$ est le terme dominant de le complexité du produit de deux matrices carrées d'ordre n. En utilisant le produit matriciel classique, $T_{trsm}(m,n) = \min\{mn^2, nm^2\}$

Démonstration. On montre que l'algorithme est correct en remarquant que

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \text{ est solution} \Leftrightarrow \begin{cases} A_3X_2 = B_2 \\ A_1X_1 + A_2X_2 = B_1 \end{cases}.$$

On tire de l'algorithme récursif la relation $T_{\texttt{trsm}}(m,n) = 2T_{\texttt{trsm}}(\frac{m}{2},n) + T_{\texttt{MM}}(\frac{m}{2},\frac{m}{2},n)$. Notons $t = \log_2(m)$. Si $m \leq n$, on a

$$\begin{aligned} T_{\texttt{trsm}}(m,n) &= 2T_{\texttt{trsm}}(\frac{m}{2},n) + \left\lceil \frac{n}{m} \right\rceil \frac{C_\omega}{2^{\omega-1}} m^\omega \\ &= 2^t T_{\texttt{trsm}}(1,n) + \frac{C_\omega}{2} m^\omega \sum_{i=1}^{t} \left\lceil \frac{2^i n}{m} \right\rceil \left(\frac{1}{2^{\omega-1}}\right)^i \\ &\leq 2^t T_{\texttt{trsm}}(1,n) + \left\lceil \frac{n}{m} \right\rceil \frac{C_\omega}{2} m^\omega \sum_{i=1}^{t} \left(\frac{2}{2^{\omega-1}}\right)^i \\ &\leq 2^t T_{\texttt{trsm}}(1,n) + \left\lceil \frac{n}{m} \right\rceil \frac{C_\omega}{2} m^\omega \frac{2}{2^{\omega-1}} \frac{1-\left(\frac{2}{2^{\omega-1}}\right)^t}{1-\frac{2}{2^{\omega-1}}}. \end{aligned}$$

Comme $T_{\texttt{trsm}}(1,n) = 2n$ et $(2^{\omega-1})^t = m^{\omega-1}$, on obtient

$$T_{\texttt{trsm}}(m,n) \leq \frac{C_\omega}{2(2^{\omega-2}-1)} \left\lceil \frac{n}{m} \right\rceil m^\omega + \mathcal{O}(m+mn).$$

Si $m \geq n$, la relation de récurrence devient

$$\begin{aligned} T_{\texttt{trsm}}(m,n) &= 4T_{\texttt{trsm}}(\frac{m}{2},\frac{n}{2}) + 2T_{\texttt{MM}}(\frac{m}{2},\frac{m}{2},\frac{n}{2}) \\ &\leq 4T_{\texttt{trsm}}(\frac{m}{2},\frac{n}{2}) + \left\lceil \frac{m^2}{n^2} \right\rceil \frac{2C_\omega}{2^\omega} n^\omega \\ &\leq 4^t T_{\texttt{trsm}}(m/2^t,1) + \left\lceil \frac{m^2}{n^2} \right\rceil \frac{C_\omega}{2} n^\omega \sum_{i=1}^{t} \left(\frac{4}{2^\omega}\right)^i \\ &\leq 4^t T_{\texttt{trsm}}(m/2^t,1) + \left\lceil \frac{m^2}{n^2} \right\rceil \frac{C_\omega}{2} n^\omega \frac{4}{2^\omega} \frac{1-\left(\frac{4}{2^\omega}\right)^t}{1-\frac{4}{2^\omega}} \end{aligned}$$

On trouve donc

$$T_{\texttt{trsm}}(m,n) \leq \frac{C_\omega}{2(2^{\omega-2}-1)} \left\lceil \frac{m^2}{n^2} \right\rceil n^\omega + \mathcal{O}\left(\left(\frac{m}{n}\right)^2\right).$$

□

Remarque 3.1. Dans le cas où m et n sont des puissances de 2, l'expression de ces complexités se simplifie en

$$T_{\texttt{trsm}}(m,n) = \begin{cases} \frac{C_\omega}{2(2^{\omega-2}-1)} m^{\omega-1} n & \text{si } m \leq n \\ \frac{C_\omega}{2(2^{\omega-2}-1)} n^{\omega-2} m & \text{si } m \geq n \end{cases}.$$

Par la suite, nous nous placerons systématiquement dans cette situation pour étudier les constantes, afin de simplifier les notations. Le résultat asymptotique reste inchangé.

Remarque 3.2. Ces complexités pourraient être améliorées dans certaines distribution de m et n en utilisant le produit de matrice rectangulaire de Huang et Pan [67]. Par la suite nous utiliserons le plus souvent cet algorithme pour dans le cas où $m = n$, pour lequel cet algorithme reste avantageux.

Au niveau de la mise en pratique, la réduction au produit matriciel, permet aussi de tirer parti de l'efficacité de la routine de produit matriciel `fgemm` étudiée dans le chapitre 2. Néanmoins, une démarche alternative consiste à appliquer l'idée de la partie 2.1.2, en utilisant la routine équivalente des BLAS : `dtrsm`.

3.1.2 Utilisation de la routine `dtrsm` des BLAS

Nous montrons dans cette partie comment utiliser la routine `dtrsm` des BLAS pour le calcul sur un corps fini. Pour ce faire nous montrerons comment contourner le problème des divisions, puis nous étudierons les limitations dues au grossissement des coefficients.

Traitement des divisions

Comme nous l'avons vu dans la partie 2.1.2, l'utilisation d'une routine des BLAS pour un corps fini, revient à plonger le calcul dans $\mathbb{Z}$, pour ensuite convertir le résultat dans le corps

fini. Pour résoudre un système triangulaire de dimension n, il faut effectuer n divisions dont le dividende est chacun des éléments diagonaux de la matrice triangulaire. Ces divisions ne sont pas a priori exactes, et un système à coefficients dans $\mathbb{Z}$ peut ainsi faire apparaître une solution rationnelle. Ceci rend impossible la conversion finale dans le corps fini. En revanche, la diagonale de la matrice triangulaire est formée de 1, les divisions seront nécessairement exactes, et le résultat appartiendra à $\mathbb{Z}$.

Nous proposons donc un pré-traitement simple permettant de se ramener au cas de la diagonale unité. Pour le système triangulaire $AX = B$, il suffit de factoriser A sous la forme $A = UD$, où D est la matrice diagonale formée par les coefficients diagonaux de A, et U, une matrice triangulaire de diagonale unité. Le système $UY = B$ peut ensuite être résolu sans division. Enfin, on trouve la solution du système initial en calculant $X = D^{-1}Y$ dans le corps fini. Ce préconditionnement demande un surcoût de :

- m inversions dans $\mathbb{Z}_p$ pour le calcul de D^{-1}.
- $(m-1)\frac{m}{2} + mn$ multiplications dans $\mathbb{Z}_p$ pour la normalisation de U et de X.

Contrôle du grossissement des coefficients

Le calcul sur $\mathbb{Z}$ est rendu possible. Il reste maintenant à s'assurer que les valeurs calculées sont toujours représentables en machine. De la même façon que nous avons procédé dans les parties 2.1.1 et 2.2.4 pour le produit matriciel, nous allons borner les valeurs possibles des calculs intervenant dans la résolution sur $\mathbb{Z}$.

On peut avoir une première idée de la croissance des coefficients en remarquant que le $k$ième coefficient x_k du vecteur solution du système $Ax = b$ est une combinaison linéaire des $n - k$ coefficients suivants : $x_i, i \in [k+1 \dots n]$. Par conséquent, la taille du plus grand des coefficients croît linéairement en fonction de la dimension du système. Nous donnons dans le théorème 3.1 une borne plus précise de la valeur des coefficients calculés. Nous donnons aussi une classe de systèmes pour lesquels la borne est atteinte, ce qui prouve son optimalité.

Theorème 3.1. *Soit $T \in \mathbb{Z}^{n\times n}$ une matrice triangulaire supérieure avec une diagonale unité et $b \in \mathbb{Z}^n$, avec $m \leq T_{i,j} \leq M$ et $m \leq b_i \leq M$. On suppose que $m \leq 0 \leq M$. Soit $x = (x_i)_{i\in[1\dots n]} \in \mathbb{Z}^n$ la solution du système $Tx = b$ dans $\mathbb{Z}$. Alors, $\forall\, k \in [0 \dots n-1]$:*

$$\begin{cases} -u_k \leq x_{n-k} \leq v_k & \textit{si } k \textit{ est pair,} \\ -v_k \leq x_{n-k} \leq u_k & \textit{si } k \textit{ est impair,} \end{cases}$$

avec

$$\begin{cases} u_k = \frac{M-m}{2}(M+1)^k - \frac{M+m}{2}(M-1)^k, \\ v_k = \frac{M-m}{2}(M+1)^k + \frac{M+m}{2}(M-1)^k. \end{cases}$$

Démonstration. On remarque tout d'abord les propriétés suivantes :

$$\forall k \begin{cases} u_k & \leq & v_k \\ -mu_k & \leq & Mv_k \\ -mv_k & \leq & Mu_k \end{cases}$$

Les deux premières sont évidentes et la troisième s'obtient en vérifiant que

$$Mu_k + mv_k = \frac{M^2 - m^2}{2}((M+1)^k - (M-1)^k) \geq 0.$$

La preuve se fait maintenant par récurrence sur k, en suivant l'odre de la résolution par remontée du système triangulaire.

Le cas initial $k = 0$ correspond à la première étape de la remontée : $x_n = b_n$, d'où l'on tire

$$-u_0 = m \leq x_n \leq M = v_0.$$

Supposons maintenant que les inégalités soient vérifiées pour $k \in [0 \dots l]$ et montrons les pour $k = l + 1$. Si l est impair, $l + 1$ est pair.

$$\begin{aligned}
x_{n-l-1} &= b_{n-l-1} - \sum_{j=n-l}^{n} T_{n-l-1,j} x_j \\
&\leq M + \sum_{i=0}^{\frac{l-1}{2}} \max(Mu_{2i}, -mv_{2i}) + \max(Mv_{2i+1}, -mu_{2i+1}) \\
&\leq M \left(1 + \sum_{i=0}^{\frac{l-1}{2}} u_{2i} + v_{2i+1}\right) \\
&\leq M \left(1 + \sum_{i=0}^{\frac{l-1}{2}} \frac{M-m}{2}(M+2)(M+1)^{2i} + \frac{M+m}{2}(M-2)(M-1)^{2i}\right) \\
&\leq M \left(1 + \frac{M-m}{2}(M+2)\frac{(M+1)^{l+1}-1}{(M+1)^2-1} + \frac{M+m}{2}(M-2)\frac{(M-1)^{l+1}-1}{(M-1)^2-1}\right) \\
&\leq \frac{M-m}{2}(M+1)^{l+1} + \frac{M+m}{2}(M-1)^{l+1} = v_{l+1}.
\end{aligned}$$

De même,

$$\begin{aligned}
x_{n-l-1} &\geq m - \sum_{i=0}^{\frac{l-1}{2}} \max(Mv_{2i}, -mu_{2i}) + \max(Mu_{2i+1}, -mv_{2i+1}) \\
&\geq m - M \sum_{i=0}^{\frac{l-1}{2}} v_{2i} + u_{2i+1} \\
&\geq m - M \sum_{i=0}^{\frac{l-1}{2}} \frac{M-m}{2}(M+2)(M+1)^{2i} - \frac{M+m}{2}(M-2)(M-1)^{2i} \\
&\geq m - M \left(\frac{M-m}{2}(M+2)\frac{(M+1)^{l+1}-1}{(M+1)^2-1} - \frac{M+m}{2}(M-2)\frac{(M-1)^{l+1}-1}{(M-1)^2-1}\right) \\
&\geq \frac{M-m}{2}(M+1)^{l+1} - \frac{M+m}{2}(M-1)^{l+1} = u_{l+1}.
\end{aligned}$$

Et on montre de la même façon que pour l pair, on a

$$-v_{l+1} \leq x_{n-l-1} \leq u_{l+1}.$$

□

Corollaire 3.2. *En reprenant les notations du théorème 3.1,*

$$|x| \leq \frac{M-m}{2}(M+1)^{n-1} + \frac{M+m}{2}(M-1)^{n-1}.$$

De plus, cette borne est optimale.

Démonstration. On remarque simplement que la suite (v_k) est croissante et toujours supérieure à la suite (u_k). Ainsi $\forall\ k \in [0 \dots n-1] |x_{n-k}| \leq\ u_k \leq v_k \leq v_{n-1}$.

Or le système suivant

$$T = \begin{bmatrix} \ddots & \ddots & \ddots & \ddots & \ddots \\ & 1 & M & m & M \\ & & 1 & M & m \\ & & & 1 & M \\ & & & & 1 \end{bmatrix}, b = \begin{bmatrix} \vdots \\ m \\ M \\ m \\ M \end{bmatrix}$$

admet pour solution le vecteur $x = (x_i)_{i\in[1\dots n]} \in \mathbb{Z}^n$ tel que $\forall\ k \in [0 \dots n-1]\ |x_{n-k}| = v_k$. La borne étant atteinte, elle est donc optimale. □

Nous appliquons ce résultat pour les représentations positives dans le corollaire 3.3 et les réprésentations centrées dans le corollaire 3.4.

Corollaire 3.3 (Représentation modulaire positive)**.** *Si $T_{i,j}, b_i \in [0 \dots p-1]$, on a*

$$|x| \leq \frac{p-1}{2}(p^{n-1} + (p-1)^{n-1}).$$

Corollaire 3.4 (Représentation modulaire centrée)**.** *Si $T_{i,j}, b_i \in [-\frac{p-1}{2} \dots \frac{p-1}{2}]$, on a*

$$|x| \leq \frac{p-1}{2}\left(\frac{p+1}{2}\right)^{n-1}.$$

Démonstration. En prenant $m = 0, M = p-1$ pour les représentations modulaires positives et $-m = M = \frac{p-1}{2}$ pour les réprésentations modulaires centrées. □

Remarque 3.3. La représentation modulaire centrée permet donc de réduire cette borne d'un facteur 2^{n-1}.

Ainsi, pour un corps fini donné, on peut calculer la solution d'un système triangulaire de diagonale unité de dimension n par des opérations arithmétiques sur des nombres entiers représentés sur γ bits si

$$\frac{p-1}{2}(p^{n-1} + (p-1)^{n-1}) < 2^{\gamma} \tag{3.1}$$

pour une représentation modulaire positive et

$$\frac{p-1}{2}\left(\frac{p+1}{2}\right)^{n} < 2^{\gamma} \tag{3.2}$$

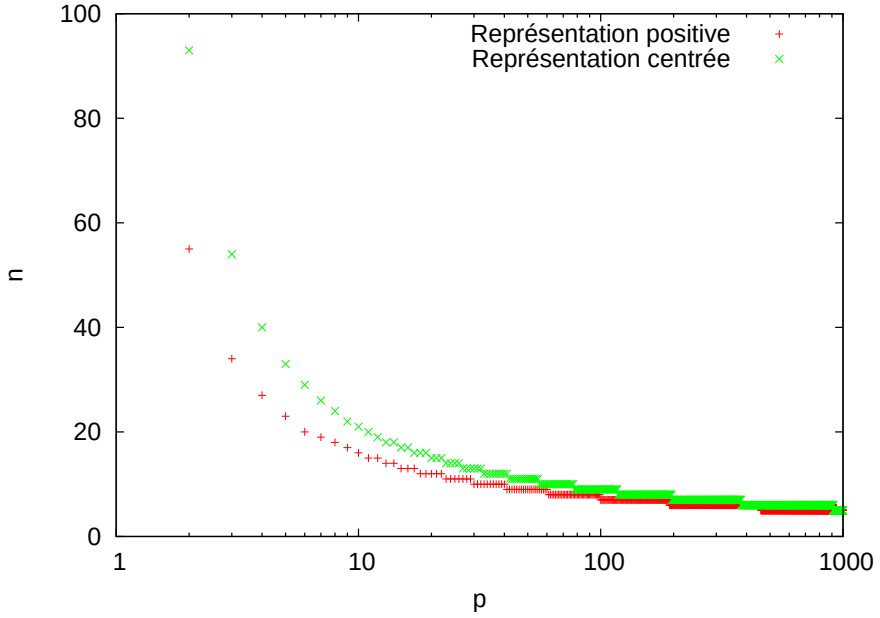


FIG. 3.1 – *Dimension maximale du système triangulaire pour une résolution sur des entiers de 53 bits*

pour une représentation modulaire centrée.

Par exemple, pour les entiers représentés sur les `double` machine (53 bits de mantisse), on obtient des dimensions maximales respectivement de 55 et 93 pour des représentations positives et centrées de $\mathbb{Z}_2$. Nous montrons l'évolution de cette dimension maximale selon p dans la figure 3.1. Pour des corps plus grands, cette dimension maximale devient très rapidement petite : pour $p = 1001$, $n \leq 5$ pour une représentation positive et $n \leq 6$ pour une représentation centrée. Réciproquement, pour un système de dimension 3, le plus grand corps fini que l'on peut utiliser est $\mathbb{Z}_{208057}$ pour une représentation positive et $\mathbb{Z}_{416107}$ pour une représentation centrée. Ces dimensions peuvent paraître trop petites, mais nous montrerons dans la partie 3.1.4 que cette technique permet pourtant d'améliorer la rapidité de la résolution de système triangulaire dans certains cas.

Intégration dans l'algorithme récursif

Cette limitation dans la dimension des systèmes pouvant être résolus sur $\mathbb{Z}$ par les BLAS nous pousse à considérer à nouveau une approche en cascade : l'algorithme récursif est utilisé en premier lieu ; la dimension décroît au fil des appels récursifs jusqu'à ce qu'elle vérifie l'une des bornes (3.1) ou (3.2), et alors, la résolution sur $\mathbb{Z}$ par la routine `dtrsm` peut être utilisée.

Dans cette configuration, on remarque que la normalisation du système n'est nécessaire qu'au niveau terminal de la partie récursive, lors de l'appel à `dtrsm`. Il est donc préférable de l'effectuer à ce niveau, plutôt que sur le système entier. Plus précisément, soit β la taille des systèmes lors de l'appel à la résolution par `dtrsm`. On supposera sans perte de généralité que la taille du système initial s'écrit $m = 2^i\beta$, où i est le nombre de niveaux récursifs effectués.

Nous proposons donc d'effectuer 2^i normalisations de systèmes de dimension β, soit un coût de :

- m inversions dans $\mathbb{Z}_p$,
- $(\beta - 1)\frac{m}{2} + mn$ multiplications dans $\mathbb{Z}_p$.

Ceci permet de gagner $\left(\frac{1}{2} - \frac{1}{2^{i+1}}\right) m^2$ multiplications dans $\mathbb{Z}_p$ par rapport à la normalisation initiale du système. Un seul niveau récursif permet déjà d'économiser $\frac{1}{4}m^2$ multiplications et ce gain peut atteindre $\frac{1}{2}(m^2 - m)$ multiplications avec $\log_2 m$ niveaux.

3.1.3 Résolution avec réduction modulaire différée

L'algorithme récursif de la partie 3.1.1 réduit la résolution de systèmes triangulaires à des produits matriciels de différentes tailles. Sur un corps fini, cette démarche engendre des réductions modulaires inutiles et on peut essayer de différer ces réductions comme nous l'avons fait pour le produit scalaire ou le produit matriciel. Il faut donc effectuer les produits matriciels des appels récursifs sur $\mathbb{Z}$ et contrôler la croissance des résultats pour appliquer la réduction au moment opportun.

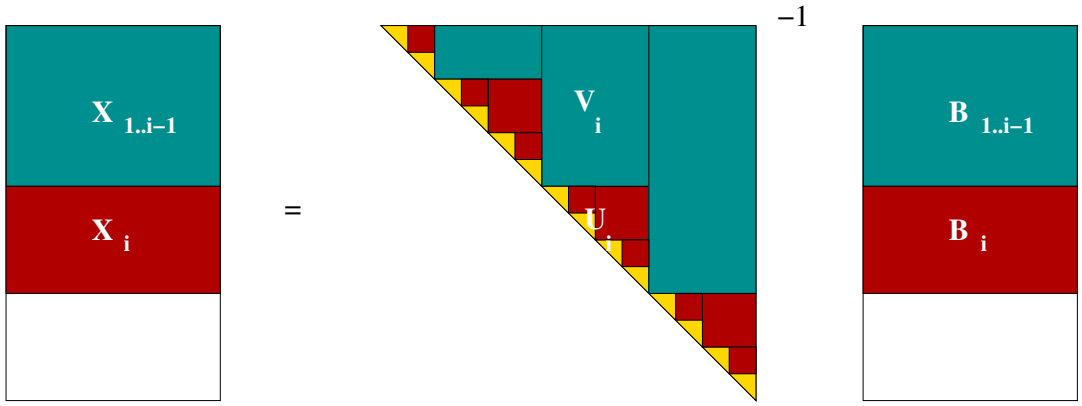


FIG. 3.2 – *Découpage pour l'algorithme* `trsm` *en double cascade*

Cette technique peut être intégrée au sein de l'algorithme en cascade précédent pour donner l'algorithme 3.2 en double cascade illustré sur la figure 3.2. Le principe est d'effectuer un découpage fin selon le grain n_{BLAS}, déterminé avec (3.2) ou (3.1), permettant l'utilisation de la routine BLAS `dtrsm` pour la résolution triangulaire. Puis un découpage au niveau supérieur selon $n_{\text{différé}}$, déterminé avec (2.1) ou (2.2), permettant l'utilisation de la routine BLAS `dgemm` pour le produit matriciel différé. La taille de ce dernier découpage doit être un multiple de la taille du découpage précédent. L'algorithme procède alors de la façon suivante : une boucle lance les résolutions des blocs triangulaires de taille $n_{\text{différé}}$ puis la mise à jour par un produit matriciel sur $\mathbb{Z}$ suivi d'une réduction modulaire. Les blocs triangulaires sont traités par l'algorithme récursif `trsm_différé` dont le niveau terminal est la résolution par la routine BLAS `dtrsm` et dont les mises à jours se font sans réduction modulaire.

Algorithme 3.2 : `trsm` : Recursif Blas Différé

Données : $A \in \mathbb{Z}_p^{m\times m}$, $B \in \mathbb{Z}_p^{m\times n}$
Résultat : $X \in \mathbb{Z}_p^{m\times n}$ tel que $AX = B$
début
  Calculer le seuil n_{BLAS} d'après (3.2) ou (3.1)
  Calculer le seuil $n_{\text{différé}}$ d'après (2.1) ou (2.2)
  $n_{\text{découpage}} \leftarrow \text{Min}(m/2, n_{\text{différé}})$
  $n_{\text{découpage}} \leftarrow (n_{\text{découpage}}/n_{\text{BLAS}})n_{\text{BLAS}}$; `/* Ajustement de` $n_{\texttt{découpage}}$ `à un multiple du grain` $n_{\texttt{BLAS}}$ `*/`
  pour tout *bloc-colonne de* A *de largeur* $n_{découpage}$ *de la forme* $\begin{bmatrix} V_i \\ U_i \\ 0 \end{bmatrix}$ **faire**
    $X_i = \texttt{trsm\_différé}(U_i, B_i)$
    $X_i = X_i \mod p$
    $B_{1\ldots i-1} = B_{1\ldots i-1} - V_i X_i$; `/* avec réduction modulaire */`
  retourner X
fin

Algorithme 3.3 : `trsm_différé`

Données : $A \in \mathbb{Z}_p^{m\times m}$, $B \in \mathbb{Z}_p^{m\times n}$, m doit être inférieur à $n_{\texttt{différé}}$
Résultat : $X \in \mathbb{Z}_p^{m\times n}$ tel que $AX = B$
début
  si $m \leq n_{BLAS}$ **alors**
    $X = \texttt{trsm\_blas}(A, B)$; `/* décrit dans la partie 3.1.2 */`
  sinon
    `/* (découpage des matrices en deux blocs de tailles` $\lfloor \frac{m}{2} \rfloor$ `et` $\lceil \frac{m}{2} \rceil$`) */`

$$\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}}^{B}$$

    $X_2 := \texttt{trsm\_différé}(A_3, B_2)$
    $B_1 := B_1 - A_2 X_2$; `/* sans réduction modulaire */`
    $X_1 := \texttt{trsm\_différé}(A_1, B_1)$
  retourner X
fin

3.1.4 Résultats expérimentaux

Comparaison des variantes

De l'étude précédente, nous retenons quatre variantes pour l'implémentation de la résolution multiple de systèmes triangulaires :

Récursive pure (`rec-pure`) : simplement l'algorithme récursif,

Récursive-BLAS (`rec-BLAS`) : la méthode en cascade de la partie 3.1.2, utilisant la routine `dtrsm`,

Récursive-BLAS-Différée (`rec-BLAS-différé`) : la méthode en double cascade de la partie 3.1.3

Récursive-Différée (`rec-différé`) : idem, sans le recours à `dtrsm`. Correspond à `rec--pure` dont les mises ont des réductions modulaires différées.

Nous comparons ces quatre variantes pour des corps finis de tailles différentes permettant de faire varier les paramètres n_{BLAS} et $n_{\text{différé}}$, comme indiqué dans le tableau 3.1.

p	$\lceil \log_2 p \rceil$	n_{BLAS}	$n_{\text{différé}}$
5	3	23	2 147 483 642
1 048 583	20	2	8190
8 388 617	23	2	126

TAB. 3.1 – *Valeurs des seuils n_{BLAS} et $n_{différé}$ pour différents corps finis*

Dans la figure 3.3 la matrice B est carrée ($m = n$) et nous utilisons l'implémentation de corps fini `Modular<double>`.

On constate tout d'abord l'avantage donné par l'utilisation de `dtrsm` en comparant les courbes `rec-blas` et `rec-pur` pour $p = 5$. Cette différence s'estompe pour les autres corps finis, puisque la taille du seuil n_{BLAS} devient trop petite : $n_{\text{BLAS}} = 2$.

Le fait de différer les réductions modulaires donne un gain d'environ 500 Mfops. Les courbes pour $p = 5$ et $p = 1\,048\,583$ sont similaires, puisque dans les deux cas $n < n_{\text{différé}}$ et il n'y a donc aucune réduction modulaire entre les produits matriciels. Enfin, pour $p = 8\,388\,617$, le découpage en blocs de taille $n_{\text{différé}}$ donne un net avantage aux variantes `rec-différé` et `rec-blas-différé`. Les variantes `rec-pur` et `rec-blas` sont pénalisées par leur découpage dichotomique des dimensions, engendrant au final un nombre excessif de réductions modulaires. En revanche nous ne pouvons pas expliquer la raison pour laquelle la variante `rec-blas-différé` est nettement supérieure à `rec-différé` pour $p = 8\,388\,617$, alors qu'elle est sensiblement inférieure pour $p = 5$ ou $p = 1\,048\,583$.

Performance de la routine `ftrsm`

Suite à la comparaison précédente, nous utiliserons pour `ftrsm` la variante `rec-BLAS--Différée`. Afin d'illustrer son efficacité, nous la comparons à la routine numérique `dtrsm` ainsi qu'au produit matriciel.

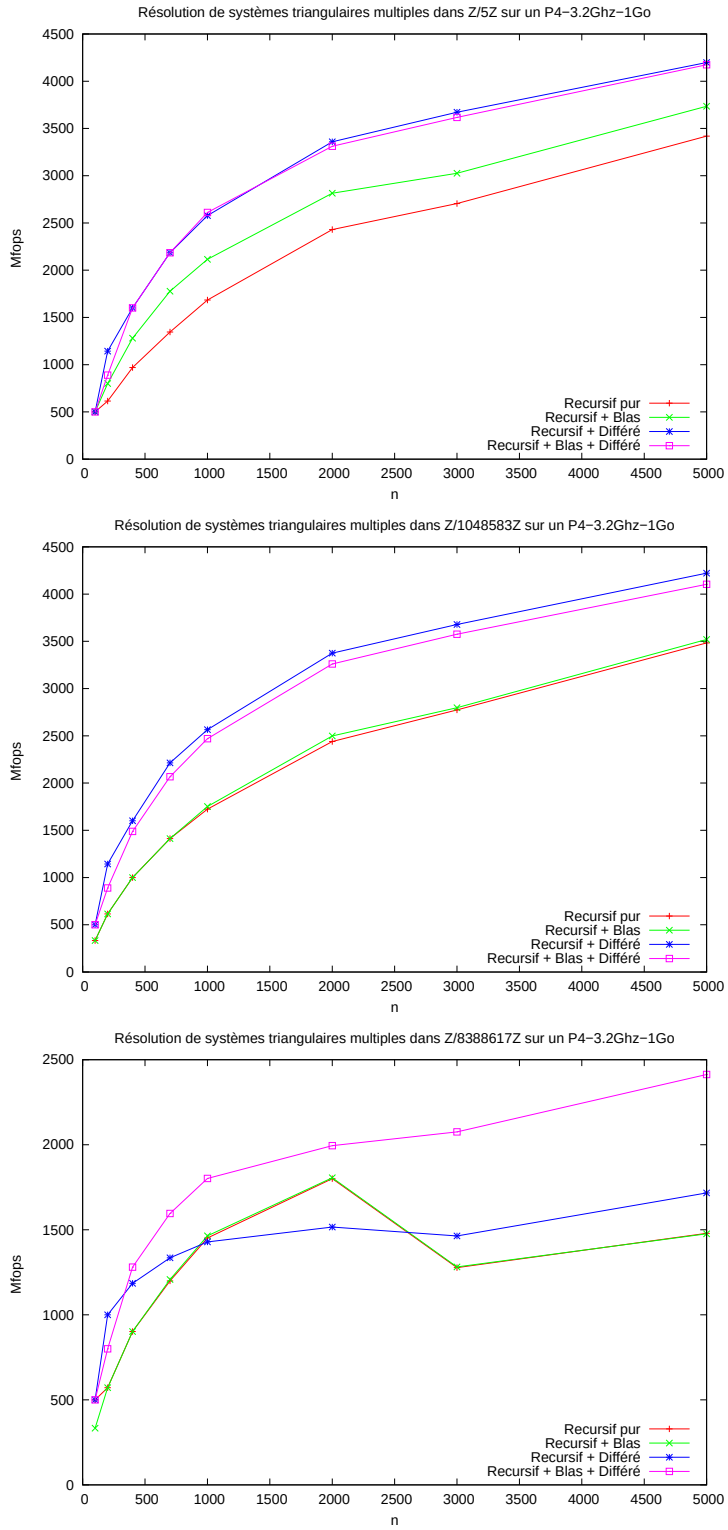


FIG. 3.3 – *Comparaison des variantes de* `trsm` *pour* $p = 5, 1\,048\,583, 8\,388\,617$ *avec le BLAS GOTO sur un Pentium4-3,2Ghz-1Go*

De la même façon que dans la partie 2.3.2, nous utilisons deux BLAS différents (ATLAS et GOTO) et deux architectures. Néanmoins nous ne présentons pas les résultats de la routine `dtrsm` d'ATLAS sur le pentium 4 tellement ses performances sont faibles. Nous utilisons le corps finis $\mathbb{Z}_{65521}$ avec l'implémentation modulaire positive `Modular<double>`.

	n	1000	2000	3000	5000	7500	10 000
GOTO	ftrsm	$0,40$s	$2,49$s	$7,46$s	$30,40$s	$96,89$s	$228,03$s
	dtrsm	$0,24$s	$1,88$s	$6,00$s	$27,21$s	$90,71$s	$226,80$s
	$\frac{ftrsm}{dtrsm}$	**1,66**	**1,33**	**1,24**	**1,12**	**1,07**	**1,01**

TAB. 3.2 – *Temps de calcul de la résolution triangulaire multiple sur un P4-3,2GHz-1Go*

	n	1000	2000	3000	5000	7500	10 000
ATLAS	ftrsm	$0,37$s	$2,30$s	$6,59$s	$31,75$s	$101,7$s	$209,39$s
	dtrsm	$0,22$s	$1,71$s	$5,50$s	$29,87$s	$99,4$s	$210,45$s
	$\frac{ftrsm}{dtrsm}$	**1,67**	**1,35**	**1,20**	**1,06**	**1,02**	**0,99**
GOTO	ftrsm	$0,33$s	$2,23$s	$5,95$s	$25,62$s	$81,18$s	$209,25$s
	dtrsm	$0,19$s	$1,59$s	$4,85$s	$22,68$s	$76,97$s	$190,87$s
	$\frac{ftrsm}{dtrsm}$	**1,72**	**1,41**	**1,22**	**1,13**	**1,05**	**1,09**

TAB. 3.3 – *Temps de calcul de la résolution triangulaire multiple sur un Itanium2-1,5GHz--24Go*

Les tableaux 3.2 et 3.3 montrent que notre routine `ftrsm` atteint des performances proches de celles de la routine numérique `dtrsm`, en particulier pour les grandes dimensions. Par ailleurs, sur un Pentium 4 avec les BLAS GOTO, elle atteint des performances meilleures qu'en numérique pour $n = 10\,000$. Cette amélioration provient de l'utilisation d'un produit matriciel rapide pour `ftrsm`, alors qu'il s'agit d'un produit classique pour `dtrsm`.

Pour illustrer l'importance du produit matriciel dans cette opération, nous montrons dans la figure 3.4 les ratios des temps de `ftrsm` et `fgemm` selon les architecture et les BLAS utilisées. Avec un produit matriciel classique, ce ratio est théoriquement de $1/2$. En utilisant le résultat du lemme 3.1, il est de $\frac{1}{2(2^{\log_2 7-2}-1)} = 2/3$ avec l'exposant $\omega = \log_2 7$ de l'algorithme de Winograd. Expérimentalement, les courbes de ratios semblent converger vers cette valeur. Par exemple le ratio est de $0,7$ pour $n = 10\,000$ sur un Pentium 4 avec ATLAS.

3.2 Triangularisation

Par triangularisation, nous désignons la mise sous forme triangulaire d'une matrice, à partir de la méthode du pivot de Gauss. Cette élimination peut être vue comme une factorisation (ou décomposition) matricielle sous la forme $A = LU$ où L et U sont respectivement triangulaires inférieures et supérieures. On peut obtenir cette décomposition par des opérations élémentaires

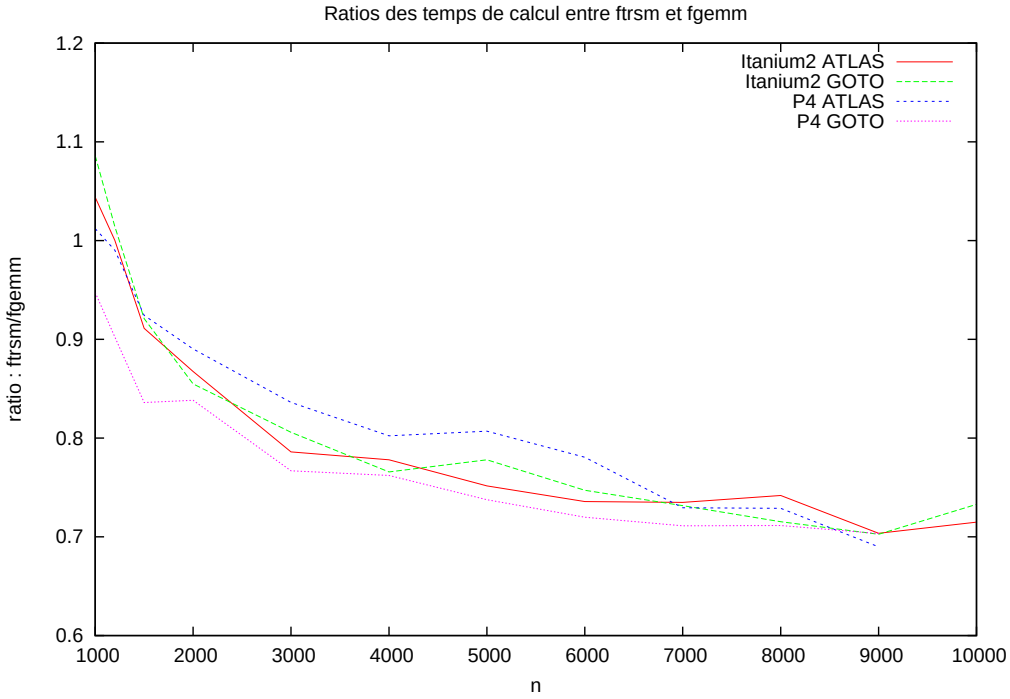


FIG. 3.4 – *Ratios des temps de calcul entre la résolution triangulaire et le produit matriciel*

sur les lignes de A. La décomposition $A = LU$ est définie pour une matrice A générique, de sorte qu'à chaque itération, le pivot soit inversible. On généralise ensuite cette décomposition aux matrices inversibles quelconques, en autorisant une permutation sur les lignes, ce qui permet de toujours trouver un pivot inversible. La décomposition devient alors $A = PLU$. Si, on choisit plutôt une permutation sur les colonnes pour trouver le pivot, on obtient une décomposition $A = LUP$. Dans les deux cas, le fait que la matrice soit inversible certifie que l'on trouvera un élément non nul sur la ligne ou la colonne où on le recherche. Enfin, les matrices non inversibles, peuvent être triangularisées sous une forme dite échelonnée par ligne ou par colonne. Nous détaillerons ce dernier point dans la partie 4.1. On associe alors à ces matrices une décomposition $A = LQUP$ ou $A = PLUQ$ en introduisant une seconde permutation, ou encore une décomposition $A = LSP$ que nous étudierons plus en détails.

Afin de réduire ces décompositions au produit matriciel, des algorithmes par blocs ont été proposés en 1974, dans [2, 10] pour les matrices inversibles. Ils sont depuis intensivement exploités dans la bibliothèque `LAPACK`. Nous rappelons avec l'algorithme 3.4 le principe de la décomposition `LU` par blocs de matrices génériques.

Pour le calcul de la décomposition `LUP` de matrices inversibles, il suffit de modifier le cas terminal $m = 1$ en ajoutant la recherche dans A d'un élément non nul et sa permutation, ainsi que la mise à jour des permutations au cours des appels récursifs.

Lemme 3.2. *Le terme dominant de la complexité algébrique de l'algorithme* `LUP` *pour* $m \leq n$ *est*

$$T_{\texttt{LUdec}}(m,n) = \frac{C_\omega}{2(2^{\omega-2}-1)} m^{\omega-1} n - \frac{C_\omega}{2(2^{\omega-1}-1)} m^\omega.$$

Algorithme 3.4 : `LUdec` : décomposition `LU`

Données : A, une matrice $m \times n$ générique
Résultat : L triangulaire inférieure unitaire de dimension $m \times m$ et U triangulaire supérieure de dimensions $m \times n$ telles que $A = LU$
début
  si *m=1* **alors**
    retourner $(L = [1], U = A)$;
  sinon
    Découper $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ où A_{11} est $m/2 \times m/2$;
    $(L_1, [U_{11}, U_{12}]) = \texttt{LU}([A_{11}\ A_{12}])$ où U_{11} est $m/2 \times m/2$;
    $G = A_{21}U_{11}^{-1}$;
    $H = A_{22} - GU_{12}$;
    $(L_2, U_2) = \texttt{LU}(H)$;
    retourner $\left(L = \begin{bmatrix} L_1 & \\ G & L_2 \end{bmatrix}, U = \begin{bmatrix} U_{11} & U_{12} \\ & U_2 \end{bmatrix} \right)$
fin

Ce dernier vaut $nm^2 - \frac{1}{3}m^3$ avec la multiplication matricielle classique.

Remarque 3.4. Avec la multiplication matricielle classique et en prenant $m = n$, on retrouve la complexité de l'algorithme de Gauss : $\frac{2}{3}n^3$.

Démonstration. D'après le lemme 0.2, ce terme dominant est de la forme $\mathcal{O}\left(m^\omega + nm^{\omega-1}\right)$. Il reste à établir les constantes α et β telles que $T_{\texttt{LUdec}}(m,n) = \alpha m^\omega + \beta nm^{\omega-1}$

La structure récursive de l'algorithme donne la relation

$$\begin{aligned} T_{\texttt{LUdec}}(m,n) = {} & T_{\texttt{LUdec}}(m/2,n) + T_{\texttt{trsm}}(m/2,m/2) + T_{\texttt{MM}}(m/2,m/2,n-m/2) \\ & + T_{\texttt{LUdec}}(m/2,n-m/2). \end{aligned} \tag{3.3}$$

D'où l'on tire

$$\begin{aligned} \alpha m^\omega + \beta nm^{\omega-1} &= 2\alpha\left(\frac{m}{2}\right)^\omega + \beta\left(2n - \frac{m}{2}\right)\left(\frac{m}{2}\right)^{\omega-1} + \frac{C_\omega}{2(2^{\omega-2}-1)}\left(\frac{m}{2}\right)^\omega \\ &\quad + C_\omega\left(n - \frac{m}{2}\right)\left(\frac{m}{2}\right)^{\omega-1} \\ &= \frac{1}{2^\omega}\left(2\alpha - \beta + \frac{C_\omega}{2(2^{\omega-2}-1)} - C_\omega\right)m^\omega + \frac{1}{2^\omega}(2\beta + C_\omega)nm^{\omega-1}. \end{aligned}$$

Ce qui donne le système

$$\begin{cases} \alpha &= \frac{1}{2^\omega}\left(2\alpha - \beta + \frac{C_\omega}{2(2^{\omega-2}-1)} - C_\omega\right) \\ \beta &= \frac{1}{2^\omega}(2\beta + C_\omega) \end{cases}$$

qui se résout en $\alpha = -\frac{C_\omega}{2(2^{\omega-1}-1)}$ et $\beta = \frac{C_\omega}{2(2^{\omega-2}-1)}$. □

En 1982, les décompositions `LSP` et `LQUP` ont été introduites dans [71] pour les matrices quelconques. Parallèlement, Keller-Gehrig a présenté en 1985 dans [81] un autre algorithme par blocs pour la mise sous forme échelonnée en ligne.

Pour l'élaboration d'une routine de triangularisation, nous avons donc retenu les décompositions `LSP` et `LQUP`. Nous montrons que la décomposition `LQUP`, qui n'est qu'une faible variation de la décomposition `LSP`, permet une meilleure gestion de la mémoire puisqu'elle peut être effectuée complètement en place. Par ailleurs ces décompositions remplacent avantageusement la mise sous forme échelonnée de Keller-Gehrig car elles produisent une véritable factorisation matricielle, dont les facteurs peuvent ensuite être utilisés dans les calculs. Par exemple, nous verrons dans le chapitre 4 comment la décomposition `LQUP` permet non seulemement de déterminer le profil de rang d'une matrice, mais fournit aussi les matrices nécessaires à sa mise sous profil de rang générique et à sa complétion en une base.

3.2.1 La décomposition `LSP`

La décomposition `LSP` est une généralisation de la décomposition `LUP` pour des matrices non nécessairement inversibles. Pour ces matrices, la recherche d'un pivot non nul sur une ligne peut échouer. Dans ce cas, la ligne est simplement laissée nulle. Par conséquent, la matrice triangulaire supérieure U est donc remplacée par une matrice S dite semi-triangulaire supérieure : la matrice extraite en prenant ses lignes non nulles est triangulaire supérieure. Nous renvoyons à [71] ou [9, §2.7c] pour plus de détails sur l'algorithme. Nous représentons sur la figure 3.5 son déroulement en quatre phases. On remarque que la matrice L ne peut être

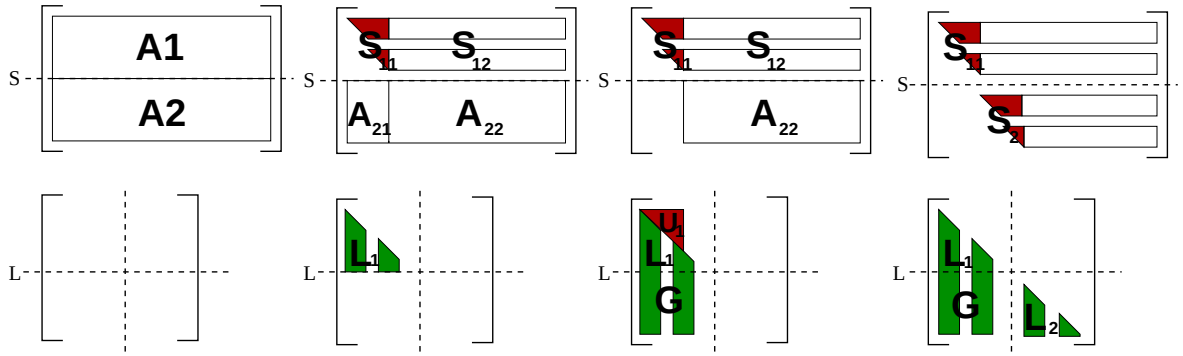


FIG. 3.5 – *Principe de l'algorithme de décomposition* `LSP`

stockée en place "sous la matrice" S, et nécessite donc l'allocation de m^2 coefficients. De plus, pour le calcul de $A_{21}U_1^{-1}$, il faut extraire la sous matrice inversible de S_1 que l'on peut copier dans l'espace libre de L. Nous verrons dans la partie 3.2.2 que la décomposition `LQUP` permet de remédier à cet inconvénient.

Remarque 3.5. Pour des matrices de plein rang, cet algorithme est équivalent à l'algorithme `LUP`, et a donc la complexité annoncée dans le lemme 3.2. Pour le cas des matrices de rang quelconque, nous ne pouvons donner une expression simple de la complexité de l'algorithme en fonction du rang.

3.2.2 La décomposition LQUP

L'inconvénient de la décomposition LSP est que les lignes nulles intercalées dans la matrice S interdisent de stocker la matrice L "en dessous" de celle-ci, c'est à dire, dans la partie triangulaire inférieure laissée nulle. Pour cette raison, il est préférable d'utiliser la décomposition LQUP, elle aussi introduite dans [71]. Il s'agit d'une variante de la décomposition LSP où la matrice S est remplacée par la matrice U triangulaire supérieure, obtenue par des permutations des lignes nulles de S. Ces permutations sont représentées par la transposée de la matrice Q. Ainsi on passe de la décomposition LQUP à la décomposition LSP en posant simplement $S = QU$.

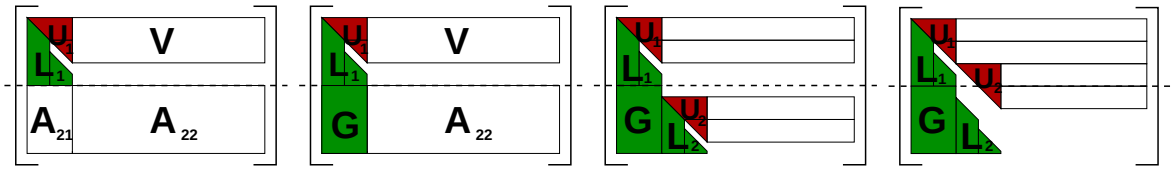


FIG. 3.6 – *Principe de l'algorithme de décomposition* LQUP

Cette décomposition permet ainsi d'effectuer toute la décomposition en place, comme indiqué dans la figure 3.6, et par ailleurs, il n'est plus nécessaire d'extraire la matrice U_1 pour effectuer l'opération $A_{21}U_1^{-1}$. Toujours dans un souci de réduire les copies de données, il est préférable de stocker la matrice L sans ses colonnes nulles, c'est à dire sous la forme LQ, comme représenté dans la figure 3.6.

Nous verrons dans la partie 3.2.3, que ces différentes améliorations permettent à la routine LQUP d'être à la fois plus efficace en temps et en mémoire que la routine LSP.

3.2.3 Comparaisons expérimentales

LSP contre LQUP

Pour ces comparaisons, nous avons à nouveau utilisé l'implémentation Modular<double> positive pour le corps fini, et les BLAS ATLAS. Nous avons utilisé des matrices denses (n^2 éléments alloués), mais avec seulement $3n$ éléments non nuls choisis aléatoirement. Ceci afin d'avoir potentiellement des matrices de rang inférieur à n.

n	*400*	*1000*	*3000*	*5000*	*8000*	*10000*
LSP	0,05s	0,48s	8,01s	32,54s	404,8s	1804s
LQUP	0,05s	0,45s	7,59s	29,90s	201,7s	1090s

TAB. 3.4 – *Comparaison du temps réel d'exécution entre* LSP *et* LQUP *dans* $\mathbb{Z}_{101}$*, sur un P4, 2,4GHz-512Mo*

Les temps de calculs donnés dans le tableau 3.4 sont globalement proches pour des dimensions allant jusqu'à 5000 car les opérations dominantes dans les deux algorithmes sont les

mêmes (produits matriciels et résolutions triangulaires). Néanmoins LSP est plus lente sans doute à cause des copies de données supplémentaires. Ce surcoût reste limité, mais à partir de dimensions supérieures à 8000, l'écart devient considérable. Il s'agit du ralentissement dû à l'utilisation de la mémoire virtuelle sur le disque dur (mémoire swap), qui intervient lorsque la mémoire vive est saturée. Ainsi, l'algorithme LSP, le plus coûteux en mémoire se retrouve aussi pénalisé en temps de calcul pour ce type de dimensions.

n	*400*	*1000*	*3000*	*5000*	*8000*	*10000*
LSP	2,83Mo	17,85Mo	160,4Mo	444,2Mo	1136Mo	1779Mo
LQUP	1,28Mo	8,01Mo	72,02Mo	200,0Mo	512,1Mo	800,0Mo

TAB. 3.5 – *Comparaison de l'allocation mémoire entre LSP et LQUP dans $\mathbb{Z}_{101}$, sur un P4, 2,4GHz-512Mo*

Le tableau 3.5 précise le volume des allocations mémoires de ces deux implémentations. LSP alloue légèrement plus du double que LQUP, pour stocker la matrice L (voir figure 3.5), ainsi que pour former les blocs contigus à partir des blocs semi-triangulaires de la matrice S. On note par ailleurs que pour $n = 8000$, LQUP alloue 512,1 Mo qui ne nécessite qu'une utilisation minime de la mémoire virtuelle (512 Mo de mémoire vive disponible), alors que LSP alloue 1136 Mo, ce qui sollicite beaucoup plus la mémoire virtuelle, provoquant ainsi un fort ralentissement.

Nous reviendrons sur le sujet de la gestion mémoire dans la partie 3.2.4, où nous étudions l'algorithme de triangularisation TURBO qui permet, outre son parallélisme, de mieux gérer les problème de débordement de la mémoire vive.

Pour des comparaisons plus détaillées, on pourra aussi se reporter à l'article [35] sur le sujet.

Comparaison avec la routine numérique `dgetrf`

Nous avons vu dans les parties 2.3.2 et 3.1.4 que l'utilisation du produit matriciel rapide de Winograd permettait aux routines sur les corps finis d'atteindre des performances similaires voire supérieures à celles en calcul numérique. Nous répétons ici cette approche en comparant la routine LQUP avec son équivalent numérique : `dgetrf` de la bibliothèque LAPACK [3]. Plus précisément, nous utilisons son implémentation fournie dans ATLAS, en la liant soit avec les BLAS d'ATLAS soit avec ceux de GOTO.

Les tableaux 3.6 et 3.7 comparent ces temps de calcul dans le corps $\mathbb{Z}_{65521}$. Les matrices tests sont formés de coefficients tirés aléatoirement dans le corps. Elles ont ainsi un profil de rang générique avec une grande probabilité, et les algorithmes n'ont donc aucune permutation de ligne ou de colonne à effectuer dans le recherche des pivots.

Pour $n = 10\,000$, LQUP est moins de 10% plus lente que `dgetrf` sur un Pentium 4. On ne peut espérer tirer autant parti du produit matriciel rapide que dans le tableau 2.6, en effet l'algorithme LQUP ne fait intervenir au plus qu'un produit matriciel de dimension $n/2$. Ainsi, le gain de l'algorithme rapide sera forcément moins important. Dans le cas du calcul sur un Pentium 4, le seuil k_{Winograd} du premier est 1000. Ainsi, le produit matriciel avec $n =$

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	lqup	$0,34$s	$1,98$s	$5,54$s	$21,38$s	$54,67$s	$79,45$s	$110,29$s	$137,56$s
	dgetrf	$0,17$s	$1,22$s	$3,93$s	$17,39$s	$46,65$s	$65,09$s	$97,99$s	$133,01$s
	$\frac{lqup}{dgetrf}$	**2,00**	**1,62**	**1,40**	**1,22**	**1,17**	**1,22**	**1,12**	**1,03**
GOTO	lqup	$0,30$s	$1,78$s	$5,08$s	$19,94$s	$49,67$s	$73,22$s	$97,71$s	$129,76$s
	dgetrf	$0,15$s	$1,14$s	$3,54$s	$16,86$s	$41,77$s	$62,28$s	$88,00$s	$120,71$s
	$\frac{lqup}{dgetrf}$	**2,00**	**1,56**	**1,43**	**1,18**	**1,18**	**1,17**	**1,11**	**1,07**

TAB. 3.6 – *Comparaison en temps de calcul de* `LQUP` *avec* `dgetrf` *sur P4, 3,4GHz-1Go*

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	lqup	$0,45$s	$2,59$s	$7,32$s	$28,94$s	$71,19$s	$101,57$s	$141,79$s	$189,39$s
	dgetrf	$0,22$s	$1,53$s	$4,84$s	$21,14$s	$55,84$s	$81,81$s	$116,02$s	$156,34$s
	$\frac{lqup}{dgetrf}$	**2,04**	**1,69**	**1,51**	**1,36**	**1,21**	**1,24**	**1,22**	**1,21**
GOTO	lqup	$0,57$s	$2,48$s	$6,32$s	$22,77$s	$49,36$s	$68,90$s	$107,46$s	$184,83$s
	dgetrf	$0,20$s	$1,39$s	$4,33$s	$18,69$s	$49,44$s	$73,28$s	$103,07$s	$140,81$s
	$\frac{lqup}{dgetrf}$	**2,85**	**1,78**	**1,45**	**1,21**	**0,99**	**0,94**	**1,04**	**1,31**

TAB. 3.7 – *Comparaison en temps de calcul de* `LQUP` *avec* `dgetrf` *sur Itanium2-1,5GHz--24Go*

10 000 utilise 4 niveaux récursifs, alors que l'algorihtme LQUP n'en utilisera au plus que 3. Ce phénomène est amplifié sur l'architecture Itanium2, pour laquelle le seuil est plus élevé : $k_{\text{Winograd}} = 2500$.

3.2.4 TURBO : un algorithme pour la localité en mémoire

Nous avons vu dans la partie 3.2.3 que la gestion de la mémoire pouvait avoir des répercussions sur le temps de calcul des implémentations, en particulier lorsque la totalité de la mémoire vive disponible est allouée, et que la mémoire virtuelle sur disque (mémoire swap) est utlisée. Dans de telles situations, l'ordonnancement des opérations effectuées séquentiellement importe fortement : un algorithme traitant les données disséminées dans toute la mémoire multipliera les appels mémoire au disque dur, ralentissant ainsi l'exécution. A l'inverse, un algorithme respectant la localité en mémoire des données qu'il manipule, limitera ce ralentissement.

Nous étudions donc maintenant un algorithme produisant la décomposition LQUP, mais dont les opérations sont organisées différemment, afin de mieux respecter la localité des données et de permettre une meilleure granularité. Il s'agit de l'algorithme TURBO, décrit dans [37] et dont la figure 3.7 donne un aperçu du principe. Cet algorithme diffère de celui dé-

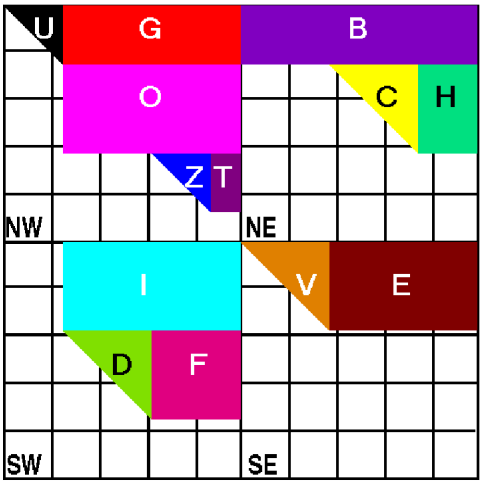


FIG. 3.7 – *Principe de l'algorithme TURBO*

crit dans la partie 3.2.2 en ce qu'il effectue un découpage récursif sur les lignes *et* sur les colonnes. Ceci permet de conserver les proportions dans les dimensions des blocs au fil des appels récursifs. A l'inverse, l'algorithme de la partie 3.2.2 produisait des blocs de plus en plus rectangulaires. De plus TURBO permet d'utiliser une structure récursive par blocs pour le stockage de la matrice. Ces deux points assurent à TURBO un meilleur respect de la localité des données en mémoire. Ceci est essentiel en parallélisme, mais aussi en calcul séquentiel, comme nous allons le voir. On pourra se rapporter à l'article [61] sur le sujet des structures matricielles récursives, et leurs avantages.

Nous rappelons d'abord brièvement le principe de l'algorithme TURBO (voir [37] pour plus de détails). A chaque niveau récursif, la matrice est partagée en quatre blocs. Puis cinq appels

récursifs triangularisent les blocs $(U, V, C, D$ et $Z)$ (dans l'ordre sur la figure 3.7), combinés avec des calculs de compléments de Schur, utilisant six inversions triangulaires multiples et quatre produits matriciels.

Nous avons implémenté une version simplifiée de cet algorithme : l'algorithme n'est appliqué que sur un seul niveau récursif, les niveaux inférieurs étant effectués par `LQUP`. En effet, nous voulons montrer le bénéfice de cet algorithme pour le cas des matrices dépassant de peu en taille la quantité de mémoire pouvant être allouée. Ainsi, le premier niveau récursif devrait permettre de traiter plus efficacement les blocs de taille critique, et les niveaux inférieurs traiteront avec des blocs totalement alloués en mémoire vive, qui conviennent à `LQUP`. De plus, la matrice L n'est pas calculée dans notre implémentation, pour des raisons de simplicité. Cela permet néanmoins d'obtenir un algorithme de calcul du rang ou du déterminant.

Dans la figure 3.8, nous comparons les vitesses de calcul de `TURBO` et de `LQUP`, utili-

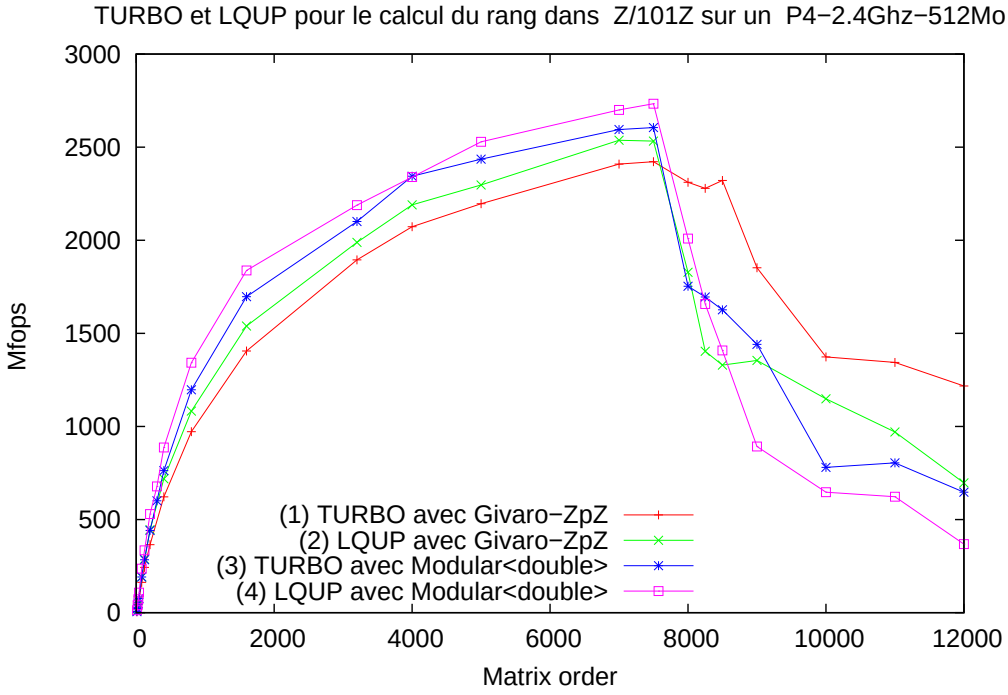


FIG. 3.8 – *Comparaison des vitesses de `TURBO` et `LQUP` pour le calcul du rang*

sant les deux représentations de corps fini modulaires : `Givaro-ZpZ` et `Modular<double>`, basées respectivement sur des entiers machines ou des flottants double précision. Les matrices carrées d'ordre inférieur à 8000 peuvent être stockées intégralement dans les 512 Mo de mémoire vive. Alors `LQUP` est légèrement plus rapide que `TURBO`, sans doute pénalisé par un morcellement plus important des blocs (le produit matriciel, qui est l'opération de base de ces algorithmes, est d'autant plus rapide que les blocs sont grands). La représentation `Givaro-ZpZ` utilise des entiers machine. Elle engendre donc des allocations mémoire supplémentaires dans les produits matriciels pour la conversion vers des flottants afin d'utiliser la routine BLAS `dgemm`. Ce surcoût en mémoire devient critique lorsque n est proche de 8000. A partir de ce seuil, les performances de `LQUP` s'effondrent alors que celles de `TURBO` se maintiennent sensiblement au même niveau jusqu'à $n = 8500$. Ceci s'explique par la moins

bonne localité en mémoire de `LQUP` : de grands blocs rectangulaires doivent être alloués pour les produits matriciels, ce qui implique une forte utilisation de la mémoire virtuelle. A l'opposé, la meilleure localité en mémoire de `TURBO` limite l'utilisation de la mémoire virtuelle et à l'implémentation de rester efficace avec des dimensions supérieures.

Concernant la représentation `Modular<double>`, il n'y a plus d'allocation de mémoire temporaire, les deux algorithmes effectuent leurs calculs "en place". L'absence de conversion permet de gagner en vitesse pour $n \leq 8000$. A partir de 8000, on constate le même effondrement en vitesse pour `LQUP`, mais aussi pour `TURBO` qui partagent la même complexité en mémoire. Cependant, l'effondrement est moindre pour `TURBO`, ce que nous attribuons à sa plus grande localité en mémoire, grâce à la structure de données récursive.

Ces comparaisons, plaident en faveur du développement des structures de données récursives et des algorithmes respectueux de la localité en mémoire. Par ailleurs, lors de calculs impliquant la mémoire virtuelle, ce n'est pas forcément la quantité totale d'allocations en mémoire qu'il faut améliorer, mais plutôt la localité de données manipulées. En effet, c'est l'implémentation de `TURBO` avec `Givaro-ZpZ` qui est la plus efficace pour $n \geq 8000$. Nous renvoyons aussi à [35, §4.5] où cette comparaison est détaillée.

4

ÉTUDE SYSTÉMATIQUE SUR CERTAINS NOYAUX D'ALGÈBRE LINÉAIRE

Les routines que nous avons étudiées aux chapitres 2 et 3 sont des briques de base sur lesquelles reposera l'efficacité de l'ensemble des routines que nous décrivons. Pour la résolution de plusieurs autres problèmes canoniques en algèbre linéaire, nous utilisons des algorithmes récursifs par blocs, permettant de réduire ces problèmes à ces briques de base et de tirer ainsi parti de leur efficacité. Nous décrivons ici en particulier les problèmes suivants :

- calcul du profil de rang,
- le calcul d'une base du noyau d'une matrice rectangulaire ou non inversible,
- le produit de matrices triangulaires,
- l'élévation au carré,
- la triangularisation de matrices symétriques,
- l'inversion de matrices (carrées, triangulaires ou même rectangulaires par l'inverse généralisé).

Tous ces problèmes sont résolus par des algorithmes de complexité $\mathcal{O}(n^\omega)$. En vue de leur mise en pratique, il reste à préciser la constante du terme dominant dans leur complexité. Ceci motive donc une description précise de chacun de ces algorithmes pour lesquels nous donnons l'expression de cette constante (en fonction de ω et C_ω). Afin de réduire ces constantes, nous proposons des algorithmes parfois différents de ceux utilisés pour la réduction au produit matriciel. Certes ils ne sont probablement pas nouveaux dans la plupart des cas, mais l'étude systématique de leur constante, et leur compilation au sein d'un chapitre comme celui-ci faisait défaut dans la littérature.

4.1 Le profil de rang

Le profil de rang d'une matrice décrit la localisation des colonnes linéairement indépendantes de cette matrice. On trouve par exemple dans [104, §2] la définition suivante :

Le profil de rang en colonne d'une matrice de rang r est la sous suite $(j_1, \ldots, j_r)$ de $(1, \ldots, m)$ lexicographiquement minimale telle que les colonnes $j_1, \ldots, j_r$ sont de rang r.

On définit de la même façon le *profil de rang en ligne*.

On dit qu'une matrice a un profil de rang générique, si tous ses mineurs principaux sont non nuls, ou de façon équivalente, si ses profils de rang en ligne et en colonne sont les suites $(1, \ldots, r)$.

Le profil de rang d'une matrice peut s'obtenir à partir de sa forme échelonnée par ligne (aussi appelée forme échelon) qui est largement étudiée dans [104, §2]. Le premier algorithme tirant parti du produit matriciel rapide est dû à Keller-Gehrig [81] en 1985. On trouvera aussi dans [13, §16.5] une étude détaillée sur la complexité de l'algorithme de Keller-Gehrig. Storjohann propose dans [104] une version de l'algorithme de Gauss-Jordan par blocs afin d'obtenir la forme échelon en $\mathcal{O}(nmr^{\omega-2})$. Cette méthode présente l'avantage d'inclure le rang r de la matrice dans l'expression de la complexité.

Nous montrons dans la partie 4.1.1, que la décomposition `LQUP` permet elle aussi d'obtenir le profil de rang d'une matrice. A partir de cette décomposition, nous proposons dans la partie 4.1.2 une transformation basée sur des permutations, pour obtenir une matrice ayant un profil de rang générique.

4.1.1 D'après la décomposition `LQUP`

Lemme 4.1. *Soit A une matrice $m \times n$ de rang r et soit la décomposition `LQUP` de sa transposée : ${}^tA = LQUP$. Alors*

$$\textit{le profil de rang de } A \textit{ est } (j_1, \ldots, j_r) \textit{ si et seulement si } \forall k \in [1 \ldots r]\ Q_{j_k,k} = 1$$

Démonstration. On utilise la décomposition `LSP` (en posant $S = QU$), pour obtenir $L^{-1}\,{}^tA = SP$. Soit $(j_1, \ldots, j_r)$ les indices des lignes non nulles de S. Ces indices vérifient la relation $\forall k \in [1 \ldots r]\ Q_{j_k,k} = 1$.

Par ailleurs, les lignes non nulles de S, donc de SP sont toutes linéairement indépendantes et le déroulement de l'algorithme `LQUP` garantit que cette séquence est lexicographiquement minimale.

Enfin, la multiplication de ${}^tA$ à gauche par une matrice triangulaire ne change pas les propriétés d'indépendance linéaire de la matrice, donc $(j_1, \ldots, j_r)$ est bien le profil de rang de A. □

On en déduit en particulier l'algorithme 4.1 qui calcule la forme réduite en colonnes d'une

Algorithme 4.1 : `FRC` : Forme Réduite en Colonne

Données : A une matrice $m \times n$ de rang r ($m, n \geq r$) sur un corps
Résultat : A' la matrice $m \times r$ formée des r premières colonnes linéairement indépendantes de A.
début
  $(L, Q, U, P, r) =$ LQUP$({}^tA)$; `/* r=rang(A) */`
  retourner ${}^t([I_r 0]({}^tQ\,{}^tA))$
fin

matrice A de rang r : c'est la matrice formée des r premières colonnes linéairement indépendantes de A. Nous l'utiliserons dans la partie 7.2 pour le maniement des matrices de Krylov

dans le calcul du polynôme caractéristique. On peut définir de la même façon l'algorithme `FRL` calculant la forme réduite en ligne, qui sera utilisée dans la partie 11.1.3.

Ainsi, le profil de rang d'une matrice peut être obtenu à partir d'une décomposition `LQUP`. Comme les algorithmes de Keller-Gehrig ou de Storjohann, elle offre une complexité sous-cubique. Cependant, nous ne connaissons pas d'expression de cette complexité en fonction du rang, comme c'est le cas pour l'algorithme de Keller-Gehrig. En revanche, elle a l'avantage d'être une factorisation matricielle, permettant d'effectuer d'autres opérations, comme l'inversion, le calcul d'une base du noyau, etc. En particulier, nous allons montrer comment cette propriété permet de mettre la matrice sous profil de rang générique.

4.1.2 Mise sous profil de rang générique

Lemme 4.2. *Soit A une matrice $m \times n$ de rang r, dont la décomposition `LQUP` s'écrit $A = LQUP$. Alors, la matrice ${}^tQA\,{}^tP$ est de profil de rang générique.*

Démonstration. Soit $\overline{L}$ le bloc diagonal principal $r \times r$ de la matrice ${}^tQLQ$. La multiplication à droite par Q consiste à effectuer sur les r premières colonnes de L l'opération $C_k \leftarrow C_{j_k}$. Ainsi, pour $k \in [1, \ldots, r]$, la $k$ème colonne de LQ correspond à la $j_k$ème de L. Ses $j_k - 1$ premiers coefficients sont donc nuls et le $j_k$ème est 1.

Puis la mutliplication à gauche par ${}^tQ$ correspond à l'opération sur $L_k \leftarrow L_{j_k}$ sur les lignes de LQ. La matrice $\overline{L}$ est donc triangulaire inférieure avec une diagonale unité. Comme la matrice U est triangulaire supérieure, avec ses r premiers éléments diagonaux non nuls, on conclut que ${}^tQA\,{}^tP = \begin{bmatrix} \overline{L} & * \\ * & * \end{bmatrix} U$ est de profil de rang générique. □

4.2 Complétion d'une famille libre en une base

Soit une famille de k vecteurs libres $(v_1, \ldots, v_k)$ dans K^n. On veut calculer $n-k$ vecteurs de K^n, $(w_1, \ldots, w_{n-k})$ tels que la famille $(v_1, \ldots, v_k, w_1, \ldots, w_{n-k})$ soit une base de K^n.

Du point de vue matriciel, on pose $V = \begin{bmatrix} v_1 & \ldots & v_k \end{bmatrix}$ et on cherche une matrice W de dimensions $n \times (n-k)$ telle que $B = \begin{bmatrix} V & W \end{bmatrix}$ soit inversible. Cette opération est élémentaire à partir d'une décomposition triangulaire de la matrice de V. En effet si la décomposition `LUP` de ${}^tV$ s'écrit ${}^tV = \begin{bmatrix} L_1 \end{bmatrix} \begin{bmatrix} U_1 & U_2 \end{bmatrix} P$, il suffit de poser

$$ {}^tB = \underbrace{\begin{bmatrix} L_1 & \\ 0 & I_{n-k} \end{bmatrix}}_{\overline{L}} \underbrace{\begin{bmatrix} U_1 & U_2 \\ & I_{n-k} \end{bmatrix}}_{\overline{U}} P = \begin{bmatrix} {}^tV \\ \begin{bmatrix} 0 & I_{n-k} \end{bmatrix} P \end{bmatrix} \tag{4.1} $$

On en déduit directement l'algorithme 4.2.

Algorithme 4.2 : `ComplèteBase` : Complétion d'une famille de vecteurs libres en une base

Données : V une matrice $n \times k$ de rang k ($k \leq n$) sur un corps
Résultat : B une matrice $n \times n$ inversible dont les k premières colonnes sont celles de V
début
  $(L, Q, U, P) = \text{LQUP}({}^tV)$; `/* Q = Id car V est de rang k */`
  retourner $B = \begin{bmatrix} V & {}^tP \begin{bmatrix} 0 \\ I_{n-k} \end{bmatrix} \end{bmatrix}$
fin

4.3 Une base du noyau

La décomposition `LQUP` permet aussi d'obtenir facilement une base du noyau d'une matrice rectangulaire ou non inversible.

Lemme 4.3. *Soit A une matrice $m \times n$ de rang r, avec $m \leq n$ et $(L, Q, \begin{bmatrix} U_1 & U_2 \\ 0 & 0 \end{bmatrix}, P)$ sa décomposition `LQUP`, où U_1 est $r \times r$.*

Alors les vecteurs colonne de la matrice ${}^tP \begin{bmatrix} U_1^{-1}U_2 \\ -I_{n-r} \end{bmatrix}$ forment une base du noyau de A et on peut les calculer en au plus

$$T_{noyau}(m,n) = \frac{C_\omega}{2^{\omega-2}-1} m^{\omega-1} n - \frac{C_\omega}{2}\left(\frac{1}{2^{\omega-2}-1} + \frac{1}{2^{\omega-1}-1}\right) m^\omega$$

opérations algébriques sur le corps de base.

Démonstration.

$$A\,{}^tP \begin{bmatrix} U_1^{-1}U_2 \\ -I_{n-r} \end{bmatrix} = LQ[0] = [0]$$

Et $T_{\text{noyau}}(m,n) = T_{\texttt{LQUP}}(m,n) + T_{\texttt{trsm}}(m, n-m)$. □

Avec le produit matriciel classique, $T_{\text{noyau}}(m,n) = 2m^2n - \frac{4}{3}m^3$.

Enfin, pour calculer une base du noyau à gauche de la matrice, il suffit d'appliquer cette méthode sur la transposée de A.

4.4 Les multiplications triangulaires

Afin d'obtenir les constantes les plus faibles possible, dans l'expression des complexités, il faut tenir compte de la structure des matrices manipulées. En particulier, nous nous intéressons

ici à la multiplication de matrices triangulaires. Avec le produit matriciel classique, on réduit la constante à sa moitié pour la multiplication d'une matrice triangulaire avec une matrice carrée et au tiers pour la multiplication de deux matrices triangulaires. Avec le produit matriciel rapide, ces gains sont réduits, comme nous allons le montrer.

4.4.1 Multiplication triangulaire avec une matrice rectangulaire

On considère l'opération $C = U \times B$ où la matrice U est triangulaire supérieure, de dimension $m \times m$ et la matrice B est rectangulaire de dimension $m \times n$ avec $m \leq n$. Cette opération est désignée par `trmm` (Triangular Matrix Multiplication) dans la terminologie des BLAS.

La réduction au produit matriciel, se fait par un découpage récursif similaire à celui de l'opération `trsm` :

$$\begin{bmatrix} U_1 & V \\ 0 & U_2 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix}.$$

On effectue successivement les opérations suivantes :

1. $C_2 \leftarrow U_2 B_2$
2. $C_1 \leftarrow U_1 B_1$
3. $C_1 \leftarrow V B_2 + C_1$

Lemme 4.4. *Le terme dominant de la complexité algébrique de l'algorithme* `trmm` *est*

$$T_{\texttt{trmm}}(m,n) = \begin{cases} \frac{C_\omega}{2(2^{\omega-2}-1)} m^{\omega-1} n & \textit{si } m \leq n, \\ \frac{C_\omega}{2(2^{\omega-2}-1)} n^{\omega-2} m^2 & \textit{si } n \leq m. \end{cases}$$

Ce dernier vaut $m^2 n$ avec la multiplication matricielle classique.

Démonstration. La relation de récurrence $T_{\texttt{trmm}}(m,n) = 2T_{\texttt{trmm}}(m/2,n) + T_{\texttt{MM}}(m/2,m/2,n)$ est similaire à celle du lemme 3.1. La preuve est donc identique. □

4.4.2 Mutliplication triangulaire supérieure - inférieure

La multiplication par blocs d'une matrice triangulaire supérieure par une matrice triangulaire inférieure s'écrit sous la forme suivante :

$$\begin{bmatrix} A_1 & A_2 \\ & A_4 \end{bmatrix} \begin{bmatrix} B_1 & \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 B_1 + A_2 B_3 & A_2 B_4 \\ A_4 B_3 & A_4 B_4 \end{bmatrix}.$$

Lemme 4.5. *Le terme dominant de la complexité algébrique de cet algorithme est*

$$T_{\texttt{UL}}(n) = \frac{C_\omega}{2^{3-\omega}(2^{\omega-2}-1)(2^{\omega-1}-1)} n^\omega.$$

Ce dernier vaut $\frac{2}{3} n^3$ avec la multiplication matricielle classique.

Démonstration. D'après le lemme 0.1, $T_{\mathtt{UL}}(n) = \alpha n^\omega$. La formule de récurrence

$$T_{\mathtt{UL}}(n) = 2T_{\mathtt{UL}}(n/2) + 2T_{\mathtt{trmm}}(n/2, n/2) + T_{\mathtt{MM}}(m/2, m/2, m/2)$$

donne $\alpha = \frac{C_\omega}{2^{3-\omega}(2^{\omega-2}-1)(2^{\omega-1}-1)}$. □

Ce résultat vaut aussi pour le produit `LU` d'une matrice triangulaire inférieure par une matrice triangulaire supérieure.

4.4.3 Multiplication triangulaire supérieure - supérieure

La multiplication par blocs de deux matrices triangulaires supérieures s'écrit :

$$\begin{bmatrix} A_1 & A_2 \\ & A_4 \end{bmatrix} \begin{bmatrix} B_1 & B_2 \\ & B_4 \end{bmatrix} = \begin{bmatrix} A_1B_1 & A_1B_2 + A_2B_4 \\ & A_4B_4 \end{bmatrix}.$$

Lemme 4.6. *Le terme dominant de la complexité algébrique de cet algorithme est*

$$T_{UU}(n) = \frac{C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)} n^\omega.$$

Ce dernier vaut $\frac{1}{3}n^3$ avec la multiplication matricielle classique.

Démonstration. D'après le lemme 0.1, $T_{\mathtt{UU}}(n) = \alpha n^\omega$. La formule de récurrence

$$T_{\mathtt{UU}}(n) = 2T_{\mathtt{UU}}(n/2) + 2T_{\mathtt{trmm}}(n/2, n/2)$$

donne $\alpha = \frac{C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)}$. □

Ce résultat vaut aussi pour le produit `LL` de deux matrices triangulaires inférieures.

4.5 Élévation au carré

Nous ne détaillerons pas le cas du calcul de AA pour une matrice carrée A quelconque, pour lequel il n'y a pas de meilleure constante connue que celle du produit matriciel habituel. En revanche, des considérations de symétrie permettent de réduire le nombre d'opérations algébriques pour le calcul de la matrice de Gram ($A\ {}^tA$) d'une matrice carrée ou triangulaire, ainsi que pour le carré d'une matrice symétrique.

4.5.1 Matrice de Gram d'une matrice carrée

La matrice de Gram d'une matrice carrée A est la matrice $A\ {}^tA$. Nous étendons l'étude au calcul de la matrice $AD\ {}^tA$, faisant intervenir la matrice diagonale D. L'introduction de cette matrice diagonale est fréquente dans le contexte du préconditionnement. La décomposition en blocs s'écrit :

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \begin{bmatrix} D_1 & \\ & D_4 \end{bmatrix} \begin{bmatrix} {}^tA_1 & {}^tA_3 \\ {}^tA_2 & {}^tA_4 \end{bmatrix} = \begin{bmatrix} A_1D_1\,{}^tA_1 + A_2D_4\,{}^tA_2 & A_1D_1\,{}^tA_3 + A_2D_4\,{}^tA_4 \\ A_3D_1\,{}^tA_1 + A_4D_4\,{}^tA_2 & A_3D_1\,{}^tA_3 + A_4D_4\,{}^tA_4 \end{bmatrix}.$$

Lemme 4.7. *Le terme dominant de la complexité algébrique du calcul du produit $AD\,{}^tA$ est*

$$T_{ADTA}(n) = \frac{C_\omega}{2(2^{\omega-2}-1)} n^\omega.$$

Ce dernier vaut n^3 avec la multiplication matricielle classique.

Démonstration. $AD\,{}^tA$ étant symétrique, ses deux blocs antidiagonaux sont transposés. Un seul calcul est donc nécessaire pour les deux. Quant aux blocs diagonaux, ils sont calculés récursivement. On a la formule de récurrence $T_{\mathtt{ADTA}}(n) = 4T_{\mathtt{ADTA}}(n/2) + 2T_{\mathtt{MM}}(n/2, n/2, n/2) + 5(n/2)^2$. D'après le lemme 0.1, $T_{\mathtt{ADTA}}(n) = \alpha n^\omega$ et on trouve $\alpha = \frac{C_\omega}{2(2^{\omega-2}-1)}$. □

Remarque 4.1. Pour une matrice A rectangulaire, de dimensions $m \times n$,on vérifie que

$$T_{\mathtt{ADTA}}(m,n) = \begin{cases} \frac{C_\omega}{2(2^{\omega-2}-1)} m^{\omega-1} n & \text{si } m \leq n, \\ \frac{C_\omega}{2(2^{\omega-2}-1)} n^{\omega-2} m^2 & \text{si } n \leq m. \end{cases}$$

4.5.2 Carré d'une matrice symétrique

Lorsque la matrice A est symétrique et que la matrice D est la matrice identité, on peut améliorer encore la constante. En effet, dans ce cas, $A_2 = {}^tA_3$ et on peut économiser le calcul de $A_3\,{}^tA_3$.

Lemme 4.8. *Le terme dominant de la complexité algébrique du calcul de A^2 pour A symétrique est*

$$T_{SymAA}(n) = \frac{(2^\omega - 3)C_\omega}{4(2^{\omega-2}-1)(2^{\omega-1}-1)} n^\omega.$$

Ce dernier vaut $\frac{5}{6}n^3$ avec la multiplication matricielle classique.

Démonstration. Les deux produits $A_1\,{}^tA_1$ et $A_4\,{}^tA_4$ sont effectués par des appels récursifs et le produit $A_2\,{}^tA_2$ est effectué par l'algorithme ADTA de la partie précédente (A_2 n'est pas symétrique). On a donc la formule de récurrence $T_{\mathtt{SymAA}}(n) = 2T_{\mathtt{SymAA}}(n/2) + T_{\mathtt{ADTA}}(n/2) + 2T_{\mathtt{MM}}(n/2, n/2, n/2) + 5(n/2)^2$. D'après le lemme 0.1, $T_{\mathtt{ADTA}}(n) = \alpha n^\omega$ et on trouve $\alpha = \frac{(2^\omega-3)C_\omega}{4(2^{\omega-2}-1)(2^{\omega-1}-1)}$. □

4.5.3 Matrice de Gram d'une matrice triangulaire

Ici, la matrice A est triangulaire inférieure, mais le résultat vaut aussi pour une matrice triangulaire supérieure. A nouveau, une matrice diagonale peut être insérée dans le produit. Le produit en blocs s'écrit :

$$\begin{bmatrix} L_1 & \\ L_3 & L_4 \end{bmatrix} \begin{bmatrix} D_1 & \\ & D_4 \end{bmatrix} \begin{bmatrix} {}^tL_1 & {}^tL_3 \\ & {}^tL_4 \end{bmatrix} = \begin{bmatrix} L_1 D_1\,{}^tL_1 & L_1 D_1\,{}^tL_3 \\ L_3 D_1\,{}^tL_1 & L_3 D_1\,{}^tL_3 + L_4 D_4\,{}^tL_4 \end{bmatrix}.$$

Lemme 4.9. *Le terme dominant de la complexité algébrique du calcul du produit $LD\,{}^tL$ pour L triangulaire est*

$$T_{LDTL}(n) = \frac{C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)} n^\omega.$$

Ce dernier vaut $\frac{1}{3}n^3$ avec la multiplication matricielle classique.

Démonstration. Les produits $L_1 D_1\,{}^tL_1$ et $L_4 D_4\,{}^tL_4$ sont effectués par des appels récursifs, le produit $L_3 D_1\,{}^tL_3$, par l'algorithme `ADTA` et $L_3 D_1\,{}^tL_1$ par `trmm`. Ainsi

$$T_{\texttt{LDTL}}(n) = 2T_{\texttt{LDTL}}(n/2) + T_{\texttt{ADTA}}(n/2) + T_{\texttt{trmm}}(n/2, n/2).$$

D'après le lemme 0.1, $T_{\texttt{LDTL}}(n) = \alpha n^\omega$ et en injectant dans la relation de récurrence précédente, on a $\alpha = \frac{C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)}$. □

4.6 Factorisation de matrices symétriques

Dans le cas d'une matrice symétrique, on privilégie une décomposition triangulaire qui respecte cette structure symétrique. C'est pourquoi la factorisation de type `LU` est remplacée par une factorisation de type `LDTL` ($A = LD\,{}^tL$) où L est triangulaire inférieure et D est diagonale (voir l'ouvrage de Golub et Van Loan [57] à ce sujet). Nous présentons ici l'algorithme de décomposition pour une matrice de profil de rang générique. Il se généralise pour les matrices quelconques en introduisant des permutations. Leur coût n'entre pas en compte dans le terme dominant de la complexité, c'est pourquoi nous les omettons par souci de simplicité.

L'algorithme repose sur la décomposition en blocs suivante :

$$A = \begin{bmatrix} A_1 & A_2 \\ {}^tA_2 & A_4 \end{bmatrix} = \begin{bmatrix} L_1 & \\ G & L_2 \end{bmatrix} \times \begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix} \times \begin{bmatrix} {}^tL_1 & {}^tG \\ & {}^tL_2 \end{bmatrix}.$$

Lemme 4.10. *Le terme dominant de la complexité algébrique de l'algorithme 4.3 est*

$$T_{LDTLdec}(n) = \frac{C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)} n^\omega.$$

Ce dernier vaut $\frac{1}{3}n^3$ avec la multiplication matricielle classique.

Démonstration. Les opérations 7 et 10 sont effectuées par deux appels récursifs, l'opération 8 par un `trsm` et l'opération 9 par un produit `ADTA`. On a donc la relation de récurrence : $T_{\texttt{LDTLdec}}(n) = 2T_{\texttt{LDTLdec}}(n/2) + T_{\texttt{trsm}}(n/2) + T_{\texttt{ADTA}}(n/2)$. D'après le lemme 0.1, $T_{\texttt{ADTA}}(n) = \alpha n^\omega$ et on trouve $\alpha = \frac{C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)}$. □

4.7 Inversion de matrices

4.7.1 Inverse d'une matrice triangulaire

On utilise à nouveau une décomposition par blocs :

$$\begin{bmatrix} A_1 & A_2 \\ & A_4 \end{bmatrix}^{-1} = \begin{bmatrix} A_1^{-1} & -A_1^{-1}A_2A_4^{-1} \\ & A_4^{-1} \end{bmatrix}.$$

Algorithme 4.3 : `LDTLdec` : décomposition triangulaire symétrique

Données : A, une matrice carrée symétrique d'ordre n de profil de rang générique
Résultat : L triangulaire inférieure unitaire et D diagonale telles que

début
  $A = LD\,{}^tL$
  si *n=1* **alors**
    retourner $(L = [1], D = A)$
  sinon
    Découper $A = \begin{bmatrix} A_1 & A_2 \\ {}^tA_2 & A_4 \end{bmatrix}$ où A_1 est $n/2 \times n/2$
    $(L_1, D_1) = \texttt{LDTLdec}(A_1)$
    $G = {}^t(D_1^{-1}L_1^{-1}A_2)$
    $H = A_4 - GD_1\,{}^tG$
    $(L_2, D_2) = \texttt{LDTLdec}(H)$
    retourner $\left(L = \begin{bmatrix} L_1 & \\ G & L_2 \end{bmatrix}, D = \begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix}\right)$
fin

Lemme 4.11. *Le terme dominant de la complexité algébrique de l'inversion d'une matrice triangulaire est*

$$T_{trINV}(n) = \frac{C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)}n^\omega.$$

Ce dernier vaut $\frac{1}{3}n^3$ avec la multiplication matricielle classique.

Démonstration. L'algorithme consiste en deux appels récursifs pour le calcul de A_1^{-1} et A_4^{-1} et deux `trmm` pour le calcul de $-A_1^{-1}A_2A_4^{-1}$. On a donc la relation de récurrence suivante : $T_{\texttt{trINV}}(n) = 2T_{\texttt{trINV}}(n/2) + 2T_{\texttt{trmm}}(n/2)$. D'après le lemme 0.1, $T_{\texttt{trINV}}(n) = \alpha n^\omega$ et on trouve

$$\alpha = \frac{C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)}.$$

□

4.7.2 Inverse d'une matrice quelconque

L'inversion d'une matrice quelconque se fait par l'intermédiaire d'une décomposition triangulaire. Dans le cas d'une matrice inversible, les décompositions `LQUP` et `LUP` sont confondues. Il reste à inverser la matrice L avec l'algorithme `trINV`, résoudre $UX = L^{-1}$ avec un `trsm`, et appliquer la permutation ${}^tP$ à droite.

Lemme 4.12. *Le terme dominant de la complexité algébrique de l'inversion d'une matrice*

inversible quelconque est

$$T_{INV}(n) = \frac{3C_\omega}{2^{3-\omega}(2^{\omega-2}-1)(2^{\omega-1}-1)}n^\omega.$$

Ce dernier vaut $2n^3$ avec la multiplication matricielle classique.

Démonstration.

$$T_{\mathtt{INV}}(n) = T_{\mathtt{LQUP}}(n,n) + T_{\mathtt{trINV}}(n) + T_{\mathtt{trsm}}(n,n) = \frac{3C_\omega}{2^{3-\omega}(2^{\omega-2}-1)(2^{\omega-1}-1)}n^\omega.$$

□

4.7.3 Inverse d'une matrice symétrique

Si A est symétrique, on remplace la factorisation `LU` par la factorisation `LDTL`.

Lemme 4.13. *Le terme dominant de la complexité algébrique de l'inversion d'une matrice symétrique est*

$$T_{symINV}(n) = \frac{3C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)}n^\omega$$

Ce dernier vaut n^3 avec la multiplication matricielle classique.

Démonstration. Après la décompostion `LDTL`, il reste à inverser la matrice triangulaire avec `trINV` puis à calculer le produit ${}^tL^{-1}D^{-1}L^{-1}$ avec `LDTL`.

Soit, $T_{\mathtt{symINV}}(n) = T_{\mathtt{LDTLdec}}(n) + T_{\mathtt{trINV}}(n) + T_{\mathtt{LDTL}}(n) = \frac{3}{2(2^{\omega-1}-1)(2^{\omega-2}-1)}n^\omega$ □

4.7.4 Pseudo inverse de Moore-Penrose

Matrices de rang plein

Soit A une matrice rectangulaire de dimensions $m \times n$ avec $m \leq n$ et de rang plein. On définit le pseudo–inverse de Moore-Penrose de A par $A^\dagger = {}^tA(A\,{}^tA)^{-1}$. Voir [97] et les références incluses à ce sujet.

Lemme 4.14. *Le terme dominant de la complexité algébrique du calcul de l'inverse de Moore-Penrose d'une matrice de dimensions $m \times n$, avec $m \leq n$ et de rang plein est*

$$T_{MPINV}(m,n) = \frac{3C_\omega}{2(2^{\omega-2}-1)}m^{\omega-1}n + \frac{C_\omega}{2(2^{\omega-1}-1)(2^{\omega-2}-1)}m^\omega.$$

Ce dernier vaut $3m^2n + \frac{1}{3}m^3$ avec la multiplication matricielle classique.

Démonstration. Pour calculer cette matrice, il suffit d'effectuer la décomposition `LDTL` de la matrice symétrique $A\,{}^tA$, suivie de deux `trsm` :

$$T_{\mathtt{MPINV}}(m,n) = T_{\mathtt{ADTA}}(m,n) + T_{\mathtt{LDTL}}(m) + 2T_{\mathtt{trsm}}(m,n).$$

On trouve alors

$$T_{\mathtt{MPINV}}(m,n) = \frac{3C_\omega}{2(2^{\omega-2}-1)}m^{\omega-1}n + \frac{C_\omega}{2(2^{\omega-1}-1)(2^{\omega-2}-1)}m^\omega.$$

□

Remarque 4.2. Ce dernier algorithme correspond à la résolution en calcul numérique des équations normales [57, algorithme 5.3.1].

Matrices de rang quelconque

Pour une matrice A de rang $r < n$, [89] donne une méthode utilisant une décomposition de A en produit de deux matrices de rang r.

Propriété 4.1 (Noble 1966)**.** *Soit A une matrice $m \times n$ de rang r. $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ où A_{11} est $r \times r$ et est inversible.*

Alors A peut s'écrire sous la forme $A = \begin{bmatrix} I \\ M \end{bmatrix} A_{11} \begin{bmatrix} I & N \end{bmatrix}$ et l'inverse généralisé de Moore-Penrose de A vérifie

$$A^\dagger = \begin{bmatrix} I \\ {}^tN \end{bmatrix} [(I + {}^tMM)A_{11}(I + N\,{}^tN)]^{-1} \begin{bmatrix} I & {}^tM \end{bmatrix}.$$

L'auteur propose d'appliquer une modification de la décomposition de Gauss-Jordan pour calculer cette matrice. A nouveau, nous proposons de la remplacer par la décomposition `LQUP` de la matrice A comme suit.

Soit (L, Q, U, P) la décomposition `LQUP` A quelconque. La première étape consiste à donner à A le profil de rang générique. En suivant la technique de la partie 4.1.2, on pose $\overline{A} = {}^tQA\,{}^tP$. On décompose en blocs les matrices suivantes : ${}^tQLQ = \begin{bmatrix} L_1 \\ G \end{bmatrix}$ et $U = \begin{bmatrix} U_1 & Y \end{bmatrix}$. En posant $M = GL_1^{-1}$ et $N = U_1^{-1}Y$, on retrouve la décomposition de $\overline{A}$ et par conséquent, on obtient

$$A^\dagger = \begin{bmatrix} I \\ {}^tY\,{}^tU_1{}^{-1} \end{bmatrix} \left((L_1 + {}^tL_1{}^{-1}\,{}^tGG)(U_1 + Y\,{}^tYU_1^{-1})\right)^{-1} [I|\,{}^tL_1{}^{-1}\,{}^tG]. \tag{4.2}$$

On calcule ensuite $W = (L_1 + L_1^{-T}\,{}^tGG)(U_1 + Y\,{}^tYU_1^{-1})$ par deux `ADTA` et deux `trsm` et un produit matriciel. Pour le calcul de W^{-1}, il est préférable de passer par une décomposition `UL` que l'on définit de la même manière que la décomposition `LU`, en permutant l'aspect supérieur ou inférieur des matrices. On calcule ainsi successivement la décomposition $W = U_W L_W$, puis ${}^tL_1U_W$ et $L_W\,{}^tU_1$ par deux produits de matrices triangulaires supérieur-supérieur. On calcule ensuite $H = ({}^tL_1U_W)^{-1}\,{}^tG$ et $Z = {}^tY(L_W\,{}^tU_1)^{-1}$ par deux `trsm`.

Pour obtenir $A^\dagger = \begin{bmatrix} W^{-1} & L_W^{-1}H \\ ZU_W^{-1} & ZH \end{bmatrix}$, on calcule W^{-1} par deux inverses de matrices triangulaires et un produit supérieur-inférieur, ZH par un produit matriciel, et les deux blocs restants par deux `trsm`.

Au final, on a

$$\begin{aligned} T_{\texttt{MPINV}}(m,n) &= T_{\texttt{LQUP}}(m,n) + T_{\texttt{ADTA}}(r,m-r) + T_{\texttt{ADTA}}(r,n-r) + 2T_{\texttt{trsm}}(r,r) \\ &\quad + T_{\texttt{MM}}(r,r,r) + T_{\texttt{LQUP}}(r,r) + 2T_{\texttt{UU}}(r) + 2T_{\texttt{trINV}}(r) + 2T_{\texttt{trsm}}(r,n-r) \\ &\quad + 2T_{\texttt{trsm}}(r,m-r) + T_{\texttt{UL}}(r) + T_{\texttt{MM}}(m-r,r,n-r). \end{aligned}$$

Nous ne pouvons exprimer cette complexité en fonction de ω car elle dépend du signe de $m-2r$ et de $n-2r$ (pour les `trsm`). En revanche, avec la multiplication classique, on obtient

$$T_{\texttt{MPINV}}(m,n) = m^2n - \frac{1}{3}m^3 + \frac{8}{3}r^3 + r^2m + r^2n + 2mnr.$$

A titre de comparaison, le calcul basé sur la décomposition de Cholesky, de $A\,{}^tA$ présenté dans [19] demande $3m^2n + 2r^2m + 3r^3$ opérations algébriques. En prenant $m = n = r$, on obtient $8n^3$ contre $\left(7 + \frac{1}{3}\right) n^3$ pour notre décomposition, et cette dernière est d'autant plus avantageuse que r est petit. Enfin les méthodes basées sur la décomposition en valeurs singulières sont encore pire [57].

4.7.5 Performances expérimentales

Comme nous l'avons fait dans les parties 3.1.4 et 3.2.3, nous comparons les performances des routines d'inversion matricielle de matrices triangulaires ou carrées avec leur équivalent numérique ainsi qu'avec la multiplication matricielle. La comparaison avec les routines numériques est toujours basée sur la bibliothèque LAPACK ainsi que deux BLAS différents (ATLAS et GOTO). Ici encore, nous ne donnons pas les résultats de l'inversion triangulaire numérique sur Pentium4 avec ATLAS à cause des mauvaises performances de la routine `dtrsm`. Nous utilisons toujours l'implémentation de corps fini `Modular<double>` pour représenter le corps Z_{65521}.

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
GOTO	tri. inv	0.10s	0.65s	1.99s	8.38s	22.00s	32.34s	45.67s	62.01s
	dtrtri	0.19s	1.06s	2.97s	11.27s	27.89s	40.13s	55.43s	74.44s
	$\frac{tri.inv}{dtrtri}$	**0.52**	**0.61**	**0.67**	**0.74**	**0.78**	**0.80**	**0.82**	**0.83**

TAB. 4.1 – *Temps de calcul de l'inversion de matrices triangulaires sur un P4, 3.4GHz*

Les tableaux 4.1 et 4.2 montrent les performances de la routine exacte d'inversion triangulaire, en comparaison avec celles de la routine `dtrtri` de LAPACK. Les résultats dépendent fortement du BLAS utilisé : avec le BLAS ATLAS, la routine numérique est meilleure, mais l'écart tend à se réduire quand la dimension augmente. A l'opposé, avec GOTO, la routine exacte parvient même à être plus rapide que `dtrtri` ; mais le ratio tend lui aussi vers 1.

Les tableaux 4.3 et 4.4 reprennent cette comparaison pour les routines d'inversion de matrices carrées. L'inversion matricielle en numérique est réalisée par la succession des opérations `dgetrf` pour la factorisation LU et `dgetri` pour l'inversion des matrices triangulaires. On constate que seule l'implémentation utilisant le BLAS GOTO sur l'architecture Itanium 2

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	tri. inv	0.20s	1.11s	3.17s	12.67s	32.05s	46.26s	64.96s	86.27s
	dtrtri	0.10s	0.75s	2.42s	10.66s	28.26s	41.65s	58.83s	79.21s
	$\frac{tri.inv}{dtrtri}$	**2.00**	**1.48**	**1.30**	**1.18**	**1.13**	**1.11**	**1.10**	**1.08**
GOTO	tri. inv	0.14s	0.84s	2.50s	10.20s	26.37s	38.61s	54.15s	73.34s
	dtrtri	0.16s	0.94s	2.72s	10.83s	27.57s	40.14s	56.16s	75.74s
	$\frac{tri.inv}{dtrtri}$	**0.87**	**0.89**	**0.91**	**0.94**	**0.95**	**0.96**	**0.96**	**0.96**

TAB. 4.2 – *Temps de calcul de l'inversion de matrices triangulaires sur un Itanium2, 1.3GHz*

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*10000*
ATLAS	inverse	0.88s	5.22s	15.00s	59.03s	152.20s	218.15s	394.56s
	dgetrf+dgetri	0.55s	4.06s	13.85s	64.51s	176.34s	298.89s	604.83s
	$\frac{inverse}{dgetrf+dgetri}$	**1.6**	**1.28**	**1.08**	**0.91**	**0.86**	**0.72**	**0.65**
GOTO	inverse	0.78s	4.66s	13.63s	54.15s	136.55s	197.26s	362.91s
	dgetrf+dgetri	0.48s	3.52s	11.81s	62.68s	137.21s	201.81s	477.27s
	$\frac{inverse}{dgetrf+dgetri}$	**1.62**	**1.32**	**1.15**	**0.86**	**0.99**	**0.97**	**0.76**

TAB. 4.3 – *Temps de calcul de l'inversion de matrices sur un P4, 3.4GHz*

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*10000*
ATLAS	inverse	1.22s	7.02s	20.13s	80.91s	201.58s	289.50s	538.93s
	dgetrf+dgetri	0.65s	4.83s	15.88s	73.17s	206.11s	308.30s	603.26s
	$\frac{inverse}{dgetrf+dgetri}$	**1.87**	**1.45**	**1.26**	**1.10**	**0.97**	**0.93**	**0.89**
GOTO	inverse	1.05s	6.14s	18.10s	73.85s	180.21s	258.17s	496.39s
	dgetrf+dgetri	0.71s	4.56s	13.96s	59.26s	155.71s	230.29s	440.82s
	$\frac{inverse}{dgetrf+dgetri}$	**1.47**	**1.34**	**1.29**	**1.24**	**1.15**	**1.12**	**1.12**

TAB. 4.4 – *Temps de calcul de l'inversion de matrices sur un Itanium2, 1.3GHz*

donne l'avantage au calcul numérique pour toutes les tailles de matrices considérées. Dans les autres cas, le produit matriciel rapide permet à la routine de calcul exact d'être plus rapide pour des tailles de matrices suffisamment grandes.

Pour illustrer la corrélation entre ces routines d'inversion matricielle et la multiplication matricielle, nous montrons dans les figures 4.1 et 4.2 les rapports des temps de calcul entre les routines d'inversion et de multiplication matricielle. D'après 4.7.1, le ratio théorique entre les termes dominant des complexités algébriques de l'inversion triangulaire et du produit matriciel est

$$\frac{1}{2(2^{\omega-2}-1)(2^{\omega-1}-1)}.$$

Ce dernier vaut $1/6$ pour $\omega = 3$ et environ $0,267$ pour $\omega = \log_2(7)$ de l'algorithme de Winograd. En pratique nos routines utilisent un produit matriciel hybride entre l'algorithme classique et l'algorithme de Winograd (voir partie 2.2.1) et il faut donc s'attendre à un ratio asymptotique compris entre $1/6$ et $0,267$. On vérifie effectivement sur la figure 4.1 que ce ratio est bien inférieur à $0,267$ pour de grandes valeurs de n pour la plupart des implémentations.

Pour la figure 4.2, le ratio théorique est

$$\frac{3}{2^{3-\omega}(2^{\omega-2}-1)(2^{\omega-1}-1)}$$

et vaut 1 pour $\omega = 3$ et $1,4$ pour $\omega = \log_2(7)$. En pratique, les ratios tendent vers $1,4$ mais restent cette fois supérieurs. Ceci s'explique sans doute par le fait que l'algorithme d'inversion de matrices carrées est plus sophistiqué et fait appel à de nombreuses routines. Cela a pour conséquence de réduire le nombre de produits matriciels de grandes dimensions et donc de réduire l'apport de l'algorithme de Winograd mais aussi des routines BLAS.

Nous résumons les résultats présentés dans cette partie dans le tableau 4.5. Pour chaque algorithme, nous donnons l'expression de la constante K_ω du terme dominant de sa complexité algébrique. Pour des raisons de simplicité, les matrices sont supposées carrées. A titre d'exemple nous donnons les valeurs de K_ω en utilisant deux algorithmes de produit matriciel : le produit classique ($\omega = 3, C_\omega = 2$) et le produit de Winograd ($\omega = \log_2 7 \approx 2,808, C_\omega = 6$).

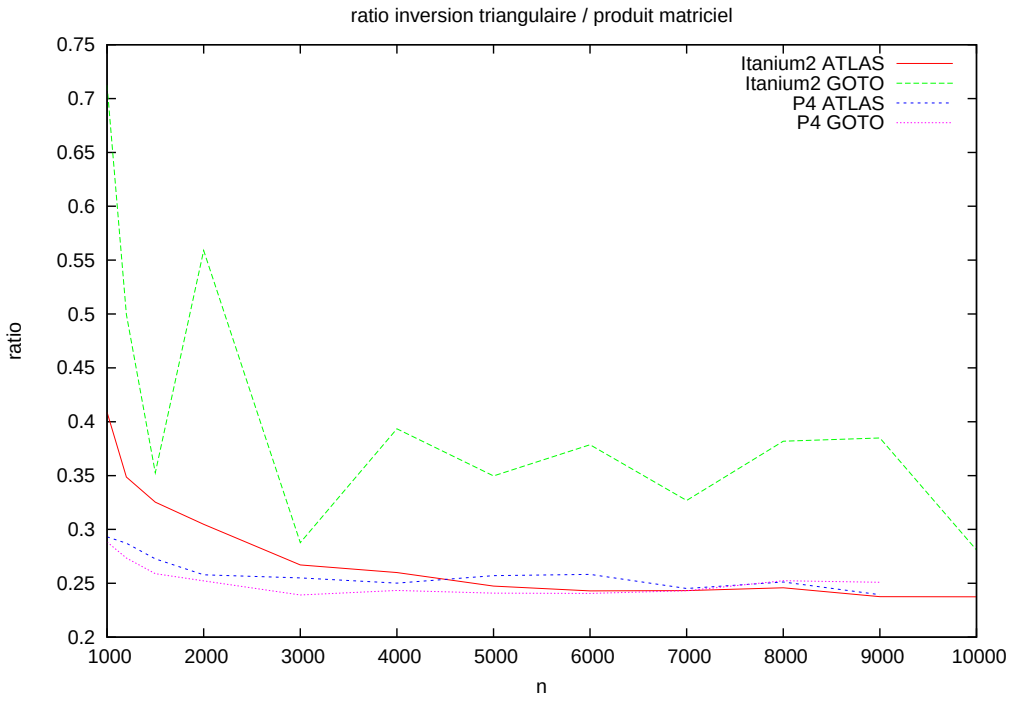


FIG. 4.1 – *Ratios des temps de calcul entre l'inversion triangulaire et le produit matriciel*

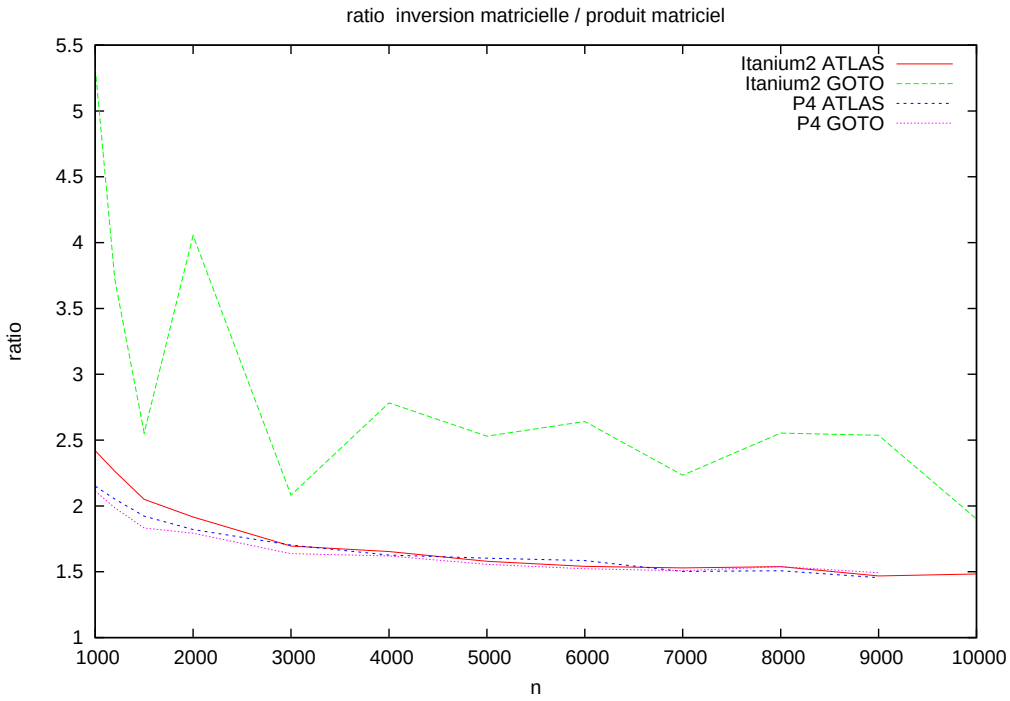


FIG. 4.2 – *Ratios des temps de calcul entre l'inversion et le produit matriciel*

Nom	Opération	K_ω	K_3	$K_{\log_2 7}$
MM	$C \leftarrow AB$	C_ω	2	6
trsm trmm	$X \leftarrow U^{-1}B$ $X \leftarrow UB$	$\frac{C_\omega}{2(2^{\omega-2}-1)}$	1	4
LUdec UL	$(L,U) \leftarrow A$ $X \leftarrow UL$	$\frac{C_\omega}{2(2^{\omega-2}-1)} - \frac{C_\omega}{2(2^{\omega-1}-1)}$	0,667	2,8
UU LDTL LDTLdec trINV	$X \leftarrow UU$ $X \leftarrow LD\,{}^tL$ $(L,D) \leftarrow A$ $X \leftarrow U^{-1}$	$\frac{C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)}$	0,333	1,6
ADTA	$X \leftarrow AD\,{}^tA$	$\frac{C_\omega}{2(2^{\omega-2}-1)}$	1	4
SymAA	$X \leftarrow AA$, A symétrique	$\frac{2^\omega-3)C_\omega}{4(2^{\omega-2}-1)(2^{\omega-1}-1)}$	0,83	3,2
INV	$X \leftarrow A^{-1}$	$\frac{3C_\omega}{2^{3-\omega}(2^{\omega-2}-1)(2^{\omega-1}-1)}$	2	8,4
SymINV	$X \leftarrow A^{-1}$, A symétrique	$\frac{3C_\omega}{2(2^{\omega-2}-1)(2^{\omega-1}-1)}$	1	4,8
MPINV	$X \leftarrow A^\dagger$	$\frac{3C_\omega}{2(2^{\omega-2}-1)} + \frac{C_\omega}{2(2^{\omega-1}-1)(2^{\omega-2}-1)}$	3,333	13,1

TAB. 4.5 – *Récapitulatif des constantes des complexités algébriques présentées*

5

APPLICATION AUX PROBLÈMES MAL CONDITIONNÉS

Au delà des champs d'application propres aux corps finis, l'efficacité de ces routines ouvre de nouvelles perspectives dans le traitement de certains problèmes en calcul numérique. Nous nous intéressons en particulier ici à la résolution de systèmes et à l'inversion de matrices.

Dans ce type de problèmes, les approximations faites sur les coefficients des entrées, ainsi que dans les méthodes de résolution approchée, peuvent avoir des répercussions plus ou moins importantes sur la qualité d'approximation de la solution obtenue. Le *nombre de conditionnement* d'une matrice A permet de borner en particulier l'erreur relative de la solution du système $Ax = b$ en fonction d'une erreur relative commise sur le vecteur b. On le définit par $\kappa(A) = \|A\| \cdot \|A^{-1}\|$. Il vérifie

$$\frac{\|x + \Delta x\|}{\|x\|} \leq \kappa(A) \frac{\|b + \Delta b\|}{\|b\|}. \tag{5.1}$$

Il permet aussi de borner l'erreur relative de la solution quand A et b sont perturbés :

$$\frac{\|x + \Delta x\|}{\|x\|} \leq \frac{\kappa(A)}{1 - \kappa(A)\frac{\|\Delta A\|}{\|A\|}} \left(\frac{\|A + \Delta A\|}{\|A\|} + \frac{\|b + \Delta b\|}{\|b\|} \right).$$

Ainsi, pour une erreur relative donnée sur les entrées, l'erreur relative sur la solution sera d'autant plus petite que le conditionnement de la matrice sera petit. On pourra se référer par exemple à l'ouvrage de Higham [65] sur le sujet.

Dans la résolution de problèmes numériques, les données sont le plus souvent approchées, et induisent de facto une certaine erreur dans la solution qu'on peut espérer obtenir du problème réel. Il s'agit là d'une limitation inhérente à cette approche, dont on ne peut s'affranchir.

En revanche, les méthodes numériques choisies pour la résolution sont basées sur une arithmétique de $\mathbb{R}$ approchée. Elles introduisent donc de nouvelles approximations dont l'effet sur la qualité de la solution dépend en général aussi du nombre de conditionnement. En effet, pour une fonction $\varphi(a)$, notons $\tilde{\varphi}(a)$ la valeur calculée par l'arithmétique approchée. Wilkinson [120] définit l'erreur inverse Δa comme la variation sur a qui aurait entraîné la même erreur sur $\varphi(a)$:

$$\tilde{\varphi}(a) = \varphi(a + \Delta a).$$

Ainsi, en étudiant l'erreur réciproque des algorithmes utilisés sur l'arithmétique approchée de $\mathbb{R}$, on peut appliquer la borne (5.1) pour majorer l'erreur dans la solution en fonction du nombre de conditionnement.

Par conséquent un mauvais conditionnement (grandes valeurs de $\kappa(A)$) peut rendre l'arithmétique approchée impraticable. De plus, on montre que lors de la résolution d'un système linéaire par une méthode itérative de type gradient conjugué, le conditionnement décrit aussi la rapidité de convergence d'un tel algorithme. Un mauvais conditionnement pénalise donc ces méthodes par une convergence trop lente. De nombreuses solutions basées sur des préconditionneurs, améliorant le conditionnement de la matrice ont été développées, pour contourner ces situations. Cependant, avec l'augmentation des dimensions, les problèmes mal conditionnés sont de plus en plus nombreux, comme le montre par exemple le survol de Schatzman [98] s'appuyant sur les travaux de Edelman [47] et Demmel [25].

Pour remédier à ce problème, la démarche générale consiste à augmenter la précision des calculs. Pour ce faire, diverses solutions peuvent être considérées. L'utilisation de nombres flottants multi-précision en fait partie. Les bibliothèques GMP ou MPFR[1] proposent des implémentations efficaces de l'arithmétique des flottants à précision arbitraire. A un niveau supérieur, la bibliothèque iRRAM permet d'utiliser ces implémentations en limitant les calculs pour ne garantir que la précision demandée par l'utilisateur sur les résultats. L'inconvénient de cette approche vient du fait que les flottants multiprécision ne peuvent être manipulés qu'au sein d'opérations scalaires. On ne peut donc bénéficier de l'efficacité des BLAS nécessitant des flottants de taille fixe (un ou deux mots machine).

Une autre approche développée par Demmel [24], concernant la résolution de systèmes linéaires, consiste à utiliser les flottants machine, mais en effectuant des raffinements itératifs des solutions. Il s'agit d'une démarche très récente, semblant prometteuse.

Une approche alternative consiste à utiliser le calcul exact. Son premier avantage est bien sûr de ne pas dépendre d'une arithmétique approchée. En outre elle permet de tirer profit de l'efficacité des BLAS, grâce aux routines que nous avons présentées dans les dernières parties. Enfin, pour des problèmes tels que la résolution de systèmes linéaires, le raffinement itératif exact de l'algorithme de Dixon assure une complexité cubique, équivalente à celle des méthodes numériques. Le principal inconvénient de cette méthode est qu'il faut calculer la solution exacte dans sa totalité, quelle que soit la précision demandée par l'utilisateur. Hormis la résolution de système linéaire, la résolution exacte des autres problèmes implique donc une complexité supérieure. Nous montrons dans cette partie que ce sur-coût reste cependant raisonnable ce qui justifie l'intérêt à porter à cette approche dans le cas de problèmes mal conditionnés.

5.1 Restes chinois ou raffinement itératif exact

Pour résoudre le problème de façon exacte, il faut d'abord le ramener à un problème entier équivalent. Si les coefficients flottants de la matrice A du problème ont un exposant compris entre $e_{\min}$ et $e_{\max}$ et que leur mantisse est de β bits, on considère la matrice $\tilde{A} = 2^{\beta - e_{\min}} A \in \mathbb{Z}$. Ses coefficients sont bornés par $2^{e_{\max} - e_{\min} + \beta}$. Pour fixer les idées, les systèmes extraits de

[1] www.mpfr.org

problèmes physiques vérifient en général $e_{\max} \leq 20$ et $e_{\min} \geq -20$. Les coefficients de $\tilde{A}$ sont donc représentés sur 93 bits.

La résolution exacte du problème entier se fait ensuite de deux façons différentes selon le problème :

Inversion de matrices : on utilise l'algorithme des Restes Chinois, que nous décrirons plus en détails dans la partie 9.1. Pour une matrice A, il consiste à calculer dans différents corps premiers $\mathbb{Z}_{p_i}$ la matrice $\det(A)A^{-1}$ et d'en déduire la valeur dans $\mathbb{Z}$ par une interpolation pour obtenir la matrice des numérateurs de A^{-1}. Le dénominateur commun étant évidemment $\det(A)$.

Résolution de systèmes : dans le cas particulier de la résolution de systèmes, on utilise la remontée p-adique de Dixon [26], qui permet de gagner un facteur n dans la complexité binaire, par rapport à la méthode des Restes Chinois. Cet algorithme fait appel à plusieurs résolutions de systèmes dans un corps fini $\mathbb{Z}_p$ pour calculer ainsi successivement les différents termes de la décomposition p-adique de la solution. En ce sens, il s'agit d'un algorithme de raffinement itératif, mais contrairement aux méthodes numériques, celui-ci débute par les termes les moins significatifs pour terminer par les plus significatifs.

Dans la suite nous présentons deux cas d'application où le calcul exact, par la méthode des restes chinois, permet de résoudre des problèmes réputés mal conditionnés pour les algorithmes numériques.

5.2 Calcul de l'inverse de la matrice de Hilbert

Il s'agit d'un exemple classique en calcul numérique de problème mal conditionné. On définit la matrice de Hilbert $H_n = [h_{i,j}]$ de dimensions $n \times n$ par

$$h_{i,j} = \frac{1}{i+j-1},\ 1 \leq i,j \leq n.$$

La formule exacte donnant la valeur du coefficient d'indices (i,j) de son inverse $H^{-1} = [\overline{h_{i,j}}]$ est connue :

$$\overline{h_{i,j}} = (-1)^{i+j}(i+j-1)\binom{n+i-1}{n-j}\binom{n+j-1}{n-i}\binom{i+j-2}{i-1}^2. \tag{5.2}$$

Mais cette matrice étant très mal conditionnée, le calcul de son inverse par les méthodes classiques d'analyse numérique représente un test pour les bibliothèques numériques multi-précision.

De la même façon nous cherchons aussi à calculer l'inverse de H de façon exacte par les algorithmes présentés dans cette partie. Pour ce faire, nous calculons d'abord la matrice H dans $\mathbb{Q}$. En appliquant l'algorithme des Restes Chinois, on calcule les inverses des matrices $H_{p_i} = H \mod p_i$ dans des corps finis premiers $\mathbb{Z}_{p_i}$, afin de reconstruire le résultat dans $\mathbb{Z}$. Pour déterminer le nombre de corps finis à utiliser, on calcule une approximation du logarithme du plus grand des coefficients de la matrice H^{-1} par la formule (5.2).

Nous n'avons pas pu effectuer ce calcul avec la bibliothèque iRRAM, et nous appuyons notre comparaison sur les temps de calcul donnés dans sa documentation [88, §15] (utilisant

Dimension	50	100	150	200	250	300	500	800
# nombre premiers	13	25	37	49	61	76	125	200
LINBOX (P4-3.2GHz)	0,103s	0,87s	3,19s	8,30s	16,84	30,83s	169s	775s
iRRAM (Athlon 800MHz)	3,2s	79s	457s	1200s	3052s			
Calcul direct	0,023s	0,215s	0,818	2,1s	4,46s	8,29s	51,6s	279s
ratio LINBOX/ direct	4,47	4,04	3,9	3,95	3,78	3,72	3,28	2,78

TAB. 5.1 – *Calcul exact de l'inverse de la matrice de Hilbert sur un Pentium4-3,2Ghz-1Go*

un Athlon 800Mhz). Nous les reportons dans le tableau 5.1 en les comparant à ceux de la bibliothèque LINBOX sur un Pentium4-3,2Ghz. Malgré la différence entre ces deux architectures, on peut constater avec le tableau 5.1 que les temps mis par la bibliothèque iRRAM restent nettement moins bons que ceux de LINBOX pour le calcul exact.

A titre de comparaison nous donnons le temps de calcul obtenu par la méthode directe, utilisant la formule (5.2) avec des entiers multiprécision. Ils sont inférieurs à ceux de LINBOX, mais on peut constater que le rapport entre ces temps se réduit quand la dimension croît. Il s'agit de l'avantage donné à LINBOX par l'utilisation des BLAS. En effet, comme nous l'avons montré dans le chapitre 2, le regroupement des opérations dans des produits matriciels augmente l'efficacité globale.

5.3 Le défi *More Digits*

Dans le cadre de la conférence RNC7, sur les nombres réels et les ordinateurs [62], la compétition amicale *More Digits* était organisée pour mesurer les performances de bibliothèques de calcul numérique en multiprécision. Une liste de 17 problèmes devaient être résolus, avec une précision imposée.

Parmi ceux-ci nous avons retenu le problème 10, d'algèbre linéaire :

Problème 5.1. *Soit $S \in \mathbb{N}$ et M une matrice de dimensions $A \times A$*

$$M = \begin{bmatrix} \chi_1 & \chi_2 & \cdots & \chi_A \\ \chi_{A+1} & \chi_{A+2} & \cdots & \chi_{2A} \\ \vdots & \vdots & & \vdots \\ \chi_{A^2-A+1} & \chi_{A^2-A+2} & \cdots & \chi_{A^2} \end{bmatrix}$$

où χ_n est le générateur pseudo-aléatoire de Marsaglia défini par

$$\begin{cases} \chi_1 &= S, \\ \chi_{i+1} &= (69069\chi_i + 3) \mod 2^{31}. \end{cases}.$$

Soit $M^{-1} = [\overline{m_{i,j}}]$, l'inverse de la matrice M et z la somme des valeurs absolues des coefficients de M^{-1}, $z = \sum_{i,j} |\overline{m_{i,j}}|$. Afficher les N décimales significatives de z.

Nous avons mis en œuvre la résolution exacte de ce problème pour calculer le numérateur et le dénominateur de z, ce qui permet ensuite de calculer toutes ses décimales.

Le dénominateur de z est simplement le déterminant de M. Pour calculer son numérateur, il suffit de calculer la matrice entière $\det(M)M^{-1}$ en utilisant l'algorithme des restes chinois et le calcul de l'inverse matricielle dans un corps fini, que nous décrirons dans la partie 4.7.1. Le déterminant s'obtient directement comme le produit des pivots de la décomposition `LUP` calculée pour le calcul de l'inverse. Ce n'est qu'une fois la matrice $\overline{M}$ reconstruite dans $\mathbb{Q}$, que l'on peut calculer la valeur de z. En effet la fonction valeur absolue n'étant pas définie dans un corps fini, on ne peut calculer directement les projections de z dans ces corps.

Dimension	LINBOX dans $\mathbb{Q}$		`iRRAM` : 10 décimales		`iRRAM` : 520 décimales	
A	# nb premiers	temps	temps	LINBOX/ `iRRAM`	temps	LINBOX/ `iRRAM`
50	82	1,01s	0,14s	7,21	1,03s	0,98
100	160	9,141s	1,19s	7,68	11,07s	0,83
200	312	85,23s	13,88s	6,14	98s	0,87
300	480	5m41s	56,1s	6,07	5m10s	1,1
500	814	32m40s	5m8s	6,35	30m20s	1,08

TAB. 5.2 – *Temps de calcul pour la résolution du problème 10 du défi More Digits sur un Pentium 4-3,2Ghz-1Go*

Les temps de calcul de notre implémentation sont reportés dans le tableau 5.2, pour des tailles de matrices considérées comme faciles (50, 100) ou difficiles (200, 300, 500) par l'énoncé. Pour ces calculs, nous avons utilisé la graine $S = 10$ générant à chaque fois une matrice inversible. A titre de comparaison, les temps de calcul de la bibliothèque `IRRAM` [88] sont aussi donnés pour le calcul approché du résultat, avec 10 ou 520 chiffres significatifs. On constate que le temps de calcul avec 10 chiffres significatifs est certes nettement inférieur à celui de LINBOX, mais reste dans un rapport constant. Pour 520 chiffres de précision, iRRAM nécessite autant de temps de calcul que LINBOX pour la solution exacte. Cette précision est inutile dans ce cas, car les numérateurs et dénominateurs de la solution rationnelle ont moins de 325 chiffres. Cependant, cela montre qu'il existe une précision limite au delà de laquelle le calcul exact est plus efficace que le calcul approché multiprécision.

Ainsi, lorsque l'on cherche à augmenter la précision dans la résolution de certains problèmes approchés, plusieurs approches sont envisageables. Pour certains problèmes comme la résolution de système linéaire, le raffinement itératif permet d'augmenter la précision tout en utilisant l'algèbre linéaire sur des mots machine, permettant l'utilisation des BLAS. Le calcul flottant multiprécision permet d'aborder tous les problèmes, et des bibliothèques comme iRRAM permettent de limiter les calculs à l'obtention d'une précision souhaitée dans le résultat. Cependant la taille variable des nombres manipulés interdit l'utilisation des BLAS. A l'inverse, le calcul exact impose le calcul de la solution dans toute sa précision, mais cette alternative permet l'utilisation de l'efficacité des BLAS. En ce sens il est complémentaire avec

le calcul approché multiprécision et leur utilisation combinée dans des algorithmes hybrides exact/appoché est une perspective prometteuse.

Deuxième partie

Le polynôme caractéristique de matrices denses

INTRODUCTION

Les algorithmes de calcul du polynôme caractéristique sur un corps avec des matrices denses sont nombreux. On trouvera dans les ouvrages [48, 66, 50, 84] une description détaillée des différents algorithmes connus jusque dans les années 1960. Ceux-ci peuvent être regroupés en trois familles distinctes :

Les algorithmes de type Krylov basés sur les itérés d'un vecteur par la matrice. L'algorithme de Danilevskiĭ [23], décrit dans [48], est basé sur des transformations élémentaires de la matrice mais peut être vu comme un algorithme de type Krylov [66, §6.3]. Nous étudierons en détails les algorithmes de Keller-Gehrig [81], tantôt explicitement ou implicitement de Krylov. L'algorithme de Giesbrecht [54] pour le calcul probabiliste de la forme normale de Frobenius est aussi de type Krylov.

Les algorithmes de type Leverrier basés sur le calcul des traces des puissances de la matrice et les formules de Newton [111]. L'algorithme initialement dû à Leverrier a ensuite été redécouvert et amélioré par Souriau [102], Faddeev [22] et Frame [49]. L'algorithme de Csanky [20] l'applique au calcul parallèle.

Les algorithmes de type Samuelson basés sur une formule de développement du déterminant. Il a été initialement proposé par Samuelson [96], puis amélioré par Berkowitz [6].

Les algorithmes de type Leverrier ont une complexité $\mathcal{O}(n^4)$ ou $\mathcal{O}(n^{\omega+1})$ en calcul séquentiel et ne peuvent donc pas rivaliser avec les algorithmes de type Krylov. Il sont en revanche adaptés au calcul parallèle, et l'algorithme de Csanky permet en particulier de prouver que le calcul du polynôme caractéristique est dans la classe $\mathcal{NC}^2$ des problèmes pouvant être résolus en temps parallèle $\log^2 n$ avec un nombre polynomial de processeurs.

Dans leur amélioration de l'algorithme de Csanky, Preparata & Sarwate [94] utilisent la complexité du produit matriciel rapide, mais n'effectuent pas la réduction au produit matriciel. Ce n'est qu'en 1985 que ce problème a été partiellement traité par Keller-Gehrig [81]. La réduction du problème à la complexité $\mathcal{O}(n^\omega)$ n'est donnée que pour les matrices génériques, alors que pour le cas des matrices quelconques, seule la complexité $\mathcal{O}(n^\omega \log n)$ est atteinte. Ces deux algorithmes sont de type Krylov : explicitement pour le deuxième, et de manière implicite pour le premier, comme nous le montrerons dans la partie 8.4.1. Ces complexités n'ont pas été améliorées depuis et le problème reste ouvert de trouver un algorithme général de complexité $\mathcal{O}(n^\omega)$.

Les algorithmes de type Samuelson sont de complexité $\mathcal{O}(n^4)$ ou $\mathcal{O}(n^{\omega+1})$, sans division. Ils sont ainsi essentiellement utiles pour le calcul dans des anneaux commutatifs.

Enfin, les méthodes de type Hessenberg sont basées sur des transformations élémentaires et une formule de développement du déterminant des matrices quasi-triangulaires. Comme l'algorithme de Danilevskiĭ, elles possèdent la meilleure complexité séquentielle, en consi-

dérant l'arithmétique matricielle classique : $2n^3 + \mathcal{O}(n^2)$. En revanche ces formules de développement du déterminant empêchent leur adaptation par blocs et donc ne permettent pas d'envisager la réduction au produit matriciel.

Pour ces raisons, nous étudierons par la suite des algorithmes de type Krylov pour le calcul sur un corps.

Pour le calcul du polynôme caractéristique sur l'anneau des nombres entiers, plusieurs méthodes sont possibles. Certaines utilisent uniquement des opérations d'anneau, avec éventuellement des divisions exactes [1] à partir de la méthode d'Hessenberg, quand d'autres évitent les divisions [6, 73, 78]. Ces méthodes permettent le calcul sur des anneaux quelconques. Une méthode plus spécifique au calcul dans l'anneau $\mathbb{Z}$ est l'utilisation du Théorème des Restes Chinois. Le calcul est effectué sur plusieurs corps finis premiers et le résultat dans $\mathbb{Z}$ peut être reconstitué pourvu que le produit des caractéristiques de ces corps soit supérieur au résultat. Cette technique relie donc l'algorithmique dans l'anneau $\mathbb{Z}$ avec celle dans les corps finis.

Du point de vue des complexités binaires de ces algorithmes, la méthode des Restes Chinois est en $\tilde{\mathcal{O}}(n^{\omega+1}\log\|A\|)$. Les techniques *Pas de Bébé, Pas de Géant* appliquées par Kaltofen dans [73] améliorent cette complexité en $\tilde{\mathcal{O}}(n^{3,5}\log\|A\|)$ (avec l'arithmétique matricielle classique). Plus récemment, les dernières améliorations de [78, §7.2], utilisant les techniques de l'algorithme de Coppersmith par blocs [15, 74, 114, 113] fixent le meilleur exposant connu à ce jour pour ce calcul : $2,697263$ en utilisant l'arithmétique matricielle rapide.

Afin de construire des algorithmes de calcul du polynôme caractéristique dans $\mathbb{Z}$ nous nous intéressons donc d'abord au calcul dans des corps finis. Nous présentons dans le chapitre 6 un premier algorithme de type Krylov, `LU-Krylov`, tirant parti des opérations par blocs des chapitres 2 et 3 mais dont la complexité asymptotique n'est que $\mathcal{O}(n^3)$. Dans le chapitre 7, nous étudions l'algorithme par branchements de Keller-Gehrig, en en donnant une application en théorie du contrôle. Le chapitre 8 traite de l'algorithme rapide de Keller-Gehrig. Enfin, nous présentons dans le chapitre 9 comment appliquer ces algorithmes au calcul dans $\mathbb{Z}$.

Ces travaux ont été initiés lors d'une collaboration avec Zhendong Wan [93, 92] puis poursuivis dans l'article [36], dont la partie présente reprend et complète les résultats.

6

L'ALGORITHME LU–KRYLOV

Les méthodes de Krylov tirent l'information de la matrice à étudier en l'appliquant successivement plusieurs fois à un même vecteur v. Les vecteurs $A^i v$ ainsi produits sont appelés *itérés de Krylov* du vecteur v. La plus petite relation de dépendance linéaire entre ces vecteurs est le polynôme minimal du vecteur v relativement à la matrice A. Nous le noterons $P^{A,v}_{\text{min}}$. C'est le polynôme unitaire P, de plus petit degré, tel que $P(A)v = 0$. Il divise le polynôme minimal P^{A}_{min} de la matrice A, car ce dernier annule aussi cette séquence.

Les méthodes de Krylov reposent sur le résultat suivant, liant les calculs d'itérés d'un vecteur à celui de son polynôme minimal :

Propriété 6.1. *Soit* $V = \begin{bmatrix} v & Av & A^2v & \dots & A^{d-1}v \end{bmatrix}$, *et* $C_{P^{A,v}_{min}} = \begin{bmatrix} 0 & & \dots & -m_0 \\ 1 & 0 & & -m_1 \\ & \ddots & \ddots & \vdots \\ & & 1 & -m_{d-1} \end{bmatrix}$ *la matrice compagnon associée au polynôme* $P^{A,v}_{min} = X^d + \sum_{i=0}^{d-1} m_i X^i$. *Alors la relation suivante est vérifiée :*

$$AV = VC_{P^{A,v}_{min}}. \tag{6.1}$$

Si le vecteur v engendre l'espace total ($d = n$), alors la matrice V est carrée et inversible. On lit alors les coefficients du polynôme caractéristique $P^{A}_{\text{car}} = P^{A,v}_{\text{min}}$ sur la matrice compagnon $V^{-1}AV = C_{P^{A,v}_{\text{min}}}$.

Dans le cas où $d < n$, la méthode de Krylov consiste à compléter la matrice V en une matrice $\overline{V}$ en lui ajoutant $n - d$ vecteurs colonnes, pour former une base de l'espace total. Cette matrice vérifie alors :

$$\overline{V}^{-1} A \overline{V} = \begin{bmatrix} C_{P^{A,v}_{\text{min}}} & X \\ 0 & Y \end{bmatrix}. \tag{6.2}$$

Le polynôme caractéristique de A s'obtient donc comme le produit de $P^{A,v}_{\text{min}}$ avec le polynôme caractéristique de Y, qui pourra être calculé récursivement.

On se reportera à [66, 50] pour une description plus détaillée des méthodes de Krylov.

Nous présentons dans la suite des techniques basées sur la décomposition LUP permettant une mise en pratique efficace de la méthode de Krylov. En particulier, nous montrons qu'une seule décomposition LUP de V, permet à la fois de calculer les coefficients de $P_{\text{min}}^{A,v}$ (sans calculer explicitement le produit $V^{-1}AV$) et de compléter V en $\overline{V}$.

6.1 Le polynôme minimal par élimination sur la matrice de Krylov

Nous présentons ici un algorithme qui détermine le polynôme minimal d'un vecteur v relativement à une matrice A. Un algorithme similaire est présenté dans [84, Algorithme 2.14], mais ne bénéficie pas des opérations par blocs pour l'élimination.

Le principe est de calculer la matrice $K_{A,v}$ de dimension $n \times n$ (appelée matrice de Krylov), dont la $i$ème colonne est formée par le vecteur $A^{i-1}v$, puis d'effectuer une élimination sur cette matrice. Plus précisément, on calcule la décomposition LQUP (voir partie 3.2 et [71]) de la matrice ${}^tK_{A,v}$. Les d premières colonnes de $K_{A,v}$ (d est le degré de $P_{\text{min}}^{A,v}$) sont linéairement indépendantes et les $n-d$ suivantes sont des combinaisons linéaires des d premières. Pour cette raison, la matrice Q de la décomposition LQUP est la matrice identité, et cette décomposition correspond à la décomposition LUP des d premières lignes de ${}^tK_{A,v}$. Elle peut être représentée comme sur la figure 6.1.

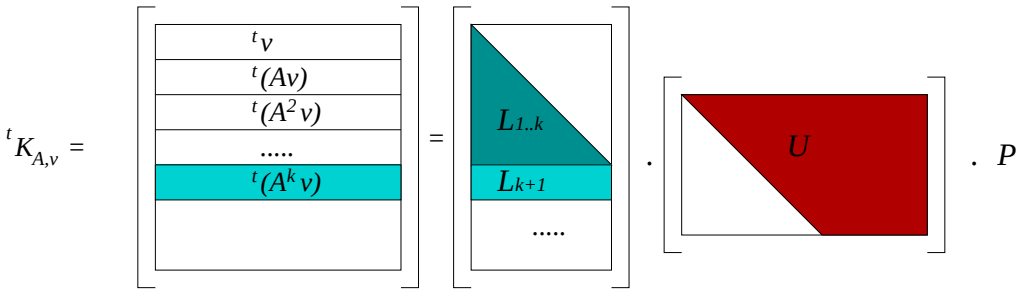


FIG. 6.1 – *Décomposition* LQUP *de la matrice de Krylov* $K_{A,v}$

A partir de cette décomposition, on obtient les coefficients du polynôme $P_{\text{min}}^{A,v}$ par une simple résolution de système linéaire :

Lemme 6.1. *Avec les notations précédentes, soit* $m = (m_0, \ldots, m_{d-1})$ *le vecteur formé par les* d *coefficients de* $P_{min}^{A,v}$. *Alors*

$$ {}^tm = -L_{d+1}L_{1\ldots d}^{-1}. $$

Démonstration. Notons $K = {}^tK_{A,v}$. On a alors

$$\begin{aligned} K_{d+1} &= {}^t(A^d v) \\ &= \sum_{i=0}^{d-1} -m_i \, {}^t(A^i v) \\ &= -m \cdot K_{1\ldots d}. \end{aligned}$$

Ainsi $L_{d+1}UP = -m \cdot L_{1\ldots d}UP$ et donc $m = -L_{d+1}.L_{1\ldots d}^{-1}$. □

On en déduit directement l'algorithme 6.1.

Algorithme 6.1 : `MinPoly` : Polynôme minimal d'un vecteur relativement à une matrice

Données : A une matrice $n \times n$ et v un vecteur de dimension n
Résultat : $P_{\text{min}}^{A,v}(X)$, le polynôme minimal de v relativement à A
début
  $K_{1\ldots n,1} = v$
  pour $i = 1 \ldots \log_2(n)$ **faire**
    $K_{1\ldots n,2^{i-1}+1\ldots 2^i} = A^{2^{i-1}} K_{1\ldots n,1\ldots 2^{i-1}}$
  $(L, Q, U, P) = \texttt{LQUP}({}^tK), d = \text{rank}(K)$
  $m = -L_{d+1}.L_{1\ldots d}^{-1}$
  retourner $P_{\text{min}}^{A,v}(X) = X^d + \sum_{i=0}^{d-1} m_i X^i$
fin

L'opération dominante dans cet algorithme est le calcul des puissances 2^i de la matrice A, soit $\log_2 n$ produits matriciels. La complexité est donc $\mathcal{O}(n^\omega \log n)$ opérations algébriques. Cependant, on peut aussi obtenir la matrice K en calculant les n itérés successivement, avec n produits matrice-vecteur de la forme : $K_{1\ldots n,j+1} = AK_{1\ldots n,j}$. La complexité devient alors $\mathcal{O}(n^3)$. Pour $\omega = 3$, cette dernière approche est donc préférable, car elle ne fait pas intervenir de facteur logarithmique.

Par ailleurs, on peut fusionner les opérations de construction des itérés et leur élimination `LQUP` dans un schéma de type maître-esclave : l'algorithme `LQUP` s'effectue sur les lignes de ${}^tK$ disponibles, et lance le calcul des lignes manquantes, lorsqu'une opération implique un bloc non complet. Ceci permet d'économiser le calcul d'itérés linéairement dépendants. La structure en bloc de l'algorithme `LQUP` impose de calculer ces itérés en blocs et ne permet donc pas d'arrêter le calcul au premier itéré linéairement dépendant. En revanche, si d est le degré de $P_{\text{min}}^{A,v}$, on peut certifier qu'au plus $2d$ itérés seront calculés. Ainsi les complexités de l'algorithme peuvent s'exprimer en fonction du degré du polynôme minimal. Avec la multiplication matricielle rapide : $\mathcal{O}(n^\omega \log d)$ en multiplication matricielle rapide, ou $\mathcal{O}(n^2 d)$ en multiplication classique.

Enfin, dans le contexte d'un calcul dans un corps fini suffisamment grand, cet algorithme permet de calculer de façon probabiliste le polynôme minimal de la matrice A. Si les coefficients de v sont tirés aléatoirement et uniformément dans un sous-ensemble S du corps, la probabilité de réussite est minorée par $1 - n/|S|$ (en appliquant l'analyse de Wiedemann [119]).

6.2 Calcul d'un sous-espace supplémentaire par complément de Schur

Une fois calculée une base du sous espace invariant engendrée par le vecteur v, il faut la compléter en une base de l'espace total pour obtenir la forme triangulaire supérieure par

blocs de l'équation (6.2). Cette opération est effectuée par l'algorithme 4.2. On notera, que la décomposition LQUP dans cet algorithme a déjà été effectuée pour le calcul de $P_{\text{min}}^{A,v}$.

On peut donc maintenant écrire l'algorithme 6.2.

Algorithme 6.2 : LU-Krylov : polynôme caractéristique par la décomposition LUP de la matrice de Krylov

---

Données : A une matrice $n \times n$ sur un corps K
Résultat : P_{car}^{A}, le polynôme caractéristique de A
début
  Tirer un vecteur v aléatoirement et uniformément dans K^n
  $P_{\text{min}}^{A,v} = \text{MinPoly}(A, v)$ de degré d ; /* la décomposition ${}^tV = \begin{bmatrix} L_1 \end{bmatrix} \begin{bmatrix} U_1 & U_2 \end{bmatrix} P$ pour $V = \begin{bmatrix} v & Av & \dots & A^{d-1}v \end{bmatrix}$ est calculée */
  si $(d = n)$ **alors**
    retourner $P_{car}^{A} = P_{min}^{A,v}$
  sinon
    $A' = P\,{}^tA\,{}^tP = \begin{bmatrix} A'_{11} & A'_{12} \\ A'_{21} & A'_{22} \end{bmatrix}$ où A'_{11} is $d \times d$
    $P_{\text{car}}^{A'_{22} - A'_{21}S_1^{-1}S_2}(V) = \texttt{LU-Krylov}(A'_{22} - A'_{21}S_1^{-1}S_2)$
    retourner $P_{car}^{A} = P_{min}^{A,v} \times P_{car}^{A'_{22} - A'_{21}S_1^{-1}S_2}$
fin

Theorème 6.1. *L'algorithme* LU-Krylov *calcule le polynôme caractéristique d'une matrice A de dimensions $n \times n$ en $\mathcal{O}(n^3)$ opérations algébriques.*

Démonstration. Soit $\overline{V}$ la complétion de V obtenue dans l'équation 4.1 et on pose

$$Y = {}^t\overline{V} = \overline{L}\,\overline{U}P.$$

On applique alors la transformation de similitude avec la matrice Y sur ${}^tA$:

$$\begin{aligned} Y\,{}^tAY^{-1} &= \left[\begin{array}{c|c} {}^tC_{P_{\text{min}}^{A,v}} & 0 \\ \hline \begin{bmatrix} 0 & I_{n-d} \end{bmatrix} P\,{}^tA\,{}^tP\overline{U}^{-1}\overline{L}^{-1} & \end{array}\right] \\ &= \left[\begin{array}{c|c} {}^tC_{P_{\text{min}}^{A,v}} & 0 \\ \hline \begin{bmatrix} A'_{21} & A'_{22} \end{bmatrix} \overline{U}^{-1}\overline{L}^{-1} & \end{array}\right] \\ &= \begin{bmatrix} {}^tC_{P_{\text{min}}^{A,v}} & 0 \\ Z & R \end{bmatrix} \end{aligned}$$

où $R = A'_{22} - A'_{21}U_1^{-1}U_2$.

Le polynôme caractéristique étant invariant par similitude, P^A_{car} est égal au produit des polynômes caractéristiques des deux blocs diagonaux de cette matrice. Ainsi,

$$P^A_{\text{car}} = P^{A,v}_{\text{min}} \times P^{A'_{22}-A'_{21}U_1^{-1}U_2}_{\text{car}}.$$

Soit $T_{\texttt{LU-Krylov}}(n)$ le terme dominant de la complexité algébrique de cet algorithme et $T_{\text{minpoly}}(n,d)$ celui de l'algorithme 6.1 appliqué à une matrice de dimensions $n \times n$ dont le polynôme minimal est de dégré d.

En utilisant le produit matriciel classique, on a :

$$\begin{aligned} T_{\texttt{LUK}}(n) &= T_{\texttt{minpoly}}(n,d) + T_{\texttt{LQUP}}(d,n) + T_{\texttt{trsm}}(n-d, d + T_{\texttt{MM}}(n-d,d,n-d) \\ &\quad + T_{\texttt{LUK}}(n-d) \\ &= \mathcal{O}\left(n^2 d + d^2 n + d^2(n-d) + d(n-d)^2\right) + T_{\texttt{LUK}}(n-d) \\ &= \mathcal{O}\left(\sum_i n^2 d_i + d_i^2 n\right) = \mathcal{O}\left(n^3\right) \end{aligned}$$

où les $(d_i - 1)$ sont les degrés de polynômes minimaux calculés à chaque appel récursif. Ils vérifient en particulier $\sum_i d_i = n$ et $\sum_i d_i^2 \leq n^2$.

□

Comme nous l'avons vu dans la partie 6.1, l'algorithme calculant $P^{A,v}_{\text{min}}$ donne de façon probabiliste P^A_{min}. Si tel est le cas pour chaque appel récursif, la suite de ces polynômes minimaux correspond aux facteurs invariants de la matrice A. Ainsi cet algorithme calcule de façon probabiliste la forme de Frobenius de A. Dans le cas contraire, l'algorithme reste cependant déterministe car la dimension du bloc R décroît strictement à chaque appel récursif. Dans ce cas les facteurs apparaissent dans un ordre différent de celui de la forme normale de Frobenius. On trouvera dans [104, §9.3] comment obtenir la suite des facteurs invariants à partir d'une décomposition en facteurs quelconque de P^A_{car}.

Bien que cet algorithme soit basé sur des opérations par blocs, sa complexité ne peut pas se réduire à celle du produit matriciel. En effet, en remplaçant $T_{\texttt{minpoly}} = \mathcal{O}\left(n^2 d\right)$ par $\mathcal{O}\left(n^\omega \log d\right)$, les termes $\log d_i$ ne peuvent se sommer en $\log n$. De même pour les termes $d_i^{\omega-2} n^2$ qui ne peuvent se sommer en n^ω pour $\omega < 3$. C'est justement sur ce point que l'algorithme par branchements de Keller-Gehrig propose une réponse en atteignant la meilleure complexité algébrique de $\mathcal{O}\left(n^\omega \log n\right)$ connue pour ce problème.

7

L'ALGORITHME PAR BRANCHEMENTS DE KELLER-GEHRIG

Nous rappelons ici l'algorithme par branchements de Keller-Gehrig, en le présentant dans un contexte plus général : le calcul de la matrice de Krylov compressée d'un ensemble de vecteurs $B = [b_1|\dots|b_m]$ relativement à une matrice A. Ceci permet à la fois une présentation simplifiée de l'algorithme et son application à un problème issu de la théorie du contrôle : le calcul de la forme de Kalman. L'algorithme original de Keller-Gehrig consiste à choisir $B = I_n$ et considérer une transformation de similitude à partir de la matrice de Krylov compressée obtenue. Nous définissons la matrice de Krylov compressée dans la partie 7.1, puis détaillons l'algorithme pour son calcul dans la partie 7.2. Nous présentons certaines améliorations pour sa mise en pratique dans la partie 7.3. Enfin nous étudions comment l'appliquer au calcul de la forme de Kalman dans la partie 7.5.

7.1 La matrice de Krylov compressée

Soit A et B deux matrices de dimensions $n \times n$ et $n \times m$ respectivement. On considère la matrice de Krylov K de dimension $n \times (mn)$, engendrée par les m vecteurs colonnes de la matrice B et leurs itérés par la matrice A :

$$K = \left[\; b_1 \;\middle|\; \dots \;\middle|\; A^{n-1}b_1 \;\middle|\; \dots \;\middle|\; b_m \;\middle|\; \dots \;\middle|\; A^{n-1}b_m \; \right]. \tag{7.1}$$

Soit r le rang de de K. Évidemment $r \geq \text{rank}(B)$. Formons la matrice $\overline{K}$ de dimension $n \times r$ et de rang plein, en prenant les r premières colonnes linéairement indépendantes de la matrice K.

Définition 7.1. $\overline{K}$ est la matrice de Krylov compressée de la matrice B relativement à la matrice A.

Si un vecteur colonne $A^k b_j$ est linéairement dépendant avec les colonnes précédentes dans K, alors tout vecteur de la forme $A^l b_j, l > k$ sera aussi linéairement dépendant. Par conséquent $\overline{K}$ est de la forme suivante :

$$\overline{K} = \left[\; b_1 \;\middle|\; \dots \;\middle|\; A^{d_1-1}b_1 \;\middle|\; \dots \;\middle|\; b_m \;\middle|\; \dots \;\middle|\; A^{d_m-1}b_m \; \right] \tag{7.2}$$

pour des d_i vérifiant $0 \leq d_i \leq n-1$ et $\sum_{i=1}^{m} d_i = r$.

L'ordre dans le choix des vecteur-colonne indépendants (de gauche à droite) peut être interprété en terme d'ordre lexicographique pour la séquence (d_i). En suivant Storjohann [104, Fact 9.4], la matrice de Krylov compressée peut donc aussi être définie comme suit :

Définition 7.2. La matrice de Krylov compressée de la matrice B relativement à la matrice A est la matrice de la forme

$$\left[\; b_1 \;\middle|\; \ldots \;\middle|\; A^{d_1-1}b_1 \;\middle|\; \ldots \;\middle|\; b_m \;\middle|\; \ldots \;\middle|\; A^{d_m-1}b_m \;\right]$$

de rang r, telle que la séquence (d_i) est lexicographiquement maximale.

Nous présentons dans la partie suivante un algorithme calculant cette matrice de Krylov compressée.

7.2 Principe de l'algorithme

Le calcul de la matrice de Krylov compressée se décompose en deux tâches : le calcul d'itérés, et la sélection de vecteurs linéairement indépendants. Cette dernière opération correspond à la mise sous forme réduite en colonne décrite dans la partie 4.1. Pour cette opération, Keller-Gehrig a proposé dans [81, §4] un algorithme de mise sous forme échelonnée. Nous proposons d'utiliser à la place l'algorithme 4.1 (page 84), de description plus simple, et permettant certaines améliorations, comme nous le verrons dans la partie 7.3. Pour le calcul des itérés, il suffit de calculer les $\lceil \log_2(n) \rceil$ puissances de A :

$$A, A^2, \ldots, A^{2^i}, A^{2^{\lceil \log_2(n) \rceil - 1}}.$$

Le schéma suivant permet alors d'obtenir tous les itérés d'un vecteur b_j.

$$\begin{cases} V_0 &= [b_j] \\ V_{i+1} &= [V_i | A^{2^i} V_i] \end{cases} \tag{7.3}$$

où la matrice V_i a 2^i colonnes.

Cependant, on ne peut pas calculer explicitement la matrice K (7.1) pour ensuite en déduire $\overline{K}$ sa forme réduite en colonne. En effet K est $n \times n^2$ et son calcul demande $\mathcal{O}(n^{\omega+1})$ opérations. L'algorithme par branchements de Keller-Gehrig consiste donc à combiner chaque étape du calcul des itérés avec mise sous forme réduite en colonne : une élimination est effectuée après chaque multiplication par A^{2^i} pour éliminer les itérés linéairement dépendants avant l'itération suivante. De plus, si à l'étape i, un vecteur b_j n'a que $k < 2^i$ itérés linéairement indépendants, il n'en aura jamais davantage. Ainsi le schéma (7.3) ne sera appliqué qu'aux blocs d'itérés de taille 2^i.

Nous pouvons maintenant présenter l'algorithme par branchements de Keller-Gehrig dans l'algorithme 7.1.

Theorème 7.1 (Keller-Gehrig)**.** *Soit A et B deux matrices de dimensions $n \times n$ et $n \times m$ respectivement, avec $m = \mathcal{O}(n)$. La matrice de Krylov compressée de B relativement à A peut être calculée en $\mathcal{O}(n^{\omega} log n)$ opérations dans le corps.*

Algorithme 7.1 : `Matrice de Krylov Compressée` [Keller-Gehrig]

Données : A a $n \times n$ matrix over a field, B, a $n \times m$ matrix
Résultat : $(\overline{K}, r)$ comme dans (7.2)
début
  $i = 0$
  $V_0 = B = (V_{0,1}, V_{0,2}, \ldots, V_{0,m})$
  $C = A$
  tant que *(*$\exists k, V_k$ *a* 2^i *colonnes)* **faire**
    pour tout *j* **faire**
      si *(* $V_{i,j}$ *a strictement moins de* 2^i *colonnes)* **alors**
        $W_j = V_{i,j}$
      sinon
        $W_j = [V_{i,j}|CV_{i,j}]$
    $W = [W_1|\ldots|W_n]$
    $V_{i+1} = \text{FRC}(W)$ $(r = rank(W))$; `/* ` $V_{i+1} = [V_{i+1,1}|\ldots|V_{i+1,n}]$ `où` $V_{i+1,j}$ `sont les vecteurs de` W_j `présents dans` V_{i+1} `*/`
    $C = C \times C$
    $i = i + 1$
  retourner (V_i, r)
fin

Démonstration. D'après l'algorithme 7.1 (cf [81]). □

L'algorithme de Keller-Gehrig a été introduit pour le calcul du polynôme caractéristique. La propriété 7.1 montre le lien entre matrice de Krylov compressée et une forme proche de la forme normale de Frobenius, permettant le calcul du polynôme caractéristique.

Propriété 7.1. *Soit* $\overline{K}$ *la matrice de Krylov compressée de la matrice identité relativement à une matrice* A*. Alors la matrice* $\overline{K}^{-1}A\overline{K}$ *est sous forme Hessenberg polycyclique : elle est triangulaire supérieure par blocs, avec des blocs compagnons sur la diagonale et ses blocs supérieurs sont nuls sauf sur leur dernière colonne.*

$$\overline{K}^{-1}A\overline{K} = \begin{bmatrix} \boxed{\begin{matrix} 0 & & & * \\ 1 & 0 & & * \\ & \ddots & \ddots & * \\ & & 1 & * \end{matrix}} & & & * \\ & & & * \\ & & & * \\ & & & * \\ & \ddots & & \\ & & \boxed{\begin{matrix} 0 & & & * \\ 1 & 0 & & * \\ & \ddots & \ddots & * \\ & & 1 & * \end{matrix}} & \end{bmatrix} \tag{7.4}$$

Corollaire 7.2 (Keller-Gehrig)**.** *Le polynôme caractéristique d'une matrice carrée A d'ordre n peut être calculé en $\mathcal{O}(n^\omega log n)$ opérations dans le corps.*

Démonstration. Le polynôme caractéristique de la matrice Hessenberg polycyclique (7.4) est le produit des polynômes associés à chacun des blocs compagnons de sa diagonale. Et comme les déterminants sont invariants par transformation de similitude, il est égal au polynôme caractéristique de A. □

7.3 Obtention de la forme polycyclique

Pour obtenir la forme Hessenberg polycyclique (7.4), Keller-Gehrig suggère de calculer simplement le produit $\overline{K}^{-1}A\overline{K}$ ce qui engendre $4,66n^3$ opérations dans le corps supplémentaires (avec le produit matriciel classique).

Or en utilisant la factorisation `LQUP` pour calculer la forme réduite en colonnes (algorithme 4.1), nous disposons gratuitement de la décomposition `LQUP` de $\overline{K}$, ce qui permet d'économiser $2/3n^3$ pour ce calcul. Mais il est même possible de réduire ce coût à ϕn^2, où ϕ est le nombre de blocs diagonaux dans la forme Hessenberg polycyclique. Il s'agit d'une généralisation de la technique de la partie 6.1 pour le calcul du polynôme minimal $P^{A,v}_{\text{min}}$.

Tout d'abord, considérons le cas où $\phi = 1$ (les n itérés d'un seul vecteur v sont linéairement indépendants). $K = [v|Av|\dots|A^{n-1}v]$ vérifie

$$K^{-1}AK = \begin{bmatrix} 0 & & & m_0 \\ 1 & 0 & & m_1 \\ & \ddots & \ddots & \vdots \\ & & 1 & m_{n-1} \end{bmatrix},$$

où $P^{A,v}_{\text{min}} = X^n - \sum_{i=0}^{n-1} m_i X^i$. On obtient donc la forme Hessenberg polycyclique en calculant le polynôme minimal de A et v avec l'algorithme 6.1.

Nous montrons ensuite le cas de $\phi = 2$ dont la généralisation est évidente. La figure 7.2 représente leur matrice de Krylov compressée et la matrice L de sa décomposition `LQUP`, fournies par l'algorithme par branchements. La première relation de dépendance pour v_1 s'obtient comme précédemment (figure 7.1). Pour le vecteur v_2, la première relation de dépendance

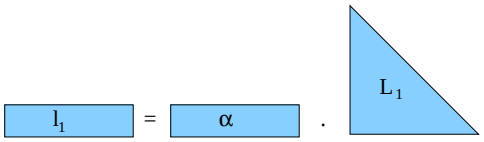


FIG. 7.1 – *Calcul des coefficients de la première relation de dépendance linéaire*

linéaire obtenue pour le vecteur $A^{d_2}v_2$ s'écrit :

$$A^{d_2}v_2 = \sum_{i=0}^{d_2-1} \beta_i A^i v_2 + \sum_{i=0}^{d_1-1} \gamma_i A^i v_1.$$

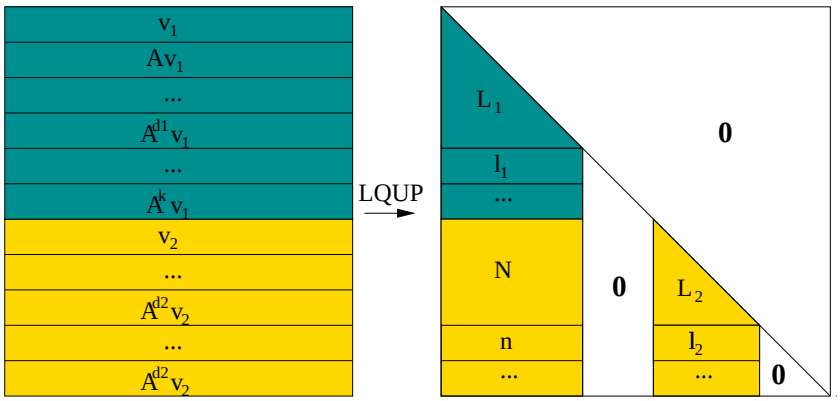


FIG. 7.2 – *Matrice de Krylov compressée à deux itérés*

On obtient alors les vecteurs de coefficients $\beta = [\beta_i]$ et $\gamma = [\gamma_i]$ comme les solutions du système décrit sur la figure 7.3.

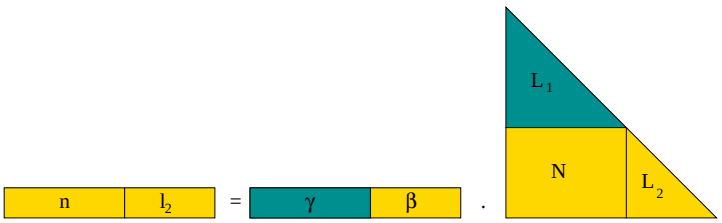


FIG. 7.3 – *Calcul des coefficients de la seconde relation de dépendance linéaire*

Il suffit ensuite de construire la matrice de Hessenberg polycyclique à partir de ces coefficients :

$$H = \begin{bmatrix} 0 & & & \alpha_0 & & & \gamma_0 \\ 1 & 0 & & \alpha_1 & & & \gamma_1 \\ & \ddots & \ddots & \vdots & & & \vdots \\ & & 1 & \alpha_{d_1-1} & & & \gamma_{d_1-1} \\ & & & & 0 & & \beta_0 \\ & & & & 1 & 0 & \beta_1 \\ & & & & & \ddots & \ddots & \vdots \\ & & & & & & 1 & \beta_{d_2-1} \end{bmatrix}.$$

Cette technique s'applique à tous les blocs d'itérés, et par conséquent, la matrice de Hessenberg polycyclique peut être calculée en ϕ résolutions de systèmes linéaires.

Remarque 7.1. Comme ces résolutions utilisent la même matrice triangulaire, on peut envisager de les regrouper en une résolution de système multiple (`trsm`). Dans l'exemple précédent,

il s'agit de calculer

$$\begin{bmatrix} l_1 & 0 \\ n & l_2 \end{bmatrix} \begin{bmatrix} L_1 & \\ N & L_2 \end{bmatrix}^{-1}.$$

Cette méthode a l'inconvénient d'ignorer les zéros du membre de gauche et peut donc augmenter le nombre d'opérations. En revanche si $\phi = \mathcal{O}(n)$, la méthode habituelle nécessiterait $\mathcal{O}(n)$ résolutions de systèmes linéaires, soit $\mathcal{O}(n^3)$ opérations, ce qui dominerait la complexité $\mathcal{O}(n^\omega \log n)$. La résolution simultanée ne nécessite quant à elle que $\mathcal{O}(\phi^{\omega-2} n^2)$ opérations, ce qui reste cohérent avec la complexité globale de l'algorithme.

Noter aussi que pour le calcul du polynôme caractéristique, la forme polycyclique n'est pas nécessaire, mais seuls les coefficients des blocs diagonaux importent. Ainsi, on peut se contenter de ne résoudre que les systèmes $l_1 = \alpha L_1$ et $l_2 = \beta L_2$. Le coût total devient alors $\mathcal{O}(n^2)$ par la méthode classique.

7.4 Comparaisons expérimentales

Nous avons implémenté les deux algorithmes `LU-Krylov` et Keller-Gehrig par branchement, en utilisant les routines présentées dans la partie I et la représentation positive des corps finis de `Modular<double>`. La figure 7.4 compare les temps de calcul des deux implémen-

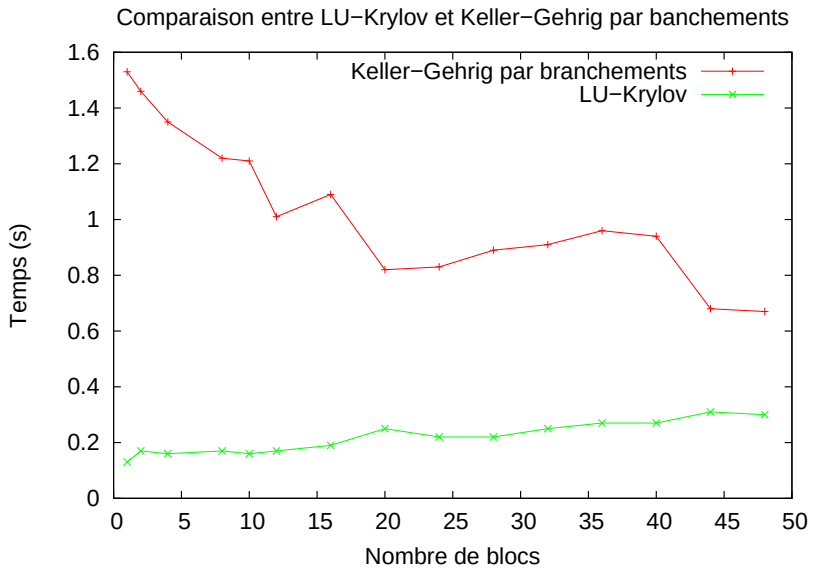


FIG. 7.4 – *Comparaison entre les algorithmes `LU-Krylov` et Keller-Gehrig par branchements sur un Pentium4-2.4Ghz-512Mo*

tations pour une suite de matrices d'ordre 300 dans $\mathbb{Z}_{101}$, dont on fait varier le nombre de blocs dans la forme de Frobenius.

Il ressort que `LU-Krylov` est le plus rapide sur toutes les matrices. Ceci est dû au facteur $\log n$ supplémentaire dans la complexité de l'algorithme par branchements. Ce surcoût n'est

pas ici compensé par le produit matriciel rapide car la dimension 300 est trop faible. Plus précisément, on remarque que le temps de calcul de ce dernier algorithme décroît quand le nombre de blocs augmente. Cela provient du fait que ce facteur $\log n$ est en fait $\log d_{\text{max}}$, où d_{max} est la dimension du plus grand des blocs diagonaux. Or pour ces matrices, cette valeur diminue quand on augmente le nombre de blocs. A l'inverse, le temps de calcul de `LU-Krylov` est presque constant, mais en légère augmentation quand le nombre de blocs croît. Ceci provient du fait que les produits matriciels mis en oeuvre ont des dimensions de plus en plus rectangulaires : ce qui pénalise l'efficacité des BLAS.

7.5 Application au calcul de la forme de Kalman en théorie du contrôle

En théorie du contrôle, la forme de Kalman intervient dans l'étude de problèmes de contrôle de systèmes différentiels linéaires. On étudie des systèmes de la forme

$$\dot{X}(t) = AX(t) + BU(t)$$

où $X(t)$ est un vecteur de dimension n, A est une matrice carrée d'ordre n, B est une matrice de dimension $n \times m$ et $U(t)$ est le vecteur de contrôle de dimension m. L'espace d'état se décompose en un sous-espace contrôlable et un sous-espace libre. Il est alors utile de représenter ce système de façon canonique, en séparant ces deux sous-espaces. C'est ce que permet la forme de Kalman, introduite dans [72].

Theorème 7.3 (Kalman)**.** *Soit A et B deux matrices dans un corps K, de dimensions $n \times n$ et $n \times m$ respectivement. Soit r la dimension de l'espace engendré par les vecteurs colonne de B et leurs itérés par la matrice A. $r = dim(\mathit{Vect}(B, AB, \ldots, A^{n-1}B))$. Il existe une matrice inversible T d'ordre n dans K qui vérifie*

$$T^{-1}AT = \begin{bmatrix} H & X \\ 0 & Y \end{bmatrix}, \begin{bmatrix} B_1 \\ 0 \end{bmatrix} = T^{-1}B$$

où H et B_1 sont respectivement de dimensions $r \times r$ et $r \times m$.

7.5.1 Complexité du calcul de la forme de Kalman

Une première étude dans [38] met en évidence les liens entre les algorithmes de calcul des matrices de Krylov et celui de la forme de Kalman. Suite à une remarque de Gilles Villard [116], nous avons poursuivi cette étude, pour aboutir au résultat suivant :

Theorème 7.4. *Soit V la matrice de Krylov compressée de B relativement à A. Soit T la matrice inversible obtenue en complétant V en une base de K^n par l'ajout de $n - r$ vecteurs colonnes à droite. Alors T vérifie la définition de la forme de Kalman de A et B.*

Démonstration. La matrice V vérifie la relation

$$AV = VH$$

où H est de dimension $r \times r$ et est sous forme Hessenberg polycyclique (7.4). Notons $T = [V|W]$.

Alors

$$AT = \left[\begin{array}{c|c} AV & AW \end{array} \right] = T \begin{bmatrix} H & X \\ 0 & Y \end{bmatrix}.$$

De plus, V est une base de Vect$(B, AB, \ldots, A^n B)$. Ainsi chaque colonne de B est combinaison linéaire de colonnes de V :

$$B = T \begin{bmatrix} B_1 \\ 0 \end{bmatrix}.$$

□

Corollaire 7.5. *La forme de Kalman de A et B peut être calculée en $\mathcal{O}(n^\omega log n)$ opérations dans le corps.*

Démonstration. D'après le théorème 7.1, la matrice V peut être calculée dans la complexité annoncée. On complète celle-ci par l'algorithme 4.2 pour obtenir

$$T = \left[\begin{array}{c|c} \overline{K} & P^T \begin{bmatrix} 0 \\ I_{n-r} \end{bmatrix} \end{array} \right],$$

où ${}^tV = LUP$ est la décomposition `LUP` de ${}^tV$.

Reste ensuite à calculer les blocs H, X, Y et B_1, avec des produits matriciels et des résolutions de systèmes. Toutes ces opérations étant en $\mathcal{O}(n^\omega)$, c'est le calcul de V qui domine la complexité. □

Ce dernier résultat améliore celui donné dans [38, Theorem 2] d'un facteur n.

7.5.2 Mise en pratique

Nous proposons l'algorithme 7.2 pour la mise en oeuvre du calcul de la forme de Kalman, à partir de l'algorithme par branchements de Keller-Gehrig.

Theorème 7.6. *L'algorithme 7.2 est correct et nécessite $\mathcal{O}(n^\omega \log n)$ opérations dans le corps.*

Démonstration. On rappelle (d'après la preuve du théorème 7.4) que

$$AT = \left[\begin{array}{c|c} AV & A\,{}^tP \begin{bmatrix} 0 \\ I_{n-r} \end{bmatrix} \end{array} \right] = T \begin{bmatrix} H & X \\ 0 & Y \end{bmatrix}$$

On rappelle aussi que

$$T = {}^tP \underbrace{\begin{bmatrix} {}^tU_1 & 0 \\ {}^tU_2 & I_{n-r} \end{bmatrix}}_{\overline{U}} \underbrace{\begin{bmatrix} {}^tL & 0 \\ 0 & I_{n-r} \end{bmatrix}}_{\overline{L}},$$

Algorithme 7.2 : `Kalman-KGB` : Kalman-Keller-Gehrig par Branchements

Données : A et B deux matrices dans un corps, de dimension $n \times n$ et $n \times m$
Résultat : r, T, H, X, Y, B_1 comme dans le theorème 7.3

$(V, r) = \texttt{CompressedKrylovMatrix}(A, B)$
si *(r=n)* **alors**
  retourner $(n, Id, A, \emptyset, \emptyset, B)$
sinon
  $(L, [U_1 U_2], P) = \text{LUP}({}^tV)$
  $T = \left[\begin{array}{c|c} V & {}^tP \begin{bmatrix} 0 \\ I_{n-r} \end{bmatrix} \end{array}\right]$
  $B_1 = L^{-T} U_1^{-T} P B$
  $A' = PA\,{}^tP = \begin{bmatrix} A'_{11} & A'_{12} \\ A'_{21} & A'_{22} \end{bmatrix}$
  $X = L^{-T} U_1^{-T} A'_{12}$
  $Y = A'_{22} - {}^tU_2\,{}^tU_1^{-1} A'_{12}$
  pour tout *j* **faire**
    $m_j = l_j L_j^{-1}$ comme expliqué dans la partie 7.3
  Construire la matrice polycyclique H d'après les m_j, comme vu dans la partie 7.3.
  retourner (r, T, H, X, Y, B_1)

Donc X et Y vérifient ${}^t\overline{U}\,{}^t\overline{L}\begin{bmatrix} X \\ Y \end{bmatrix} = PA\,{}^tP\begin{bmatrix} 0 \\ I_{n-r} \end{bmatrix}$.

En posant $A' = PA\,{}^tP = \begin{bmatrix} A'_{11} & A'_{12} \\ A'_{21} & A'_{22} \end{bmatrix}$, on obtient

$$\begin{bmatrix} {}^tU_1 & 0 \\ {}^tU_2 & I_{n-r} \end{bmatrix}\begin{bmatrix} {}^tLX \\ Y \end{bmatrix} = \begin{bmatrix} A'_{12} \\ A'_{22} \end{bmatrix},$$

et le système

$$\begin{cases} {}^tU_1\,{}^tLX & = & A'_{12}, \\ {}^tU_2\,{}^tLX + Y & = & A'_{22}, \end{cases}$$

admet la solution suivante :

$$\begin{cases} X & = & {}^tL^{-1}\,{}^tU_1{}^{-1}A'_{12}, \\ Y & = & A'_{22} - {}^tU_2\,{}^tU_1{}^{-1}A'_{12}, \end{cases}$$

Enfin, pour B_1, on tire de $TB_1 = B$ l'équation :

$$B_1 = {}^tL^{-1}\,{}^tU_1{}^{-1}PB.$$

□

Remarque 7.2. La décomposition `LUP` de ${}^tV$ est déjà effectuée à la sortie de l'appel à `CompressedKrylovMatrix`. L'étape 5 peut donc être supprimée.

Pour le calcul du polynôme caractéristique, l'algorithme `LU-Krylov` est une alternative à l'algorithme par branchements de Keller-Gehrig, et nous avons montré dans la partie 7.4 qu'elle est plus efficace en pratique. De la même façon, nous proposons l'algorithme 7.3, inspiré de l'algorithme `LU-Krylov`, dont la complexité algébrique est $\mathcal{O}(n^3)$ mais ayant une meilleure efficacité pratique (présumée).

Algorithme 7.3 : `Kalman-LUK` : Kalman-LU-Krylov

Données : A et B deux matrices dans un corps, de dimensions $n \times n$ et $n \times m$
Résultat : r, T, H, X, Y, B_1 comme dans le théorème 7.4
début

$v = B_1$

$\begin{cases} K = \begin{bmatrix} v & Av & A^2v & \dots \end{bmatrix} \\ (L, [U_1|U_2], P) = \texttt{LUP}(K^T), r_1 = \text{rank}(K) \end{cases}$; `/* La matrice` K `est calculée au vol : au plus` $2r_1$ `colonnes sont calculées */`

$m = (m_1, \dots, m_{r_1}) = L_{r_1+1} L_{1\dots r_1}^{-1}$

$f = X^{r_1} - \sum_{i=1}^{r_1} m_i X^{i-1}$

si $(r_1 = n)$ **alors**

retourner $(n, Id, A, \emptyset, \emptyset, B)$

sinon

$A' = PAP^T = \begin{bmatrix} A'_{11} & A'_{12} \\ A'_{21} & A'_{22} \end{bmatrix}$ où A'_{11} est $r_1 \times r_1$.

$A_R = A'_{22} - U_2^T U_1^{-T} A'_{12}$

$B' = \begin{bmatrix} L^{-T} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} U_1^{-T} & 0 \\ -U_2^T U_1^{-T} & I \end{bmatrix} PB$

Calculer la permutation Q telle que $B'Q = \begin{bmatrix} * & * \\ 0 & Z \end{bmatrix}$ où Z est $(n - r_1) \times \mu$

si $(\mu = 0)$ **alors**

$X = L^{-T} U_1^{-T} A'_{12}$

$Y = A_R$

$T = \left[\begin{array}{c|c} & \\ K & P^T \begin{bmatrix} 0 \\ I_{n-r_1} \end{bmatrix} \\ & \end{array}\right]$

retourner (r_1, T, C_f, X, Y, X)

sinon

$(r_2, T^{(2)}, H^{(2)}, X^{(2)}, Y^{(2)}, B_1^{(2)}) = \texttt{Kalman-LUK}(A_R, Z)$

$T = \left[\begin{array}{c|c} & \\ K & P^T \begin{bmatrix} 0 \\ T^{(2)} \end{bmatrix} \\ & \end{array}\right]$

$J = L^{-T} U_1^{-T} A'_{12} T^{(2)} = \begin{bmatrix} J_1 & J_2 \end{bmatrix}$; `/*` J_1 `et` J_2 `sont resp.` $r_1 \times r_2$ `et` $r_1 \times (n - r_1 - r_2)$ `*/`

$H = \begin{bmatrix} C_f & J_1 \\ 0 & H^{(2)} \end{bmatrix}, X = \begin{bmatrix} J_2 \\ X^{(2)} \end{bmatrix}$

retourner $(r_1 + r_2, T, H, X, Y^{(2)}, B_1)$

fin

8

L'ALGORITHME RAPIDE DE KELLER-GEHRIG

Le calcul du polynôme caractéristique sur un corps est le dernier problème canonique en algèbre linéaire qui n'a pas été réduit au produit matriciel. En effet la meilleure complexité connue à ce jour est celle de l'algorithme par branchement du chapitre 7 en $\mathcal{O}(n^\omega \log n)$. Cependant Keller-Gehrig décrit dans [81] un algorithme en $\mathcal{O}(n^\omega)$, appelé *algorithme rapide*, pour des matrices génériques.

Nous rappelons son principe dans la partie 8.1, puis nous mettons en évidence, dans la partie 8.2, l'intérêt pratique de cet algorithme par une étude fine de sa complexité et en nous appuyant sur des résultats expérimentaux. Ceci justifie que l'on s'intéresse à sa généralisation, afin de supprimer la contrainte de généricité. Nous avons initié des travaux sur le sujet en proposant dans la partie 8.3 des modifications significatives de l'algorithme, pour relâcher la contrainte de généricité et permettre à l'algorithme de traiter plus de cas. En outre nous fournissons, dans la partie 8.3.5, une preuve de validité avec les matrices génériques pour ce nouvel algorithme.

Une autre piste pour sa généralisation est le recours aux techniques de préconditionnement, développées pour les algorithmes de type Krylov. Dans son article, Keller-Gehrig dit s'être inspiré de l'algorithme de Danilevskiĭ ([23] et sa description dans [66]) pour son algorithme rapide. Il semble en particulier ne pas être basé sur une méthode de Krylov. Nous montrons dans la partie 8.4.1 qu'il s'agit implicitement d'un algorithme de Krylov par blocs, ce qui permet d'envisager l'utilisation de préconditionnements.

8.1 Notations et rappel

On définit les matrices $J_{i,j}$ de dimensions $n \times 2^i$, en plaçant I_{2^i} entre les lignes $(j-1)2^i+1$ et $j2^i$. Il s'agit d'une généralisation "par blocs" des vecteurs de la base canonique. En particulier, multiplier une matrice à droite par $J_{i,j}$ revient à en extraire le $j$ème bloc colonne.

On rappelle qu'une matrice carrée d'ordre n est sous la forme m-Frobenius si elle s'écrit

$$\begin{bmatrix} 0 & * \\ I_{n-m} & * \end{bmatrix}.$$

En particulier, toute matrice d'ordre n est sous forme n-Frobenius et les matrices sous forme 1-Frobenius sont les matrices compagnon.

L'algorithme de Keller-Gehrig consiste à transformer la matrice initiale $A_0 = A$ en une série de matrices A_i sous forme $n/2^i$-Frobenius par des transformations de similitude. Au final, la matrice $A_{\log_2 n} = C_{P^A_{\text{car}}}$ est une matrice compagnon, dont on lit les coefficients du polynôme caractéristique sur la dernière colonne.

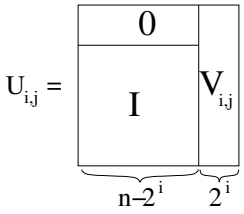
Chacune de ces *grandes étapes* s'écrit $A_i = K_i^{-1} A_{i-1} K_i$ et se décompose en $\frac{n}{2^i} - 1$ *petites étapes* qui sont elles-mêmes des transformations de similitude avec une matrice $U_{i,j}$. On définit ainsi les matrices intermédiaires $A_{i,j}$ telles que :

$$A_{i,0} = A_{i-1}, A_{i,2^i-1} = A_i \text{ et } A_{i,j+1} = U_{i,j}^{-1} A_{i,j} U_{i,j}.$$

On en déduit que

$$K_i = U_{i,1} \dots U_{i,2^i-1}.$$

L'algorithme consiste à choisir les matrices $U_{i,j}$ comme suit :



avec $V_{i,j} = A_{i,j} J_{\frac{n}{2^i}}$ (dernier bloc colonne de $A_{i,j}$). La figure 8.1 illustre le principe d'une *petite étape*.

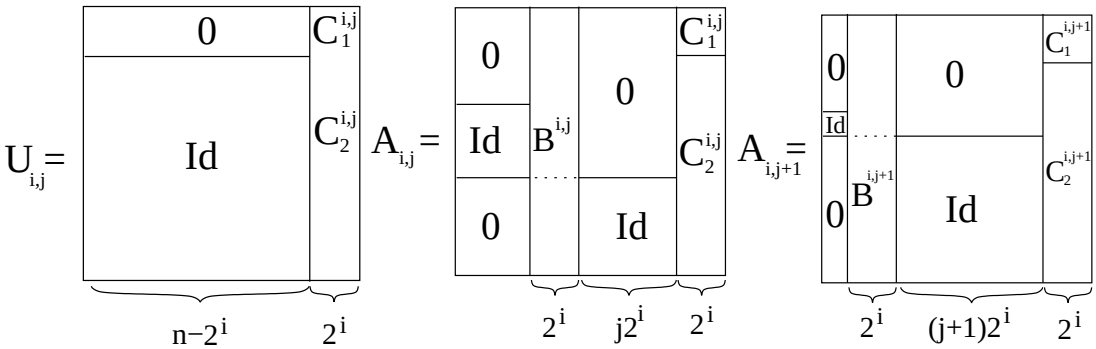


FIG. 8.1 – *Principe d'une petite étape de l'algorithme rapide de Keller-Gehrig*

Cet algorithme est valable si, au cours du calcul, toutes les matrices $U_{i,j}$ sont inversibles, permettant d'effectuer ainsi les transformations de similitude. Dans [81], Keller-Gehrig esquisse une preuve pour montrer que c'est toujours le cas avec des matrices génériques. Néanmoins cette preuve reste très allusive et sujette à caution. Nous donnons dans la partie 8.3.5, une preuve de généricité pour une version modifiée de l'algorithme.

8.2 Intérêt pratique de l'algorithme

8.2.1 Etude de la constante dans la complexité algébrique

L'algorithme rapide de Keller-Gehrig est asymptotiquement le meilleur, mais la notation $\mathcal{O}(n^\omega)$ cache souvent une constante multiplicative grande, rendant l'algorithme inapte à la mise en pratique. Comme dans les chapitres 3 et 4, nous donnons dans le lemme 8.1 le terme dominant de la complexité algébrique de cet algorithme, montrant ainsi qu'il est du même ordre de grandeur que celui de l'algorithme `LU-Krylov`.

Lemme 8.1. *Le calcul du polynôme caractéristique d'une matrice $n \times n$ avec l'algorithme rapide de Keller-Gehrig (s'il aboutit) nécessite $\kappa_\omega n^\omega + o(n^\omega)$ opérations sur le corps, où*

$$\kappa_\omega = C_\omega \left(\frac{2^{\omega-1}(5 \cdot 2^{\omega-3} - 1)}{(2^\omega - 1)(2^{\omega-1} - 1)^2(2^{\omega-2} - 1)} - \frac{2}{2^{\omega-1} - 1} + \frac{2}{2^{\omega-2} - 1} - \frac{1}{2^\omega - 1} - \frac{1}{(2^{\omega-1} - 1)(2^\omega - 1)} \right)$$

Remarque 8.1. En particulier, avec la multiplication matricielle classique, ($\omega = 3, C_\omega = 2$), on obtient $\kappa_\omega = 176/63 \approx 2,794$. A titre de comparaison, l'algorithme `LU-Krylov` appelé sur une matrice générique nécessite $(2 + 2/3)n^3 \approx 2,667n^3$ opérations sur le corps ($2n^3$ pour le calcul de la matrice de Krylov et $2/3n^3$ pour sa décomposition `LUP`). Les deux algorithmes ont donc une complexité algébrique voisine (avec l'arithmétique classique des matrices). Il est donc intéressant de les comparer en pratique.

Démonstration. Nous noterons $X_{a\ldots b}$ la sous-matrice extraite de X en prenant les lignes d'indice compris entre a et b. Comme dans la partie 8.1, on indice par i les *grandes étapes* et j les *petites*. Chaque *petite étape* peut être décrite par les opérations suivantes :

$$\begin{array}{lrcl}
1 & B'_{n-2^i+1\ldots n} & = & C^{-1}_{1\ldots 2^i} B_{1\ldots 2^i} \\
2 & B'_{1\ldots n-2^i} & = & -C_{2^i+1\ldots n} B'_{n-2^i+1\ldots n} + B_{2^i+1\ldots n} \\
3 & C' & = & B' C_{\lambda+1\ldots\lambda+2^i} \\
4 & C'_{2^i+1\ldots 2^i+\lambda} & += & C_{1\ldots\lambda} \\
5 & C'_{2^i+\lambda+1\ldots n} & += & C_{2^i+\lambda+1\ldots n}
\end{array}$$

La première opération est une résolution de systèmes multiples : elle consiste en une décomposition `LUP` et deux résolutions de systèmes triangulaires multiples. Les deux opérations suivantes sont des produits matriciels. Enfin, les deux dernières sont des additions dont le coût est dominé par les opérations précédentes. Le coût d'une transformation de similitude est donc :

$$T_{i,j} = T_{\text{LUP}}(2^i, 2^i) + 2T_{\text{TRSM}}(2^i, 2^i) + T_{\text{MM}}(n - 2^i, 2^i, 2^i) + T_{\text{MM}}(n, 2^i, 2^i).$$

D'après les lemmes 3.1 et 3.2 on obtient

$$\begin{aligned}
T_{i,j} &= \frac{C_\omega 2^{\omega-2}}{2\,(2^{\omega-2} - 1)\,(2^{\omega-1} - 1)}(2^i)^\omega + \frac{C_\omega}{(2^{\omega-2} - 1)}(2^i)^\omega + C_\omega(n - 2^i)(2^i)^{\omega-1} + C_\omega n(2^i)^{\omega-1} \\
&= C_\omega(2^i)^\omega \underbrace{\left(\frac{2^{\omega-3} + 2^{\omega-1} - 1}{(2^{\omega-2} - 1)\,(2^{\omega-1} - 1)} - 1 \right)}_{D_\omega} + 2nC_\omega(2^i)^{\omega-1}.
\end{aligned}$$

Le terme dominant dans la complexité de l'algorithme est donc

$$\begin{aligned} T &= \sum_{i=1}^{\log(n/2)} \sum_{j=1}^{n/2^i-1} T_{i,j} = \sum_{i=1}^{\log(n/2)} \left(\frac{n}{2^i} - 1\right) \left(C_\omega D_\omega (2^i)^\omega + 2nC_\omega (2^i)^{\omega-1}\right) \\ &= C_\omega \sum_{i=1}^{\log(n/2)} (D_\omega - 2)n(2^i)^{\omega-1} + 2n^2(2^i)^{\omega-2} - D_\omega(2^i)^\omega. \end{aligned}$$

Et comme

$$\sum_{i=1}^{\log(n/2)} (2^i)^x = \frac{n^x - 1}{2^x - 1} = \frac{n^x}{2^x - 1} + o(n^x)$$

on obtient :

$$\kappa_\omega = C_\omega \frac{D_\omega 2^{\omega-1}}{(2^\omega - 1)(2^{\omega-1} - 1)} - \frac{2}{2^{\omega-1} - 1} + \frac{2}{2^{\omega-2} - 1}.$$

Or

$$D_\omega = \frac{5 \cdot 2^{\omega-3} - 1}{(2^{\omega-2} - 1)(2^{\omega-1} - 1)} - 1$$

on a donc

$$\kappa_\omega = C_\omega \left(\frac{2^{\omega-1}(5 \cdot 2^{\omega-3} - 1)}{(2^\omega - 1)(2^{\omega-1} - 1)^2(2^{\omega-2} - 1)} - \frac{2}{2^{\omega-1} - 1} + \frac{2}{2^{\omega-2} - 1} - \frac{1}{2^\omega - 1} - \frac{1}{(2^{\omega-1} - 1)(2^\omega - 1)} \right).$$

□

8.2.2 Comparaison expérimentale

Avec le produit matriciel classique, cet algorithme est proche de l'algorithme `LU-Krylov` en terme de complexité ($2,794n^3$ contre $2,667n^3$). Il est donc intéressant de les comparer en pratique. Nous les avons donc implémentés, avec l'implémentation de corps fini `Modular<double>` et les routines de la bibliothèque `fflas_ffpack` décrites dans les chapitres 2 et 3. Pour simuler des matrices génériques, nous utilisons des matrices denses aléatoires sur le corps fini $\mathbb{Z}_{65521}$.

La figure 8.2 compare les vitesses en Mfops de ces deux implémentations. On constate que l'algorithme `LU-Krylov` est plus rapide pour des matrices de petites tailles, mais l'ordre s'inverse pour des dimensions supérieures à 1500 et la différence tend à augmenter. Ce comportement provient de la façon dont les $\mathcal{O}(n^3)$ opérations sont effectuées. En effet, `LU-Krylov` est basé en partie sur des produits matrice-vecteurs (calcul de la base de Krylov) alors que l'algorithme rapide de Keller-Gehrig est basé sur des produits matriciels. Or les BLAS sont plus efficaces pour les produits matriciels que les produit matrices-vecteur, et ceci, d'autant plus que la dimension est grande. Ceci corrobore le point vue énoncé dans l'introduction du chapitre 2 : les algorithmes reposant sur le produit matriciel, sont à la fois intéressants en théorie (pour la réduction de la complexité en $\mathcal{O}(n^\omega)$) mais aussi en pratique.

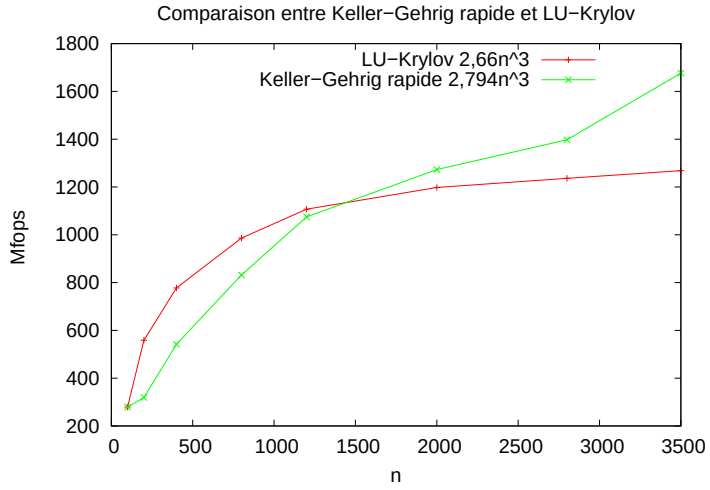


FIG. 8.2 – *Comparaison des vitesses entre l'algorithme rapide de Keller-Gehrig et* `LU−−Krylov` *sur un Pentium4-2,4GHz-512Mo*

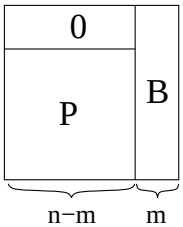
8.3 Vers une généralisation pour les matrices quelconques

Nous proposons ici quelques idées pour élargir le champ d'application de l'algorithme rapide de Keller-Gehrig. La limitation de l'algorithme vient de ce qu'il suppose le bloc $C_1^{i,j}$ de la figure 8.1 inversible. Ce bloc joue un rôle de pivot pour une élimination par blocs (la multiplication à gauche par U^{-1}). Nous montrons comment relâcher cette contrainte en composant un bloc pivot de taille aussi grande que possible, permettant d'effectuer une *petite étape* similaire à celle de la figure 8.1.

Pour ce faire, il faut mettre la colonne C sous un profil de rang générique en appliquant des permutations, d'après la méthode de la partie 4.1.2. Ces permutations modifient la structure de la matrice, c'est pourquoi nous commençons par donner les nouvelles notations dans la partie 8.3.1. La partie 8.3.2 montre comment mettre la colonne C sous profil de rang générique et la partie 8.3.3 indique comment appliquer la transformation de similitude pour effectuer la *petite étape*. Nous décrivons enfin, dans la partie 8.3.4, les limitations de cette méthode.

8.3.1 Généralisation des notations

Nous appelons forme m-Frobenius généralisée, une matrice sous la forme



où P est une matrice de permutation. Chaque *grande étape* consiste à transformer une forme 2^{i+1}-Frobenius généralisée en une forme 2^i-Frobenius généralisée. Avant et après chaque *petite étape*, la matrice est sous la forme indiquée dans la figure 8.3. On retrouve la forme de

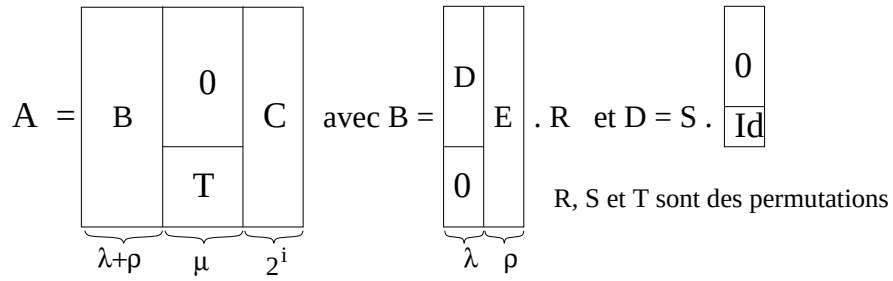


FIG. 8.3 – *Structure de la matrice à chaque itération*

l'algorithme initial (figure 8.1) en prenant toutes les permutations R, S et T égales à l'identité.

On définit les lignes *permises*, comme étant les lignes d'indice inférieur à $\lambda + \rho$ pour lesquelles D est nul. $I_0 = \{i \in [1 \dots \lambda + \rho], D_i = [0 \dots 0]\}$.

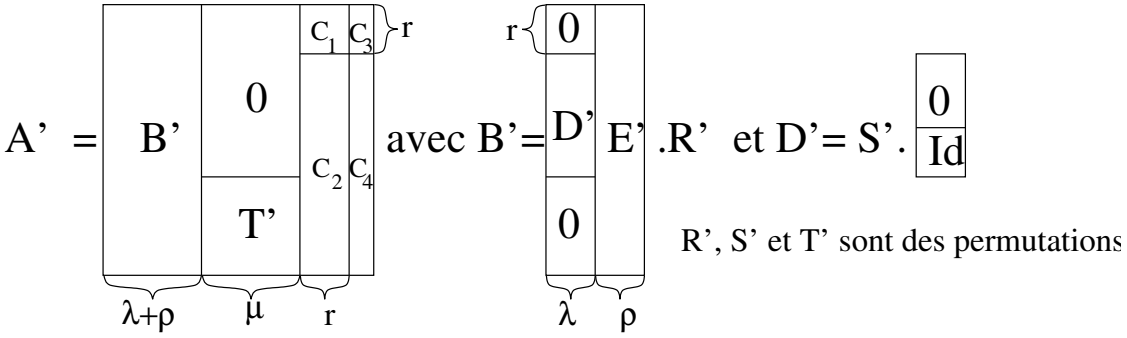
8.3.2 Mise sous profil de rang générique de C

Nous nous plaçons ici dans le cas $\mu \geq 2^i$. Le traitement du cas opposé sera précisé dans la partie 8.3.4.

Pour donner au bloc C un profil de rang générique, nous appliquons la méthode de la partie 4.1.2. Pour éviter le remplissage lors de la transformation de similitude, cette méthode doit être appliquée uniquement aux lignes permises de C. On définit donc $\overline{C}$ de dimensions $(\lambda+\rho) \times 2^i$ tel que

$$\begin{cases} \overline{C}_i = C_i & \forall i \in I_0 \\ \overline{C}_i = [0 \dots 0] & \text{sinon.} \end{cases}$$

Lemme 8.2. *Soit $\overline{C} = LQUP$ la décomposition* `LQUP` *de $\overline{C}$. Soit $\overline{P} = \begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix}$ et $\overline{Q} = \begin{bmatrix} Q & 0 \\ 0 & I \end{bmatrix}$ de dimensions $n \times n$. Alors la matrice $A' = ({}^t\overline{P}\,\overline{Q})^{-1} A\, {}^t\overline{P}\,\overline{Q}$ est sous la forme*



où C_1 est inversible.

Démonstration. P et Q sont respectivement de dimensions $2^i \times 2^i$ et $(\lambda+\rho)\times 2^i$. Or $2^i+\lambda+\rho < n$ donc les matrices $\overline{P}$ et $\overline{Q}$ commutent. Ainsi

$$({}^t\overline{P}\,\overline{Q})^{-1} A\, {}^t\overline{P}\,\overline{Q} = \overline{P}\ {}^t\overline{Q} A\, {}^t\overline{P}\,\overline{Q}$$

et d'après le lemme 4.2, C_1 est bien inversible.

On vérifie ensuite que sur le reste de la matrice, ces permutations ont les effets suivants :

multiplication à gauche par $\overline{P}$: modifie T (dans l'hypothèse que $\mu \geq 2^i$),

multiplication à gauche par ${}^t\overline{Q}$: modifie S et E,

multiplication à droite par $\overline{Q}$: modifie R,

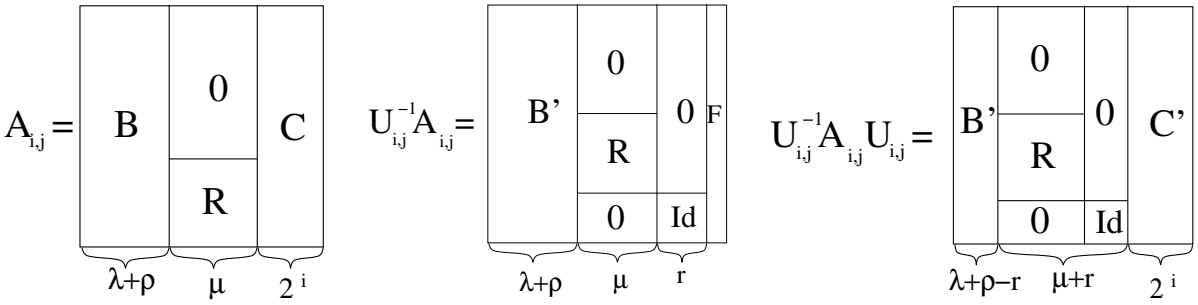
multiplication à gauche par ${}^t\overline{P}$: modifie C,

mais ne modifient pas la structure des blocs. □

8.3.3 Application de la transformation de similitude

Après l'application de ces permutations, nous nous sommes ramenés à une situation proche de celle de l'algorithme initial de Keller-Gehrig : le bloc C_1 est inversible, mais peut être de rang $r < 2^i$. Nous montrons ici qu'on peut toujours construire une transformation de similitude, faisant croître μ de r (au lieu de 2^i).

Il suffit d'effectuer la transformation de similitude avec la matrice $U = \begin{bmatrix} 0 & C_1 \\ I & C_2 \end{bmatrix}$. Les matrices calculées ont la forme suivante.



Une *petite étape* préserve donc la structure de la figure 8.3, en augmentant μ, la taille de la matrice R, de r. Ainsi, tant que la matrice $\overline{C}$ est non nulle, l'algorithme progresse, jusqu'à ce que $\mu = n - 2^i$, marquant comme précédemment la fin d'une *grande étape*.

8.3.4 Compléments et limitations de la généralisation

Rendre la matrice inversible

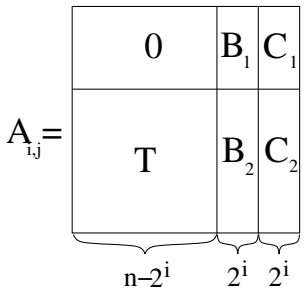
Pour réduire les chances d'avoir $\overline{C}$ nulle, il est préférable de considérer une matrice A inversible. On peut soit imposer cette condition dans les conditions de validité de l'algorithme, soit s'y ramener en utilisant des préconditionneurs.

On peut par exemple utiliser la matrice $\tilde{A} = A + \alpha I$, où α est un élément du corps tiré aléatoirement. La probabilité que $\tilde{A}$ soit singulière est la probabilité que α soit une valeur propre de A, donc inférieure à n/p (où p est la cardinalité du corps fini).

Traitement du cas $\mu < 2^i$

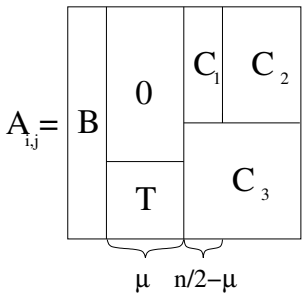
Dans la partie précédente, nous avons laissé de coté le cas où $\mu < 2^i$, car l'application de la permutation $\overline{P}$ à gauche pouvait détruire la structure des blocs. Nous montrons ici, que ce cas n'a jamais lieu en dehors de la première *grande étape* ($i = \log n - 1$) si la matrice est supposée inversible.

Pour $i < \log n - 1$**,** la matrice se présente avant la première *petite étape* sous la forme :



Les lignes *permises* pour former la matrice $\overline{C}$ sont, pour ce cas, les 2^{i+1} premières, donc $\overline{C} = C_1$. Si la matrice est inversible alors, le bloc $\begin{bmatrix} B_1 & C_1 \end{bmatrix}$ est de rang 2^{i+1} et donc forcément, C_1 est de rang 2^i. Ainsi, pour la première *petite étape*, $r = 2^i$ et donc $\mu = 2^i$ avant la deuxième *petite étape*. Il est ensuite croissant pendant le reste de la *grande étape* en cours. Ainsi, la situation $\mu < 2^i$ quand $i < \log n - 1$, n'apparaît que pour $\mu = 0$ et dans ce cas, aucun problème ne se pose.

Pour $i = \log n - 1$**,** si la première *petite étape* génère une matrice T d'ordre $\mu < n/2$, la matrice a la forme suivante :



On ne peut effectuer des permutations avec toutes les colonnes de C, car elles impliqueraient, par transformation de similitude, des permutations entre les $n/2$ dernières lignes de la matrice, ce qui détruirait la structure du bloc T. C'est pourquoi, on doit choisir la matrice $\overline{C} = C_1$ ayant $n/2 - \mu$ colonnes, pour effectuer la mise sous profil de rang générique. Si cette matrice est nulle, l'algorithme est en situation de blocage.

Les situations de blocage

Pour résumer, l'algorithme peut connaître les deux situations de blocage décrites sur les figures 8.4 et 8.5.

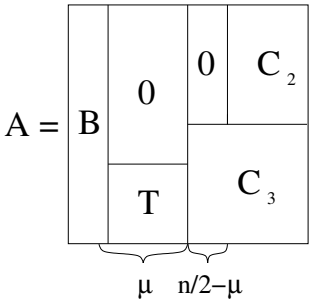


FIG. 8.4 – *Première situation de blocage de l'algorithme généralisé pour* $i = \log n - 1$

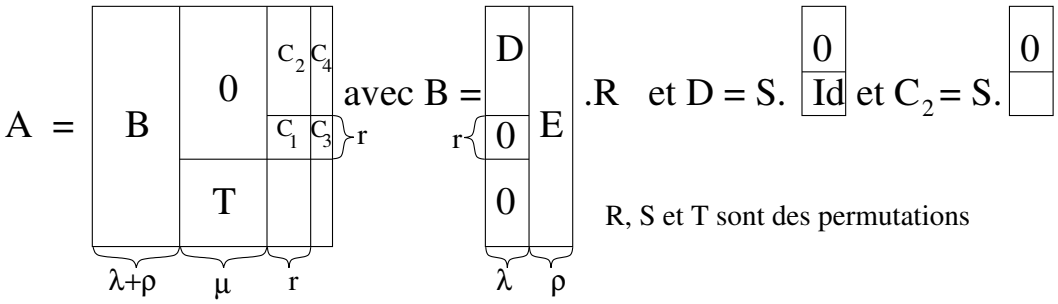


FIG. 8.5 – *Deuxième situation de blocage de l'algorithme généralisé pour* $i < \log n - 1$

Pour la première, $0 < \mu < 2^i$ comme nous l'avons vu précédemment. La deuxième situation de blocage correspond à la situation où l'ensemble des lignes permises est de rang nul. Ainsi, l'algorithme génère la transformation de similitude triviale laissant la matrice inchangée. Dans cette situation de blocage, on peut aussi supposer que $\lambda > 0$ et $\rho > 0$. En effet, si $\lambda = 0$, la matrice est triangulaire inférieure par blocs, et si $\rho = 0$, elle est triangulaire supérieure par blocs. Dans les deux cas, on peut donc calculer récursivement les polynômes caractéristiques des deux blocs diagonaux.

L'alternative : un algorithme en cascade

Pour une mise en pratique de cet algorithme, nous proposons de le combiner avec un autre algorithme de calcul du polynôme caractéristique pour résoudre les cas de blocage. Pour minimiser le nombre d'opérations algébriques, on doit essayer de tirer parti de la structure de la matrice laissée en situation de blocage. C'est ce que permettent les algorithmes basés sur le calcul successif d'itérés par une même matrice, en spécialisant le produit matrice-vecteur. Nous avons donc choisi l'algorithme `LU-Krylov`.

Un produit matrice-vecteur, spécialisé pour les matrices sous la forme de la figure 8.5 coûte $2n(2^i + \rho) + \mathcal{O}(n)$. Dans l'hypothèse où l'algorithme ne calcule qu'un seul bloc d'itérés, on

peut majorer le nombre d'opérations algébriques nécessaires par $2n^2 2^{i+1} + 2/3n^3 + \mathcal{O}(n^2)$. Avec le produit matriciel classique, cet algorithme en cascade a une complexité en $\mathcal{O}(n^3)$, tout comme les algorithmes `LU-Krylov` et Keller-Gehrig rapide. La figure 8.6 représente l'évolution de la constante K dans la complexité Kn^3 de l'algorithme en cascade, selon le niveau

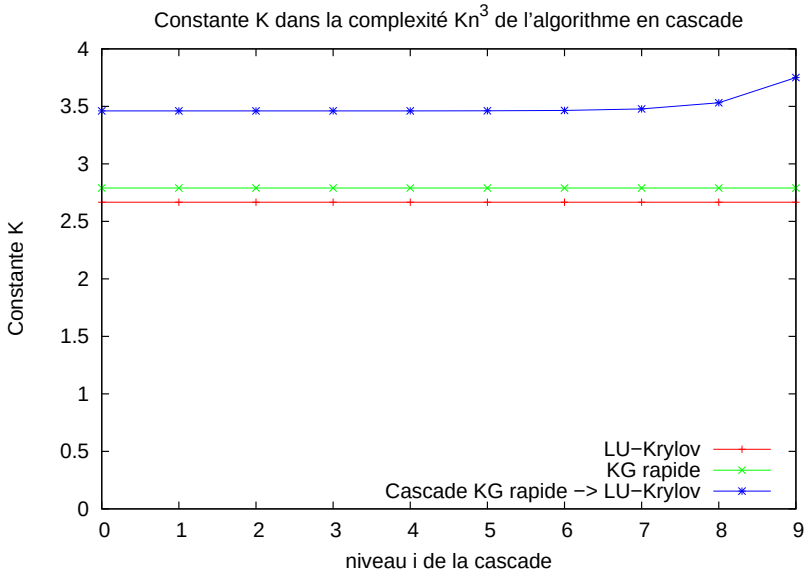


FIG. 8.6 – *Constante dans la complexité de l'algorithme en cascade pour* $n = 2^{10}$

récursif i où la cascade intervient. On rappelle que $i = \log n - 1$ correspond à la première *grande étape* et $i = 0$ correspond à la dernière. On donne à titre de comparaison les constantes des algorithmes `LU-Krylov` $(2,667)$ Keller-Gehrig rapide $(2,794)$. Lorsque la cascade intervient lors des premiers appels récursifs ($i = 9; 8$), le surcoût reste inférieur à $1n^3$. Pour les appels récursifs suivants, la spécialisation du produit matrice-vecteur rend le coût la construction de la matrice de Kylov négligeable. Il reste un surcoût proche de $0,667$ correspondant à la décomposition `LQUP` de la matrice de Krylov, dans l'algorithme `LU-Krylov`.

8.3.5 Preuve de généricité

Dans [81], Keller-Gehrig esquisse une preuve montrant que son algorithme rapide aboutit avec des matrices génériques. Nous proposons ici une démonstration détaillée de ce résultat pour l'algorithme généralisé que nous venons de présenter.

Lemme 8.3. *Soit $n \geq 0$ et K un corps. Il existe un polynôme $\Phi_n \in K[X_{1,1}, \ldots, X_{n,n}]$ à n^2 variables tel que l'algorithme de Keller-Gehrig rapide généralisé peut calculer le polynôme caractéristique d'une matrice $A = [a_{i,j}]$ si et seulement si*

$$\Phi_n(a_{1,1}, \ldots, a_{n,n}) \neq 0$$

Démonstration. Pour que l'algorithme puisse aboutir il faut et il suffit que tous les blocs C_1, apparaissant dans les *petites étapes* (voir partie 8.3.2) soient inversibles. C'est à dire, que le

produit de leur déterminant soit non nul. Chacun de ces blocs étant obtenu par des transformations élémentaires à partir de la matrice A, ces déterminants sont donc des polynômes en les coefficients $a_{i,j}$. □

Pour prouver que l'algorithme aboutit avec des matrices génériques, il suffit donc de prouver que ce polynôme Φ_n est non identiquement nul. Nous allons donc exhiber une famille de matrices $G_n \in K^{2^n \times 2^n}$ et montrer que l'algorithme peut calculer leur polynôme caractéristique. Ainsi on aura montré que $\Phi_{2^n}(G_n) \neq 0$ et donc que $\Phi_{2^n} \neq 0 \; \forall n \geq 0$.

Soit $(G_n)_{n\geq 0}$, la famille de matrices définie par

$$\begin{cases} G_0 & = & [1] \\ G_{n+1} & = & \begin{bmatrix} 0 & G_n \\ I_{2^n} & 0 \end{bmatrix} \end{cases}$$

(G_n est de dimensions $2^n \times 2^n$).

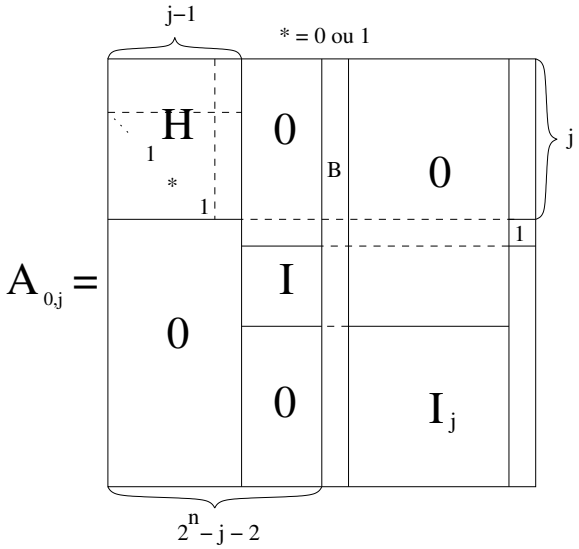
Remarque 8.2. Soit $n \geq 0$.
- G_n est une matrice de permutation.
- Toutes les transformations de similitude appliquées par l'algorithme sur G_n sont aussi des matrices de permutation, et toutes les matrices intermédiaires sont donc des permutations.
- En particulier, il n'y a toujours dans ces matrices qu'un seul élément non nul, égal à 1, par ligne et par colonne.

Pour montrer que l'algorithme ne rencontre aucun cas de blocage dans son exécution, nous allons d'abord montrer cette propriété au niveau de la dernière *grande étape* . Puis nous montrerons comment toute *grande étape* est équivalente à la dernière pour une matrice d'ordre inférieur, ce qui permettra enfin de conclure.

Preuve pour la dernière *grande étape*

Soit $n \geq 1$. On s'intéresse à la dernière *grande étape* de l'algorithme de Keller-Gehrig rapide généralisé, appliqué à la matrice G_n. Nous verrons dans la propriété 8.3 que celle-ci consiste à transformer la matrice $A_{0,0} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ I_{2^n-2} & 0 & 0 \end{bmatrix}$ en la matrice $A_{0,2^n-1} = \begin{bmatrix} 0 & 1 \\ I_{2^n-1} & 0 \end{bmatrix}$.
Comme précédemment, on indexe par j les *petites étapes* intermédiaires, et $A_{0,j}$ est l'état de la matrice après la $j$ième petite étape. Le but de cette preuve est de montrer que l'élément non nul de la dernière colonne de la matrice se trouve toujours sur une ligne permise, ce qui évite ainsi la deuxième situation de blocage (figure 8.5). Pour ce faire nous montrons qu'il se trouve sur la ligne $j+1$ au cours des $2^n/2$ premières *petites étapes*, et qu'il fait toujours partie des $n-j$ premières lignes au cours des $2^n/2$ dernières *petites étapes*.

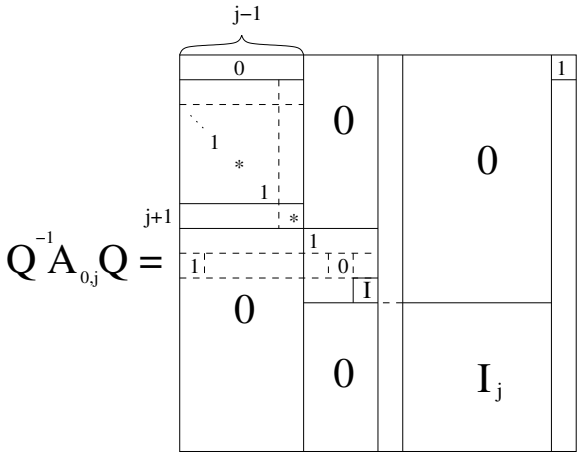
Propriété 8.1. *Si $j < 2^n/2$, alors $A_{0,j}$ s'écrit*



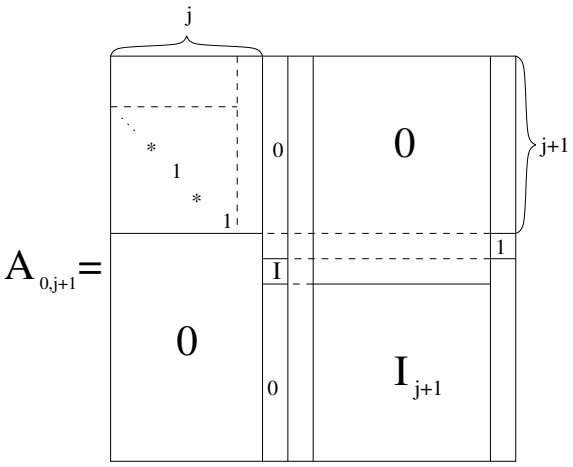
La notation utilisée pour le bloc H de dimension $j \times (j-1)$ signifie que $h_{j,j-2} = h_{j-2,j-4} = h_{j-4,j-6} = \cdots = 1$.

Démonstration. Pour $j = 0$, $A_{0,0}$ vérifie bien cette propriété.

Considérons maintenant le passage de $A_{0,j}$ à $A_{0,j+1}$. La mise sous profil de rang générique de la dernière colonne consiste à permuter les lignes d'indice 1 et $j+1$. Comme c'est une transformation de similitude, il faut aussi permuter les colonnes des même indices. On obtient alors la matrice

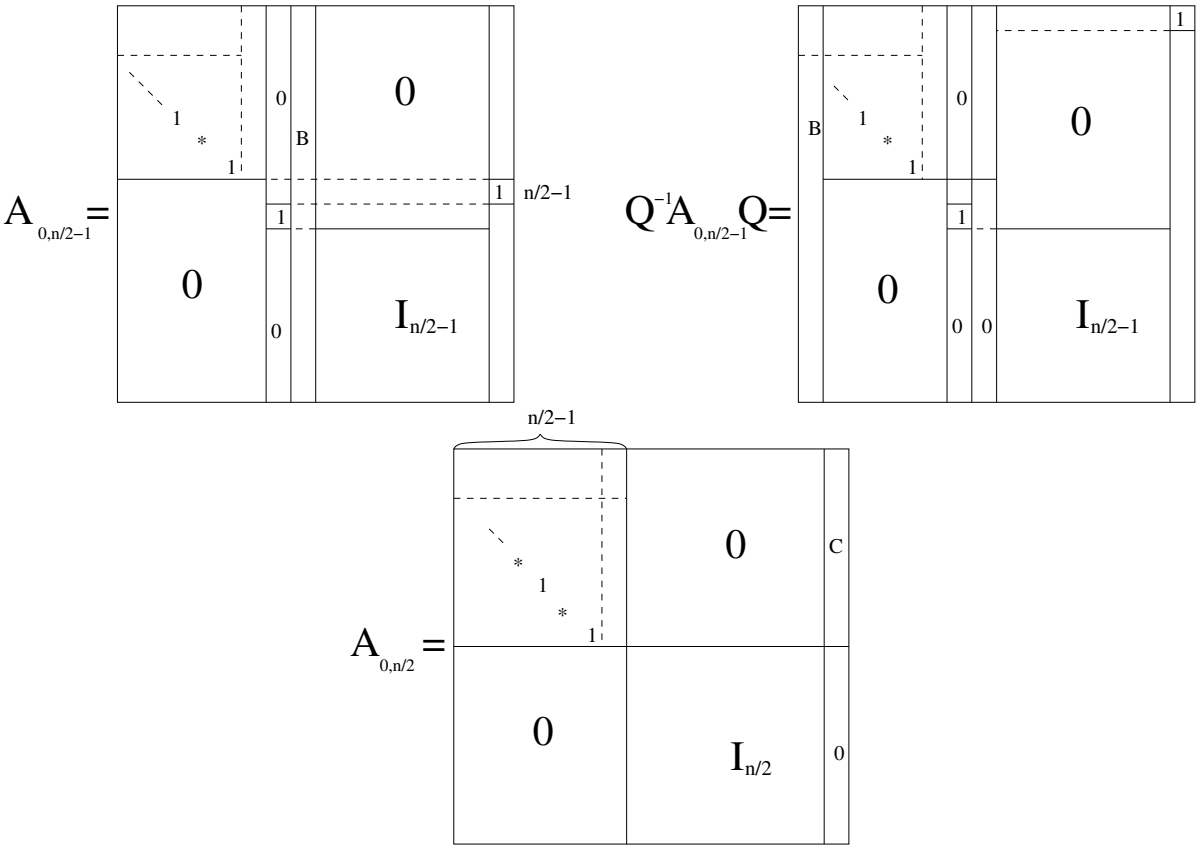


On applique alors la transformation de similitude avec la matrice $U = \begin{bmatrix} 0 & 1 \\ I_{2^n-1} & 0 \end{bmatrix}$ pour obtenir



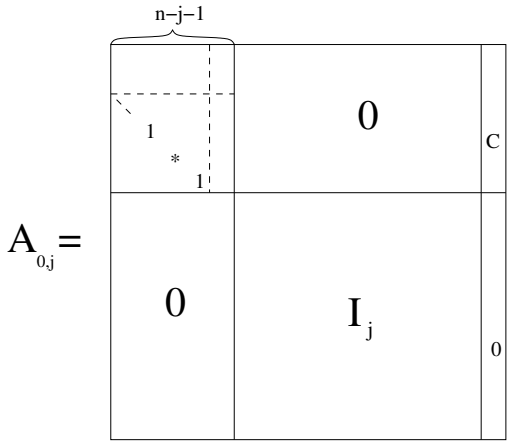
qui est bien de la forme annoncée. □

La *petite étape* intermédiaire entre les états $j = 2^n/2 - 1$ et $j = 2^n/2$ élimine la colonne B :



Pour les $2^n/2$ dernières étapes, la matrice $A_{0,j}$ va garder la structure décrite dans la propriété 8.2. Le coefficient de coordonnées $(n-j, n-j-2)$ valant 1, on est assuré que le coefficient non nul de la colonne C est sur une ligne d'indice strictement inférieur à $n-j$, ce qui empêche l'apparition du cas de blocage 2.

Propriété 8.2. *Pour $j \geq 2^n/2$, la matrice $A_{0,j}$ a la forme suivante*



Démonstration. Soit k l'indice de la ligne où se trouve l'élément non nul du bloc C à permuter en ligne 1. L'étape de mise sous profil de rang générique consiste à permuter les lignes et les colonnes d'indices 1 et k. Comme les lignes d'indice $n-j, n-j-2, \ldots$ comportent déjà des 1, on en déduit que $n-j-k$ est impair. Ainsi, cette étape ne modifiera pas la structure des 1 en diagonale, ni par son action sur les lignes ni sur les colonnes. La transformation de similitude avec la matrice $U = \begin{bmatrix} 0 & 1 \\ I_{2^n-1} & 0 \end{bmatrix}$ effectuera un décalage vers le haut et vers la gauche de ce bloc, en préservant ainsi la forme annoncée pour la matrice $A_{0,j+1}$. □

Nous avons donc montré que l'algorithme effectue la dernière *grande étape* sans rencontrer de situation de blocage. Nous allons maintenant montrer que la structure récursive des matrices G_n permet de généraliser ce résultat aux autres grandes étapes.

Réduction d'une *grande étape* quelconque

Propriété 8.3. *Soit* $2 \leq i < k$ *et* $A = \begin{bmatrix} 0 & G_{i+1} \\ I_{2^k-2^{i+1}} & 0 \end{bmatrix}$.

L'algorithme rapide de Keller-Gehrig généralisé calcule sans blocage la grande étape faisant passer A *d'une forme* 2^{i+1}*-Frobenius à une forme* 2^i*-Frobenius. Le résultat est la matrice*

$$\begin{bmatrix} 0 & G_i \\ I_{2^k-2^i} & 0 \end{bmatrix}.$$

Démonstration. $A \in K^{2^k \times 2^k}$ peut être vue comme une matrice $B \in (K^{2^i \times 2^i})^{2^{k-i} \times 2^{k-i}}$ dont les coefficients sont des matrices d'ordre 2^i. Plus précisément,

$$B = \begin{bmatrix} 0 & 0 & G_i \\ 0 & 1 & 0 \\ I_{2^{k-i}-2} & 0 & 0 \end{bmatrix}.$$

Ainsi, faire passer la matrice A de la forme 2^{i+1}-Frobenius à la forme 2^i-Frobenius revient à faire passer la matrice B de la forme 2-Frobenius à la forme 1-Frobenius. On peut donc

utiliser le résultat de la partie précédente, en remplaçant les coefficients 1 par des blocs I_{2^i} sauf l'élément non nul de la dernière colonne que l'on remplace par G_i. Au final, on obtient bien la forme 2^i-Frobenius

$$\begin{bmatrix} 0 & G_i \\ I_{2^k-2^i} & 0 \end{bmatrix}.$$

□

Nous avons donc montré ici que chaque *grande étape* est équivalente à la dernière *grande étape* du même problème en plus petite dimension. Il ne reste qu'à assembler ces éléments pour conclure.

Preuve de la généricité

Theorème 8.1. *L'algorithme rapide de Keller-Gehrig généralisé calcule le polynôme caractéristique des matrices génériques.*

Démonstration. Il suffit de montrer que le calcul aboutit pour la famille des matrices (G_n). Or d'après la propriété 8.3, chaque grande étape fait passer sans blocage d'une matrice de la forme $\begin{bmatrix} 0 & G_{i+1} \\ I_{2^k-2^{i+1}} & 0 \end{bmatrix}$ à une matrice de la forme $\begin{bmatrix} 0 & G_i \\ I_{2^k-2^i} & 0 \end{bmatrix}$. Les polynômes Φ_{2^k} sont donc non identiquement nuls et on généralise facilement ce résultat aux matrices d'ordre n quelconque. □

8.4 Vers une généralisation par des méthodes de préconditionnement

Une autre piste, pour relâcher les contraintes de généricité de l'algorithme rapide, est de considérer des préconditionnements de la matrice, permettant d'éviter avec grande probabilité les situations de blocage.

Dans la partie 8.4.1, nous montrons que chaque *grande étape* peut être vue comme une transformation de similitude avec une matrice de Krylov par blocs. Ce dernier résultat doit permettre d'envisager d'autres méthodes pour la généralisation de l'algorithme, en particulier en utilisant des techniques de préconditionnements des méthodes de Krylov par blocs, ce que nous évoquerons dans la partie 8.4.2.

8.4.1 Implicitement un algorithme de type Krylov

Propriété 8.4. *Les matrices K_i sont des matrices de Krylov par blocs, engendrées par la matrice A_i à partir du bloc vecteur $J_{\frac{n}{2^i}}$.*

Démonstration. On note $K_i = [K_{i,1}|\dots|K_{i,\frac{n}{2^i}}]$ la décomposition en blocs colonnes de K_i.

On commence par montrer que le premier bloc vecteur est $J_{\frac{n}{2^i}}$. On remarque que la multiplication à droite par $U_{i,j}$ consiste à décaler vers la gauche les $\frac{n}{2^i} - 1$ derniers blocs colonnes. Ainsi, $K_{i,1}$ est la $(\frac{n}{2^i} - 1)$ème colonne de $U_{i,1}$, c'est à dire $J_{\frac{n}{2^i}}$.

Il suffit ensuite de montrer que les blocs colonnes sont obtenus en itérant la matrice A_i, c'est à dire $K_{i,j} = A_i K_{i,j-1}$.

$$\begin{aligned} K_{i,j} = K_i J_j &= U_{i,1} \dots U_{i,\frac{n}{2^i}} J_j \\ &= U_{i,1} \dots U_{i,\frac{n}{2^i}-1} J_{j+1} \\ &= U_{i,1} \dots U_{i,j} J_{\frac{n}{2^i}} \\ &= U_{i,1} \dots U_{i,j-1} V_{i,j}. \end{aligned}$$

Or $V_{i,j} = A_{i,j} J_{\frac{n}{2^i}} = (U_{i,1} \dots U_{i,j-1})^{-1} A_i (U_{i,1} \dots U_{i,j-1}) J_{\frac{n}{2^i}}$, donc

$$\begin{aligned} K_{i,j} &= A_i (U_{i,1} \dots U_{i,j-1}) J_{\frac{n}{2^i}} \\ &= A_i (U_{i,1} \dots U_{i,j-2}) V_{i,j-1} \\ &= A_i K_{i,j-1}. \end{aligned}$$

□

8.4.2 Perspectives

Ainsi, chaque *grande étape* peut être vue comme une transformation de similitude à partir d'une matrice de Krylov par blocs. Si la condition de généricité est mise en défaut par la matrice, c'est que cette matrice de Krylov n'est pas inversible. En suivant les travaux [45], on doit pouvoir borner la probabilité qu'un tel événement se produise, et l'améliorer en préconditionnant la matrice A_i et en utilisant des projections de blocs efficaces.

Par ailleurs, en cas de blocage, la matrice de Krylov par blocs est partiellement calculée. On peut donc aussi envisager d'utiliser des techniques d'extraction de sous-espace supplémentaire, comme pour l'algorithme `LU-Krylov`, afin de poursuivre le calcul.

9

CALCUL DANS L'ANNEAU DES ENTIERS

Il n'est pas question ici de faire une étude exhaustive sur les algorithmes de calcul du polynôme caractéristique entier. Notre but est de montrer comment les algorithmes sur les corps finis, efficaces en pratique, peuvent s'appliquer de façon simple au calcul sur $\mathbb{Z}$ en dépassant les performances des meilleures implémentations actuelles.

Nous donnons un premier algorithme déterministe dans la partie 9.1, reposant simplement sur le Théorème des Restes Chinois. Ces travaux suivent ceux effectués par Giesbrecht [53] puis Giesbrecht et Storjohann [55] pour le calcul de la forme normale de Frobenius dans $\mathbb{Z}$. Notre étude porte principalement sur l'amélioration de la borne sur les coefficients du polynôme caractéristique, afin de réduire le nombre de corps fini à utiliser.

Cet algorithme peut être transformé en un algorithme probabiliste grâce à un mécanisme de terminaison anticipée inspirée de [40, §3.3] que nous décrivons dans la partie 9.2.1. Nous proposons ensuite un autre algorithme dans la partie 9.2.2, basé sur une terminaison anticipée pour le calcul du polynôme minimal et une recherche des multiplicités des facteurs du polynôme caractéristique. Cette dernière méthode, inspirée de [105] aussi développée dans [78, §7.2] ouvre les perspectives d'un calcul sur les matrices *boîte noire*, que nous aborderons dans la partie III. Enfin nous donnons une comparaison en pratique de ces implémentations.

9.1 Un algorithme déterministe par le Théorème des Restes Chinois

Le Théorème des Restes Chinois établit un homéomorphisme entre le produit d'anneaux $\mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_k}$ et l'anneau $\mathbb{Z}_{p_1 \times \cdots \times p_k}$ sous la condition que les p_i soient premiers entre eux.

Ainsi, la méthode classique pour effectuer un calcul sur $\mathbb{Z}$, consiste à déterminer une borne supérieure sur le résultat attendu, choisir suffisamment de nombres p_i premiers entre eux, puis effectuer ce même calcul dans chacun des anneaux $\mathbb{Z}_{p_i}$. Le résultat sur $\mathbb{Z}$ sera ensuite calculé grâce à une formule d'interpolation. On se reportera à [51, §5.4] pour une description plus détaillée de la méthode et son application au calcul du déterminant.

Pour appliquer cette méthode au calcul du polynôme caractéristique sur $\mathbb{Z}$, il faut fournir une borne supérieure sur les coefficients de ce polynôme. La précision de cette borne est importante car elle détermine le nombre de calculs modulaires devant être effectués.

Lemme 9.1. *Soit $A \in \mathbb{Z}^{n\times n}$, avec $n > 4$, dont les coefficients sont bornés en valeur absolue par $B > 1$. Soit (c_j) les coefficients de son polynôme caractéristique. Alors*

$$\log_2(|c_j|) \leq \frac{n}{2}\left(\log_2(n) + \log_2(B^2) + 0.21163175\right)$$

Démonstration. La définition du déterminant (1), appliquée à P^A_{car} donne :

$$\det(XI - A) = \sum_{\sigma\in\mathfrak{S}_n} \epsilon(\sigma)\omega(\sigma)$$

avec

$$\omega(\sigma) = \prod_{j=1}^{n} (\delta_{\sigma(j),j}X - a_{\sigma(j),j}).$$

Le terme de degré k de $\omega(\sigma)$ existe si σ laisse invariant un sous-ensemble I_k de k valeurs de $[1\dots n]$. Il vaut alors $\pm\prod_{j\notin I_k} a_{\sigma(j),j}X^k$.

Ainsi le i-ème coefficient c_i du polynôme caractéristique est une somme alternée de tous les mineurs principaux d'ordre $n-i$ de A. En appliquant la formule de Hadamard [51, Theorème 16.6], pour borner les déterminants d'ordre $n-i$, on obtient

$$|c_i| \leq H(n,i) = \binom{n}{i}\sqrt{(n-i)B^2}^{(n-i)}.$$

Tout d'abord, la symétrie des coefficients binoniaux implique que seuls les $\lfloor n/2\rfloor$ derniers coefficients doivent être considérés, car $\sqrt{(n-i)B^2}^{(n-i)} > \sqrt{iB^2}^{i}$ pour $i < \lfloor n/2\rfloor$.

Pour $i = 0$, le coefficient $c_0 = \det(A)$ vérifie la borne du lemme d'après la borne de Hadamard [51, Theorem 16.6].

Pour $i = 1$, on a

$$\log_2\left(H(n,1)\right) < \frac{n}{2}(\log_2(n) + \log_2(B^2) + 0.21163175)$$

car la différence est négative et décroissante pour $n > 6$.

Nous allons maintenant borner $\log_2(H(n,i))$ pour $i \in [2\dots\lfloor n/2\rfloor]$. Tout d'abord, la formule de Stirling

$$n! = (1+\epsilon(n))\sqrt{2\pi n}\frac{n^n}{e^n}$$

appliquée trois fois permet de borner $\binom{n}{i}$:

$$\forall i \geq 2 \quad \binom{n}{i} < \frac{1+\epsilon(n)}{\sqrt{2\pi}}\sqrt{\frac{n}{i(n-i)}}\left(\frac{n}{i}\right)^i\left(\frac{n}{n-i}\right)^{n-i}.$$

Or d'après [95], $\frac{1}{12n} < \ln(1+\epsilon(n)) < \frac{1}{12n+1}$. Par conséquent pour $n > 4$,

$$\log_2\left(\frac{1+\epsilon(n)}{\sqrt{2\pi}}\right) < \frac{1}{(12n+1)\ln 2} - \log_2\sqrt{2\pi} < -1,2963.$$

De plus $\frac{n}{i(n-i)}$ est décroissant en i pour $i < \lfloor n/2\rfloor$ donc son maximum est $\frac{n}{2(n-2)}$ car $i \in \left[2\dots\left\lfloor\frac{n}{2}\right\rfloor\right]$.

Ainsi on a

$$\log_2 H(n,i) \leq -1,296 + \frac{1}{2}\log_2\left(\frac{n}{2(n-2)}\right) + \underbrace{\log_2\left(\sqrt{(n-i)B^2}^{(n-i)}\left(\frac{n}{i}\right)^i\left(\frac{n}{n-i}\right)^{n-i}\right)}_{K(n,i)}.$$

Or

$$\begin{aligned} K(n,i) &= \frac{n-i}{2}(\log_2(n-i)+\log_2 B^2) + i(\log_2 n - \log_2 i) + (n-i)(\log_2 n - \log_2(n-i)) \\ &= \frac{n}{2}\log_2 n + \frac{n-i}{2}\log_2 B^2 + \frac{n}{2}\left(\log_2\left(\frac{n}{n-i}\right) + \frac{i}{n}\log_2\left(\frac{n-i}{i^2}\right)\right). \end{aligned}$$

On en déduit que

$$\log_2 H(n,i) \leq \frac{n}{2}\left(\log_2 n + \log_2 B^2 + L(n,i)\right)$$

où

$$L(n,i) = \log_2\left(\frac{n}{n-i}\right) + \frac{i}{n}\log_2\left(\frac{n-i}{i^2}\right) - \frac{2}{n}1,296 + \frac{1}{n}\log_2\left(\frac{n}{2(n-2)}\right).$$

Pour un n donné, cette fonction est maximale en $i_n = \frac{-1+\sqrt{1+4en}}{2e}$.

Enfin, la valeur maximale de $T(n,i_n)$ est atteinte en $n = 15$ et vaut approximativement $0,208935$, ce qui est strictement inférieur à 0.21163175. □

Ce résultat rappelle tout d'abord que les coefficients du polynôme caractéristique ont une taille de l'ordre de $\tilde{\mathcal{O}}(n)$ bits. Plus précisément, il montre que cette taille diffère de celle donnée par la borne de Hadamard de $\frac{n}{2}0.21163175$. Ce résultat améliore celui de [55, lemme 2.1] où est donné $2+\log_2(e) \approx 3,4427$ au lieu de 0.21163175.

Par exemple, pour la matrice

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & -1 \\ 1 & -1 & -1 & -1 & 1 \end{bmatrix},$$

le polynôme caractéristique est $X^5-5X^4+40X^2-80X+48$. La borne de Hadamard qui vaut $55,9$ est inférieure à $80 = \binom{5}{1}\sqrt{4}^4$. La borne calculée d'après [55, lemme 2.1] vaut $21792,7$ alors que le lemme 9.1 permet d'obtenir la borne beaucoup plus précise de $80,66661$.

En utilisant l'arithmétique entière rapide et l'algorithme rapide des Restes Chinois [51, Theorem 10.25], on obtient la complexité pour le calcul du polynôme caractéristique dense par les Restes Chinois : $\mathcal{O}\left(n^4(log(n)+log(B))\right)$.

Nous allons dorénavant étudier les variantes probabilistes de ce type d'algorithme, qui permettent d'améliorer considérablement les temps de calcul.

9.2 Algorithmes probabilistes

Nous rappelons dans la partie 9.2.1 comment l'algorithme des Restes Chinois permet d'obtenir des algorithmes probabilistes, grâce à la terminaison anticipée. Nous verrons ensuite dans la partie 9.2.2 un autre algorithme de type *Monte Carlo* réduisant l'utilisation des Restes Chinois au calcul du polynôme minimal.

9.2.1 Terminaison anticipée

Le théorème des Restes Chinois assure qu'un nombre entier peut être reconstruit à partir de ses images modulaires, pour peu que le produit des p_i soit supérieur à ce nombre. Pour le calcul déterministe, nous avons utilisé une borne supérieure sur ce nombre, mais s'il est nettement inférieur à cette borne, de nombreux calculs modulaires seront inutiles. Ne connaissant pas a priori ce résultat, on ne peut le calculer de façon déterministe qu'en effectuant tous les calculs modulaires. En revanche, du point de vue probabiliste, si la valeur reconstituée par le théorème des Restes Chinois reste inchangée après l'ajout d'un calcul modulaire, il y a une forte probabilité que cette valeur soit le résultat attendu. On peut alors arrêter le calcul ou le poursuivre encore sur quelques calculs modulaires, afin d'augmenter cette probabilité.

Plus précisément, nous appliquons le résultat de [40, §3.3] au calcul du polynôme caractéristique dans le lemme 9.2.

Lemme 9.2. *[40] Soit $v \in \mathbb{Z}$ un coefficient du polynôme caractéristique d'une matrice A d'ordre n et U, une borne supérieure sur $|v|$. Soit P un ensemble de nombres premiers et soit $\{p_1 \dots p_k, p^*\}$ un sous-ensemble de P. Soit l tel que $p^* > l$ et $M = \prod_{i=1}^{k} p_i$. On pose $v_k = v \mod M$, $v^* = v \mod p^*$ et $v_k^* = v_k \mod p^*$.*

Si $v_k^* = v^*$, alors $v = v_k$ avec une probabilité supérieure à $1 - \frac{\log_l(\frac{U-v_k}{M})}{|P|}$.

La preuve est celle donnée dans [40, lemma 3.1].

Remarque 9.1. Ce résultat concerne la reconstruction d'un seul coefficient du polynôme caractéristique. En pratique, on teste la stabilisation des n coefficients. Reconstruire n coefficients entiers avec le Théorème des Restes Chinois à chaque nouveau calcul modulaire, peut être coûteux. A la place, on peut calculer une combinaison linéaire de ces coefficients et tester la terminaison anticipée sur cette valeur témoin. Après terminaison, on pourra alors effectuer la reconstruction de l'ensemble des coefficients par l'algorithme des Restes Chinois rapide.

L'algorithme 9.1 résume cette combinaison des Restes Chinois avec la terminaison anticipée.

Ainsi, l'algorithme est devenu sensible à la taille du résultat : pour une même borne a priori sur le résultat, il sera plus rapide si celui-ci est petit. Pour poursuivre dans cette direction, nous allons désormais essayer de le rendre sensible au degré du polynôme minimal.

9.2.2 L'algorithme `HenselPolcar`

La relation de divisibilité entre le polynôme caractéristique et le polynôme minimal d'une matrice apporte une information supplémentaire liant les coefficients du polynôme caractéristique. Or le calcul du polynôme minimal est moins coûteux que celui du polynôme caractéristique, d'autant plus que son degré est faible. Il paraît donc préférable d'utiliser l'algorithme

Algorithme 9.1 : Restes Chinois et terminaison anticipée

Données : A une matrice $n \times n$ sur $\mathbb{Z}$, et $0 < \epsilon < 1$
Résultat : P^A_{car}, le polynôme caractéristique de A avec une probabilité de $1 - \epsilon$
début
  Soit $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{Z}^n$ choisis uniformément dans un intervalle borné (entiers de 20 bits par exemple)
  $i = 1$
  $B = 2^{\frac{n}{2}\left(\log_2(n)+\log_2(||A||^2)+0.21163175\right)}$
  Tirer p, uniformément dans un ensemble de $\frac{1}{\eta}\log_2(\sqrt{n+1}\, 2^{n+1}B + 1)$ nombres premiers
  $(r_{1,1}, \ldots, r_{1,n}) =$ `LU-Krylov`$(A \bmod p_1)$
  $T = t_1 = \sum_{j=1}^{n} \alpha_j r_{1,j} \bmod p_1$
  répéter
    $i = i + 1$
    $T =$ `TRC`$((t_i), (p_i))$; `/* reconstruction par le Théorème des Restes Chinois */`
    Tirer aléatoirement p_i parmi les nombres premiers de 20 bits
    $(r_{i,1}, \ldots, r_{i,n}) =$ `LU-Krylov`$(A \bmod p_i)$
    $t_i = \sum_{j=1}^{n} \alpha_i r_{1,j} \bmod p_1$
    si *($t_i = T$ mod p_i)* **alors**
      Calculer μ, la probabilité du lemme 9.2
      tant que *($\mu < 1 - \epsilon$)* **faire**
        $i = i + 1$
        $T =$ `TRC`$((t_i), (p_i))$; `/* reconstruction par le Théorème des Restes Chinois */`
        Tirer aléatoirement p_i parmi les nombres premiers de 20 bits
        $(r_{i,1}, \ldots, r_{i,n}) =$ `LU-Krylov`$(A \bmod p_i)$
        $t_i = \sum_{j=1}^{n} \alpha_i r_{1,j} \bmod p_1$
        Calculer μ, la probabilité du lemme 9.2
      retourner $P = (r_1, \ldots, r_n)$
  juqu'à *l'infini*
fin

des Restes Chinois avec terminaison anticipée uniquement pour le calcul du polynôme minimal. Les coefficients du polynôme caractéristique pouvant être obtenus à partir de ceux-ci par d'autres moyens.

Ainsi Storjohann propose dans [105] un algorithme qui à partir du polynôme minimal entier et du polynôme caractéristique sur un corps fini, calcule le polynôme caractéristique entier, grâce à des calculs de bases pgcd-libres.

Pour une première mise en pratique, nous en proposons une version modifiée dans l'algorithme 9.2, qui remplace le calcul de la base gcd-libre par une factorisation du polynôme minimal sur $\mathbb{Z}$. En théorie, cette opération est trop coûteuse mais des implémentations efficaces de la remontée de Hensel, comme celle disponible dans la bibliothèque `NTL` [100] la rendent praticable dans beaucoup de cas.

Algorithme 9.2 : `HenselPolcar` : Polynôme caractéristique entier par la factorisation de Hensel.

Données : $A \in \mathbb{Z}^{n\times n}$, $0 < \epsilon < 1$.
Résultat : P^A_{car}, le polynôme caractéristique de A avec une probabilité de $1-\epsilon$ ou `ECHEC`.
début
  $\eta = 1 - \sqrt{1-\epsilon}$
  P^A_{min} = `PolMinTRC`(A, η) ; `/* par [40, §3] ou par l'algorithme 9.1 appliqué au calcul du polynôme minimal */`
  Factoriser $P^A_{\text{min}} = \prod_i f_i$ en facteurs irréductibles sur $\mathbb{Z}$, par la remontée de Hensel
  $B = 2^{\frac{n}{2}\left(\log_2(n)+\log_2(||A||^2)+0.21163175\right)}$
  Tirer p, uniformément dans un ensemble de $\frac{1}{\eta}\log_2(\sqrt{n+1}\,2^{n+1}B+1)$ nombres premiers
  $P_p = $ `LU-Krylov`$(A \mod p)$
  pour tout *f_i facteur irréductible factor of P^A_{min}* **faire**
    $\overline{f}_i \equiv f_i \mod p$
    Déterminer α_i, la multiplicité de $\overline{f}_i$ dans P_p
    si $\alpha_i = 0$ **alors**
      retourner *ECHEC*
  $P^A_{\text{char}} = \prod_i f_i^{\alpha_i} = X^n - \sum_{i=0}^{n-1} a_i X^i$
  si *($\sum \alpha_i degree(f_i) \neq n$)* **alors**
    retourner *ECHEC*
  si *(trace$(A) \neq a_{n-1}$)* **alors**
    retourner *ECHEC*
  retourner P^A_{char}
fin

Theorème 9.1. *L'algorithme 9.2 est correct.*

Démonstration. Soit P^A_{min} le polynôme minimal entier de A et $\tilde{P}^A_{\text{min}}$ le résultat de l'appel à `PolMinTRC`. Avec une probabilité supérieure à $\sqrt{1-\epsilon}$, $P^A_{\text{min}} = \tilde{P}^A_{\text{min}}$. Dans ce cas, le seul problème pouvant arriver est qu'un des facteurs irréductible de P^A_{min} divise un autre facteur modulo p, ou de manière équivalente, que p divise le résultant de ces deux polynômes.

Or d'après [51, Algorithm 6.38] et le lemme 9.1, la taille de ce résultant est bornée par $\log_2(\sqrt{n+1}\,2^{n+1}B+1)$. Par conséquent, la probabilité de choisir un mauvais nombre premier est inférieure à η. Le résultat sera donc correct avec une probabilité supérieure à $1-\epsilon$ □

Cet algorithme est aussi capable de détecter différents résultats faux en retournant `ECHEC`. Dans ce cas, on peut relancer le calcul. Cependant, il ne s'agit pas d'un certificat pour toutes les erreurs possibles et par conséquent il ne s'agit pas d'un algorithme de type *Las-Vegas*.

Dans le cas où l'algorithme probabiliste de calcul du polynôme minimal entier donne un résultat correct, $P^A_{\text{min}} = \tilde{P}^A_{\text{min}}$, la seule erreur pouvant arriver est qu'un facteur de P^1_{min} divise un autre facteur modulo p. Dans ce cas, l'exposant de ce facteur apparaîtra deux fois dans le polynôme caractéristique reconstruit et le degré total sera supérieur à n. L'algorithme retournera alors `ECHEC`.

Dans le cas où le polynôme minimal est incorrect, $P^{\text{min}} \neq \tilde{P}^{\text{min}}$, le test $\alpha_i > 0$ le détectera sauf si $\tilde{P}^{\text{min}}$ est un diviseur de P^{min} modulo p. Dans ce cas, soit Q tel que $P^{\text{min}} = \tilde{P}^A_{\text{min}} Q \mod p$. Si Q ne divise pas $\tilde{P}^A_{\text{min}}$ modulo p, c'est que certains facteurs de P^A_{min} ont été omis dans $\tilde{P}^A_{\text{min}}$ donc le degré total sera inférieur à n et l'algorithme retournera `ECHEC`. Dans le cas contraire, un polynôme caractéristique incorrect, de degré n, sera reconstruit, mais le test de la trace détectera la plupart des cas.

C'est ce dernier point qui empêche d'avoir un certificat de correction et donc un algorithme de type *Las Vegas*.

La complexité binaire de l'algorithme est moins bonne que celles de [105] ou [78] mais les résultats expérimentaux, présentés dans la partie 9.3 montrent son intérêt pratique.

9.2.3 Vers un algorithme pour les matrices *boîte noire*

Cet algorithme est avant tout conçu pour le calcul avec des matrices denses. On y utilise les algorithmes denses 6.1 et 6.2 pour le calcul des polynômes minimal et caractéristique dans des corps finis.

Cependant, le polynôme minimal entier peut être calculé par d'autres algorithmes. En particulier, avec des matrices *boîte noire*, on peut utiliser l'algorithme de Wiedemann, combiné de la même façon avec les Restes Chinois (voir [40, §3]). Le calcul du polynôme caractéristique modulo p reste quant à lui un calcul dense. Il en découle un algorithme tirant parti de la structure des matrices pour son opération dominante (le calcul du polynôme minimal dans $\mathbb{Z}$), mais limité au niveau de la mémoire par l'algorithme dense `LU-Krylov`. Ce premier pas vers les méthodes *boîte noire* s'avère intéressant en pratique, comme nous le montrons dans la partie 9.3. Mais nous nous intéressons plus particulièrement aux algorithmes creux dans la partie III.

9.3 Résultats expérimentaux

Nous comparons ici ces algorithmes en pratique. Nous désignons par `LUK-det` l'algorithme déterministe présenté dans la partie 9.1 basé sur l'algorithme `LU-Krylov`, les Restes Chinois et la borne du lemme 9.1. Nous déclinons l'algorithme probabiliste `HenselPolcar` en une version dense, `HPC-dense` et une version *boîte noire*, `HPC-BN`, comme expliqué dans la partie 9.2.3. Pour la factorisation par la remontée de Hensel, nous avons utilisé la bibliothèque NTL [100].

Le choix des moduli est lié aux contraintes du produit matriciel. En effet, pour appliquer les techniques décrites dans la partie 2.1.1 et utiliser les BLAS, il faut que $n(p-1)^2 < 2^{53}$ (avec une arithmétique matricielle classique). Nous avons donc choisi de tirer les nombres premiers aléatoirement entre 2^m et 2^{m+1} (où $m = \lfloor 25,5 - \frac{1}{2}\log_2(n) \rfloor$). Cet intervalle est toujours suffisant. Même pour les matrices d'ordre 5000, $m = 19$ et il y a $38\,658$ nombres premiers entre 2^{19} and 2^{20}. Et si les coefficients de la matrice sont compris entre -1000 et 1000, la borne supérieure sur les coefficients du polynôme minimal est $\log_{2^m}(U) \approx 4267,3$. Par conséquent la probabilité de trouver un mauvais nombre premier est inférieure à $4267,3/38658 \approx 0,1104$. Puis, en effectuant quelques calculs modulaires supplémentaires pour vérifier que le résultat reconstruit reste invariant, on pourra rapidement réduire cette probabilité. Dans cet exemple, seulement 18 calculs supplémentaires sont nécessaires (à comparer avec les 4268 pour le calcul déterministe) pour assurer une probabilité d'erreur inférieure à 2^{-55}. A ce niveau, il y a par exemple plus de chance qu'un rayonnement cosmique ait perturbé le calcul sur la machine physique.[1] Dans la suite, les temps donnés pour les algorithmes probabilistes correspondent à des probabilités d'erreurs inférieures à 2^{-55}.

n	Maple	Magma	LUK-det	HPC-dense
100	163s	0,34s	0,23s	**0,20s**
200	3355s	4,45s	3,95s	**3,25s**
		11,1Mo	3,5Mo	**3,5Mo**
400	74970s	**69,8s**	91,4s	71,74s
		56Mo	10,1Mo	**10,1Mo**
800		1546s	1409s	**1110s**
		403Mo	36,3Mo	**36,3Mo**
1200		8851s	7565s	**5999s**
		1368Mo	81Mo	**81Mo**
1500		DM	21 010s	**16 705s**
			136Mo	**136Mo**
3000		DM	349 494s	**254 358s**
			521Mo	**521Mo**

DM : Débordement Mémoire

TAB. 9.1 – *Calcul du polynôme caractéristique de matrices entières denses (temps de calcul en s et allocation mémoire en Mo)*

Dans le tableau 9.1 les matrices sont denses, et à coefficients choisis uniformément entre 0 et 10. Leur polynôme minimal égale leur polynôme caractéristique. Nous comparons les com-

[1] En effet, les rayonnements cosmiques sont responsables de 10^5 erreurs de calcul toutes les 10^9 heures sur une puce au niveau de la mer ([90]). Avec une fréquence de 1GHz, cela donne une erreur tous les 2^{55} cycles d'horloge.

portements de `LUK-det`, `HPC-dense` de `Maple-v9` et `Magma-2.11`. Ces calculs ont été effectués sur un Athlon 2200 (1,8 Ghz) avec 2Go de mémoire vive sous Linux-2.4.[2] Dans le logiciel `Maple`, la fonction `LinearAlgebra :-CharacteristicPolynomial` implémente l'algorithme de Berkowitz. Celui-ci est très vite trop coûteux en temps de calcul pour des matrices d'ordre supérieur à 100.

Le logiciel `Magma` utilise un algorithme p-adique combinant un calcul probabiliste et une preuve de correction, rendant le résultat déterministe. Pour ces matrices, cette vérification est négligeable car le polynôme caractéristique égale le polynôme minimal [103]. Concernant les algorithmes déterministes, `LUK-det` est sensiblement plus rapide que `magma` dans la plupart des cas. Pour des matrices d'ordre supérieur à 800, `magma` cherche à allouer plus de 2Go de mémoire vive pour stocker la matrice de Krylov correspondante (à coefficients entiers). Par conséquent le calcul est abandonné, ce que nous désignons par *DM* (pour *Débordement Mémoire*). Les allocations de l'algorithme `LUK-dense` sont nettement inférieures et permettent de manipuler des matrices d'ordre largement supérieur.

Concernant les algorithmes probabilistes, l'algorithme `HPC-dense` améliore les temps de calcul de `LUK-dense` d'environ 20%.

Nous nous intéressons désormais à des matrices ayant une structure plus riche. Les matrices A et B, d'ordre 300 et 600 ont un polynôme minimal de degré respectivement 75 et 424. A est la matrice `Frob08blocks` sous forme normale de Frobenius, avec 8 blocs compagnons. Elle est donc très creuse (moins de 2 éléments non nuls pas ligne). B est la matrice `ch5-5.b3.600x600.sms` présentée dans [40].

Matrice	A	$U^{-1}AU$	${}^tAA$	B	$U^{-1}BU$	${}^tBB$
n	300	300	300	600	600	600
d	75	75	21	424	424	8
ω	1,9	300	2,95	4	600	13
`Magma-prob`	1,14	7,11	**0,23**	6,4	184,7	6,04
`Magma-det`	1,31	10,55	0,24	6,4	185	6,07
`LUK-det`	1,1	93,3	64,87	68,4	2305	155,3
`HPC-BN`	**0,32**	4,32	0,81	**4,4**	352,6	**2,15**
`HPC-dense`	1,22	**1,3**	0,87	38,9	**42,6**	2,57
`PolMinTRC-BN`	0,03	4,03	0,02	1,62	349	0,08
`PolMinTRC-dense`	0,93	1,0	0,08	36,2	39,9	0,50
`Fact`	0,04	0,04	0,01	0,61	0,58	0,01
`LU-Krylov`	0,25	0,25	0,78	2,17	2,08	2,06

TAB. 9.2 – *Calcul sur des matrices creuses ou structurées, temps en s*

[2]Nous remercions le centre de calcul *Médicis*, `http://medicis.polytechnique.fr/medicis`, du laboratoire CNRS STIX pour avoir mis à notre disposition ses machines ainsi que ses logiciels.

Dans le tableau 9.2, d est le degré du polynôme minimal et ω le nombre moyen d'éléments non nuls par ligne.

Nous donnons les temps de calcul de l'algorithme 9.2, décomposé en chacune de ses opérations principales : le calcul du polynôme minimal entier (`PolMinTRC-BN` pour l'algorithme [40, §3] ou `PolMinTRC-dense` for 6.1 combiné avec les Restes Chinois ; la factorisation de ce polynôme sur $\mathbb{Z}$ (`Fact`) et le calcul du polynôme caractéristique modulo p (`LU-Krylov`). Ces temps sont comparés à ceux de l'algorithme déterministe `LUK-det`, ainsi qu'à l'algorithme de `magma`, avec ou sans vérification d'exactitude (respectivement `magma-det` et `magma-prob`). A nouveau, `LUK-det` est plus rapide que `magma`. Pour les deux matrices A et B, `HPC-BN` est le plus rapide, grâce à l'avantage que lui confère l'algorithme de Wiedemann de calcul du polynôme minimal pour les matrices *boîte noire* (`PolMinTRC-BN`).

Nous avons ensuite appliqué une transformation de similitude dense sur ces matrices, afin de conserver leur structure de Frobenius (en particulier, un polynôme minimal de faible degré) mais sous forme d'une matrice dense. La densité des matrices ralentit `magma` et `HPC-BN`, alors que `HPC-dense` maintient des temps similaires et dépasse donc les autres algorithmes.

Nous donnons aussi des temps de calcul pour des matrices symétriques avec un polynôme minimal de faible degré : Il s'agit des matrices (${}^tAA$ et ${}^tBB$). `HPC-BN` est toujours efficace (meilleur pour ${}^tBB$), mais `magma` se révèle particulièrement rapide sur ${}^tAA$.

Matrice	n	ω	`magma-prob`	`HPC-BN`	`HPC-dense`
`TF12`	552	7,6	10,1s	**6,8s**	61,77s
`Tref500`	500	16,9	112s	**65,1s**	372,6s
`mk9b3`	1260	3	48,4s	**31,3s**	433s

TAB. 9.3 – *Avantage de `HPC` sur d'autres matrices creuses*

Enfin, nous donnons dans le tableau 9.3 des comparaisons supplémentaires avec des matrices creuses tirées de diverses applications. `LUK-det` est relativement coûteux, tout en restant légèrement plus rapide que `magma`. L'approche probabiliste de `HPC-dense` ou `HPC-BN` permet d'atteindre les meilleurs temps de calcul dans presque tous les cas.

Conclusions : Ces expérimentations ont mis en évidence l'efficacité de l'approche que nous avons suivie pour l'algorithmique dense. Elle permet dores et déjà d'améliorer les performances des meilleurs logiciels de calcul, grâce à des routines efficaces sur les corps finis. Mais l'algorithme `HPC` peut être encore amélioré, par exemple en remplaçant la factorisation avec la remontée de Hensel par des manipulations de bases pgcd-libres, en suivant [105]. Cet algorithme nous a par ailleurs permis d'entrevoir la problématique de l'algorithmique pour les matrices *boîte noire*, ce que nous allons développer dans la partie III.

Troisième partie

Le polynôme caractéristique de matrices *boîte noire*

INTRODUCTION

L'opération de multiplication d'une matrice par un vecteur est récurrente dans les méthodes de type Krylov. Pour certaines classes de matrices, cette opération peut être spécialisée et ne nécessiter qu'un coût de $o(n^2)$ opérations. Il s'agit par exemple des matrices creuses (ayant $o(n^2)$ coefficients non nuls) ou des matrices structurées (descriptibles avec $o(n^2)$ coefficients). Afin de préserver leur structure, seule l'opération de multiplication à un vecteur est permise. Ces matrices sont donc vues comme des endomorphismes dont on a abstrait la structure interne. On les regroupe sous l'appellation *boîtes noires*.

L'algorithmique pour les matrices *boîtes noires* consiste donc à réduire les opérations utilisant la matrice à des produits matrice-vecteur. En reprenant les conventions de [112, §6], on notera $\mathrm{E}(n)$ le coût du produit matrice vecteur d'une matrice boîte noire carrée d'ordre n.

Le calcul du polynôme minimal d'une matrice creuse sur un corps a déjà été largement étudié : Wiedemann [119] a adapté en 1986 les méthodes itératives du calcul numérique (Lanczos, Krylov) au calcul exact. Ses travaux ont été analysés plus en détails, et appliqués à la résolution de systèmes linéaires par Kaltofen et Saunders dans [76]. Plus récemment, une version par blocs de cet algorithme a été introduite par Coppersmith [15] et analysée successivement par [74] et [113]. L'algorithme de Wiedemann conduit à des implémentations efficaces et constitue une brique de base pour l'algèbre linéaire creuse de la bibliothèque LINBOX.

En revanche, le calcul du polynôme caractéristique de matrices *boîtes noires* reste un problème largement moins maîtrisé. Kaltofen fait figurer parmi sa liste de problèmes ouverts favoris [75, Problème 3] le problème suivant :

Problème 9.1 (Kaltofen 1998)**.** *Calculer le polynôme caractéristique d'une matrice boîte noire avec $\mathcal{O}(n)$ produits matrice-vecteur, $n^2 \log(n)^{\mathcal{O}(1)}$ opérations arithmétiques supplémentaires et $\mathcal{O}(n)$ allocations mémoire supplémentaires.*

Nous présentons dans le chapitre 10 les meilleurs résultats asymptotiques obtenus pour la résolution de ce problème par Villard. Même s'ils ne résolvent pas le problème de Kaltofen, ils constituent les premiers algorithmes donnant asymptotiquement l'avantage aux méthodes *boîte noire*.

Comme dans les parties précédentes, nous étudions ensuite ce problème du point de vue de la mise en pratique. Nous présentons dans le chapitre 11 des méthodes basées sur le calcul des multiplicités des facteurs du polynôme minimal. Elles prolongent celles présentées dans la partie 9.2.2 pour les matrices *boîtes noires* et s'assemblent en deux algorithmes hybrides selon que l'on recherche l'efficacité pratique ou la meilleure complexité asymptotique. Nous appliquons ensuite ces résultats au calcul du polynôme caractéristique entier de matrices *boîtes noires* dans le chapitre 12.

10

ÉTUDE ASYMPTOTIQUE DES MÉTHODES BOÎTE NOIRE

Nous présentons ici deux algorithmes *boîte noire* pour le calcul de la forme normale de Frobenius sur un corps. Le premier, présenté dans la partie 10.1 est l'algorithme du $k^{\text{ième}}$ facteur invariant [115] pour le calcul de la forme normale de Frobenius, nécessitant $\mathcal{O}^{\sim}(n^{1,5})$ produits matrice-vecteur dans le pire cas. Dans le cadre de l'arithmétique matricielle classique et avec $E(n) = \mathcal{O}^{\sim}(n)$, il s'agit du premier algorithme *boîte noire* asymptotiquement plus rapide que les algorithmes denses ($\mathcal{O}^{\sim}(n^{2,5})$ contre $\mathcal{O}(n^3)$).

Mais, avec le produit matriciel rapide, la meilleure complexité asymptotique reste atteinte par l'algorithme dense de Keller-Gehrig par branchements en $\mathcal{O}^{\sim}(n^{2,376})$ opérations arithmétiques. Ainsi, l'étape suivante dans la résolution du problème de Kaltofen est de donner un algorithme *boîte noire* ayant un exposant inférieur à celui du produit matriciel, pour $E(n) = \mathcal{O}^{\sim}(n)$. Cela prouverait l'intérêt asymptotique de l'approche *boîte noire* pour le calcul du polynôme caractéristique.

C'est précisément ce qu'apporte l'adaptation par blocs de cet algorithme [112, Théorème 6.4.4], que nous présentons dans la partie 10.2. Grâce au produit de matrices rectangulaires, cet algorithme ne nécessite plus que $\mathcal{O}^{\sim}(n^{1,357})$ produits matrice-vecteur. Avec $E(n) = \mathcal{O}^{\sim}(n)$, on obtient la complexité $\mathcal{O}^{\sim}(n^{2,357})$, asymptotiquement meilleure que celle du produit matriciel.

Enfin, les récents travaux présentés dans [45] sur les projections de blocs efficaces, laissent entrevoir la possibilité de réduire cet exposant. Dans la partie 10.2.2, nous montrons que sous l'hypothèse de la conjecture formulée dans cet article, cette complexité se réduit à $\mathcal{O}^{\sim}(n^{2,1365})$ opérations arithmétiques, pour un produit matrice-vecteur coûtant $\mathcal{O}^{\sim}(n)$.

Pour compléter l'étude, nous mentionnons aussi l'algorithme d'Eberly [44, 43], qui propose une complexité en $\mathcal{O}(nE(n) + \phi n^2)$ où ϕ est le nombre de facteurs invariants de la matrice. Ainsi, pour de petites valeurs de ϕ cet algorithme est particulièrement intéressant, mais peut devenir trop coûteux dans le pire cas où $\phi = \mathcal{O}(n)$.

10.1 L'algorithme du k-ième facteur invariant

Cet algorithme repose sur une technique de perturbation additive permettant de calculer directement le $k^{\text{ième}}$ facteur invariant. Une recherche récursive permet ensuite de déterminer la

totalité de la forme de Frobenius en ne calculant que $\mu \log(n)$ facteurs invariants, où μ est le nombre de facteurs invariants distincts.

10.1.1 Perturbation additive de rang k

Villard réduit le calcul du $k^{\text{ième}}$ facteur invariant à celui d'un polynôme minimal grâce à l'introduction de perturbations additives de rang $k-1$. Plus précisément :

Theorème 10.1 (Villard 2000)**.** *Soit A une matrice dans $K^{n\times n}$ ayant pour facteurs invariants $f_1, \ldots, f_n$ et S un sous-ensemble fini de K. Soit $U \in K^{n\times k}$ et $V \in K^{k\times n}$ des matrices Toeplitz dont les coefficients sont choisis uniformément et aléatoirement dans S. Le polynôme minimal de $A + UV$ est $f_{k+1}t$, où $t \in K[X]$ est premier avec le polynôme minimal de A avec une probabilité supérieure à $1 - (nk + n + 1)/|S|$.*

Pour rester dans le modèle boîte noire, il faut que le produit matrice-vecteur avec les matrices U et V soient peu coûteux. C'est pourquoi nous utilisons les matrices Toeplitz, dont le produit matrice-vecteur est basé sur la Transformée de Fourrier Rapide et coûte donc $\tilde{\mathcal{O}}(n)$.

L'algorithme s'écrit donc comme suit :

Algorithme 10.1 : `InvFact` : $k^{\text{ième}}$ facteur invariant

Données : A : une matrice d'ordre n sur dans un corps K, $k \in 1 \ldots n$
Résultat : f_k : le $k^{\text{ième}}$ facteur invariant de A
début
  Calculer $f_1 = P^A_{\text{min}}$
  si *k=1* **alors**
    retourner f_1
  Choisir uniformément et indépendamment des matrice Toeplitz U et V de rang $k-1$
  Calculer $g = P^{A+UV}_{\text{min}}$
  retourner $pgcd(f_1, g)$
fin

Ainsi le calcul du $k^{\text{ième}}$ facteur invariant peut être effectué en $\mathcal{O}(n)$ produits matrice-vecteurs et $\mathcal{O}(n^2 \log n \log\log n)$ opérations supplémentaires sur le corps. La probabilité de réussite est supérieure à $1 - (n^2 + 5n + 1)/|S|$. On se reportera à [115] pour plus de détails sur ces résultats.

10.1.2 Recherche récursive

Soit $A \in K^{n\times n}$. Pour déterminer la forme normale de Frobenius, on pourrait calculer ses Φ facteurs invariants non triviaux avec l'algorithme 10.1. Or $\Phi = \mathcal{O}(n)$, ce qui conduirait à $\mathcal{O}(n^2)$ produits matrice vecteurs.

Cependant les propriétés particulières de ces facteurs invariants permettent d'éviter certains calculs. En particulier, ils vérifient :

1. $f_1 = P^A_{\text{min}}$,
2. $f_{i+1} | f_i$,

3. il y a au plus $3\sqrt{n}/2$ facteurs invariants distincts.

Pour vérifier le troisième point, on considère μ le nombre de facteurs invariants distincts, et $(g_i)_{i=1\dots\mu}$ la séquence de ces facteurs distincts. D'après la règle de divisibilité, ils sont tous de degré distinct, par conséquent le polynôme $\prod_{i=1}^{\mu} g_i$ est de degré au moins égal à $\sum_{i=1}^{\mu} i = \mu(\mu+1)/2$. On en déduit que $\mu < 3\sqrt{n}/2$.

La règle de divisibilité implique que si deux facteurs invariants f_i et f_j sont égaux, tous les facteurs invariants intermédiaires $f_k, i < k < j$ leur seront aussi égaux. Ainsi l'algorithme de recherche récursive 10.2 calcule les facteurs invariants de façon dichotomique, jusqu'à trouver deux facteurs égaux.

Algorithme 10.2 : `recherche seuils` : Recherche dichotomique des facteurs invariants distincts

Données : $[(l, f_l), (m, f_m)], l < m$

Résultat : $[(l, \psi_1), \dots, (l_k, \psi_k)]$, une décomposition de $f_l \dots f_m$ vérifiant

$$\begin{cases} f_l \dots f_m = \psi_1^{(1)}\psi_1^{(l_1)}\psi_2^{(1)}\psi_2^{(l_2)} \dots \psi_\kappa^{(1)}\psi_\kappa^{(l_\kappa)} \\ \psi_1^{(l_1)} = f_l, \psi_\kappa^{(l_\kappa)} = f_m \\ \psi_i^{(1)} = \psi_i^{(2)} = \dots = \psi_i^{(l_i)} = \psi_i, \ \ 1 \leq i \leq \kappa \\ \psi_{i+1} | \psi_i, \ \ 1 \leq i < \kappa \end{cases}$$

début

- **si** $l = m - 1$ **alors**
  - **si** $f_l = f_m$ **alors retourner** $[2, f_l]$
  - **sinon retourner** $[1, f_m, 1, f_l]$
- $k = \lceil (m+l)/2 \rceil$
- $f_k = \texttt{InvFact}(A, k)$
- **si** $f_l \neq f_k$ **alors**
  - $[k_1, \psi_1, \dots, k_r, \psi_r] = \texttt{recherche seuils}([(l, f_l), (k, f_k)])$
- **sinon**
  - $r = 1$; $[k_1, \psi_1] = [k - l + 1, f_l]$
- **si** $f_k \neq f_m$ **alors**
  - $[m_1, \tilde{\psi}_1, \dots, m_s, \tilde{\psi}_s] = \texttt{recherche seuils}([(k, f_k), (l, f_l)])$
- **sinon**
  - $s = 1$; $[m_1, \tilde{\psi}_1] = [m - k + 1, f_m]$
- **retourner** $[k_1, \psi_1, \dots, k_r + m_1 - 1, \psi_r, m_2, \tilde{\psi}_2, \dots, m_s, \tilde{\psi}_s]$

fin

L'algorithme final 10.3 consiste à effectuer la recherche dichotomique entre les deux facteurs invariants extrémaux. Cette démarche permet de n'appeler l'algorithme 10.1 que $\mu \log n$ fois, où μ est le nombre de facteurs invariants distincts. On obtient alors bien une complexité de $\mathcal{O}(\mu n \log n E(n)) + \tilde{\mathcal{O}}(\mu n^2)$ opérations dans le corps.

Algorithme 10.3 : `Forme normale de Frobenius` : par la recherche des facteurs invariants distincts

Données : A : une matrice carrée d'ordre n
Résultat : La forme de normale de Frobenius de A
début
  $f_1 = \texttt{InvFact}(A, 1)$
  $f_n = \texttt{InvFact}(A, n)$
  retourner *recherche seuils*$(1, f_1, n, f_n)$
fin

10.2 Intérêt asymptotique de l'approche boîte noire

10.2.1 Amélioration grâce aux calculs par blocs

Villard a ensuite récemment amélioré ce dernier résultat dans [112, §6.4] en introduisant une méthode par blocs dans cet algorithme. En remplaçant le calcul du polynôme minimal de $A + UV$ par celui du polynôme minimal matriciel $F_X^{A+UV,Y} \in K^{m\times m}[X]$ pour des matrices $X \in K^{m\times n}$ et $Y \in K^{n\times m}$ aléatoires, on peut calculer simultanément m facteurs invariants de la matrice A avec une grande probabilité.

Le principe est de construire la séquence des matrices $X(A+UV)^iY$ pour $i = 1 \dots 2n/m$. Le calcul des $(A+UV)^iY$ nécessite $2n/m(\mathcal{O}(mE(n)) + \mathcal{O}^\sim(mn))$ opérations. La projection à gauche sur le bloc X se fait par un produit de matrices rectangulaires en $T_{\text{MM}}(m, n, n)$. Le polynôme minimal matriciel est ensuite calculé par un algorithme d'Euclide matriciel [78, section 3] en $\mathcal{O}^\sim(m^{\omega-1}n)$. Enfin m facteurs invariants de $A+UV$ sont obtenus par le calcul de la forme normale de Smith de la matrice $F_X^{A+UV,Y}$ grâce à l'algorithme de *High Order Lifting* de Storjohann [107, Proposition 24] en $\mathcal{O}^\sim(m^{\omega-1}n)$.

La procédure de recherche binaire des facteurs invariants de l'algorithme 10.2 s'adapte facilement aux blocs de m facteurs invariants : soit μ le nombre de blocs distincts comportant m facteurs invariants. Ils ont tous un degré distinct, au moins égal à m. Le degré de leur produit est donc au moins égal à $m + m + 1 + \cdots + m + \mu - 1$ et doit être inférieur à n. Ainsi $1 + \cdots + \mu < n/m$ soit $\mu = \mathcal{O}\left(\sqrt{n/m}\right)$.

On pose $m = n^\kappa$ ($0 \le \kappa \le 1$). L'approche par blocs permet l'utilisation du produit de matrices rectangulaires. Coppersmith a établi dans [16] que $T_{\text{MM}}(n^\kappa, n, n) = \mathcal{O}\left(n^{\omega(\kappa)}\right)$ avec

$$\omega(\kappa) = \begin{cases} 2 + o(1) & \text{si } 0 \le \kappa \le \alpha = 0,2946, \\ (2(1-\kappa) + \omega(\kappa - \alpha))/(1-\alpha) & \text{si } \alpha < \kappa < 1. \end{cases}$$

Le coût total de l'algorithme s'écrit donc

$$T = \mathcal{O}^\sim\left(n^{\frac{1-\kappa}{2}}(nE(n) + n^2 + n^{\kappa(\omega-1)+1} + n^{\omega(\kappa)})\right).$$

En particulier, pour $E(n) = \mathcal{O}^\sim(n)$, on vérifie que c'est le produit de matrices rectangulaires qui domine : $T = \mathcal{O}^\sim\left(\frac{1-\kappa}{2}n^{\omega(\kappa)}\right)$. L'exposant est minimal pour $\kappa = \alpha$ ce qui donne la complexité $\mathcal{O}^\sim(n^{2,3527})$.

Ce résultat est de première importance puisqu'il s'agit du premier exposant inférieur à celui du produit matriciel, atteint par un algorithme de type boîte noire pour résoudre ce problème. Il prouve ainsi **au niveau asymptotique** qu'il est avantageux d'utiliser des méthodes boîte noire, lorsque le produit matrice-vecteur est suffisamment économique.

10.2.2 Vers l'utilisation des projections de blocs efficaces

L'opération dominante dans l'algorithme précédent était le produit de matrices rectangulaires. Il est donc naturel d'essayer de le remplacer par un produit de matrices structurées plus économique. C'est l'approche qui est développée dans [45] pour la résolution de systèmes linéaires. Les projections de blocs efficaces sont définies comme un triplet $(R, u, v) \in K^{n\times n} \times K^{m\times n} \times K^{n\times m}$ tel que :

1. les matrices de Krylov $\mathcal{K}(AR, v)$ et $\mathcal{K}({}^t(AR),\ {}^tu)$ sont inversibles,
2. le produit de R par un vecteur nécessite $\tilde{\mathcal{O}}(n)$ opérations dans K,
3. on peut appliquer des vecteurs à gauche et à droite de u et v en $\tilde{\mathcal{O}}(n)$ opérations dans K.

Dans cet article, les auteurs émettent la conjecture que de telles projections existent pour toute matrice A. Il l'auraient par ailleurs vérifié depuis [117].

En introduisant un préconditionnement par transformation de similitude ($R^{-1}AR$ au lieu de AR), utilisant par exemple des matrices Toeplitz, on pourrait alors appliquer cette approche à la projection à gauche sur la matrice X, évitant ainsi le recours à la multiplication de matrices rectangulaires. Le terme $T_{\text{MM}}(m, n, n)$ dans la complexité serait alors remplacé par $n\tilde{\mathcal{O}}(n) = \tilde{\mathcal{O}}(n^2)$. Par conséquent, l'exposant dominant dans la complexité totale serait alors

$$\max\left(\frac{1-\kappa}{2}+2, \frac{1-\kappa}{2}+\kappa(\omega-1)+1\right)$$

qui est minimal pour $\kappa = \frac{1}{\omega-1} \approx 0,72702$ et vaut $2,1365$.

Ainsi la complexité du calcul du polynôme caractéristique dans le modèle boîte noire avec $E(n) = \tilde{\mathcal{O}}(n)$ deviendrait $\tilde{\mathcal{O}}(n^{2,1365})$. La validité de ces projection de blocs par transformation de similitude doit être étudiée et devrait ainsi débouché sur une amélioration du meilleur exposant de la complexité *boîte noire* du calcul du polynôme caractéristique.

Ces algorithmes présentent un intérêt majeur au niveau théorique. Cependant, dans l'étude de la résolution en pratique du problème, ils ne sont pas encore envisageables en l'état. En particulier le calcul de la forme de Smith de matrices polynomiales demanderait d'une part le développement de toute l'arithmétique des matrices polynomiales, inexistante actuellement dans LINBOX. D'autre part, les expérimentations de Dumas et Urbanska [41] pour le calcul du déterminant ont montré que l'algorithme du *High-Order Lifting* n'était pas le plus avantageux en pratique. Enfin, les multiples facteurs logarithmiques dissimulés dans la notation $\tilde{\mathcal{O}}()$ réduisent aussi la praticabilité de tels algorithmes. Pour ces raisons, nous aborderons la question de la mise en pratique du calcul *boîte noire* du polynôme caractéristique sous un autre point de vue : le calcul de multiplicités.

11

CALCUL BOÎTE NOIRE DANS UN CORPS, PAR LA RECHERCHE DES MULTIPLICITÉS

Nous reprenons dans ce chapitre, la démarche de la partie 9.2.3 : connaissant le polynôme minimal de la matrice et sa décomposition en facteurs irréductibles, il reste à déterminer les multiplicités de chacun de ces facteurs dans le polynôme caractéristique.

Avec les matrices denses, ce calcul était effectué grâce à un algorithmes déterministe pour le calcul du polynôme caractéristique dans un corps fini. Avec les matrices *boîte noire*, nous pourrions également utiliser les algorithmes probabilistes de Villard [115] ou Eberly [44] directement, mais nous proposons des techniques alternatives visant à améliorer la complexité.

Nous présentons dans la partie 11.1 différentes méthodes pour calculer ces multiplicités, puis montrons comment les combiner en un algorithme probabiliste adaptatif efficace dans la partie 11.2. L'étude de la probabilité de succès de cet algorithme est inachevée. Menée à terme, elle conduirait à un algorithme nécessitant $\mathcal{O}(n\sqrt{n})$ produits matrices vecteurs, ce qui améliorerait d'un facteur $\log n$ la complexité de [115].

Dans ce chapitre, nous considérons une matrice A sur un corps parfait, dont nous rappelons la définition :

Définition 11.1 (Corps parfait)**.** Un corps est parfait si toutes ses extensions algébriques sont séparables.

Nous utiliserons la caractérisation des corps parfaits selon laquelle les racines d'un polynôme irréductible dans une clôture algébrique d'un corps parfait sont simples. Puisque tout corps fini est parfait, il s'agit simplement d'un cadre plus général. En pratique, nous nous intéressons essentiellement aux corps finis. Soit $P^A_{\text{min}} = \prod_{i=1}^{k} P_i^{e_i}$ la factorisation du polynôme minimal de A en facteurs irréductibles unitaires. Le polynôme caractéristique s'écrit alors $P^A_{\text{car}} = \prod_{i=1}^{k} P_i^{m_i}$ pour des $m_i \geq e_i$. Enfin, nous noterons d_i les degrés des facteurs : $d_i = \text{degree}(P_i)$.

11.1 Heuristiques de calcul des multiplicités

Nous présentons trois techniques basées sur des calculs *boîte noire* avec la matrice A, pour déterminer les multiplicités m_i. La première, (partie 11.1.1) utilise le lien entre la nullité de

la matrice $P_i(A)$ et la multiplicité m_i. La deuxième (section 11.1.2) est une simple recherche combinatoire sur les valeur possibles des m_i. Enfin, dans la dernière (section 11.1.3), les multiplicités sont les solutions d'un système linéaire obtenu en évaluant le polynôme caractéristique en des valeurs aléatoires.

11.1.1 Méthode des nullités

Définition 11.2. La nullité d'une matrice A, notée $\nu(A)$ est la dimension de son noyau

Soit P un polynôme et C_P la matrice compagnon associée. Nous rappelons que le polynôme minimal et le polynôme caractéristiques de C_P sont égaux à P.

Nous rappelons aussi la définition de la forme primaire de la matrice A, aussi appelée la seconde forme de Frobenius dans [50] : *toute matrice A est similaire à une somme directe de matrices compagnons ; chacune étant associée à une puissance d'un polynôme irréductible.* Cette forme est unique, à l'ordre des blocs près. Elle peut être vue comme une décomposition plus avancée de la forme normale de Frobenius : chaque facteur invariant est décomposé en un produit de facteurs irréductibles élevé à une certaine puissance.

Exemple 11.1. La matrice

$$A = \text{Diag}(C_{X^3-X^2}, C_{X^2-X}) = \begin{bmatrix} 0 & 0 & 0 & & & \\ 1 & 0 & 0 & & & \\ 0 & 1 & 1 & & & \\ & & & 0 & 0 & \\ & & & 1 & 1 & \end{bmatrix}$$

est sous forme normale de Frobenius et sa forme primaire est :

$$\text{Diag}(C_{X-1}, C_{X^2}, C_{X-1}, C_X) = \begin{bmatrix} 1 & & & & & \\ & 0 & 0 & & & \\ & 1 & 0 & & & \\ & & & 1 & & \\ & & & & 0 & \end{bmatrix}$$

La méthode des nullités s'appuie sur le lemme suivant :

Lemme 11.1. *Avec les notations précédentes,*

$$\nu(P_i^{e_i}(A)) = m_i d_i$$

Démonstration. Soit F la forme primaire de A sur K : $\exists\ U \in \text{GL}_n(K),\ U^{-1}AU = F$. F est diagonale par blocs, ses blocs diagonaux sont des matrices compagnons associées à des puissances des polynômes irréductibles P_j. Comme ces polynômes sont issus des facteurs invariants de la matrice, la plus grande puissance avec laquelle un facteur P_j peut apparaître dans un bloc compagnon est sa multiplicité e_j dans le polynôme minimal.

Alors

$$P_i^{e_i}(A) = U^{-1} P_i^{e_i}(F) U = U^{-1} \text{Diag}(P_i^{e_i}(C_{P_j^{k_j}})) U.$$

D'une part $P_i^{e_i}(C_{P_i^k}) = 0 \ \forall k \leq e_i$. D'autre part la matrice $P_i^{e_i}(C_{P_j^k})$ est régulière pour $j \neq i$ car P_i et P_j sont premiers entre eux.

Par conséquent, la nullité de $P_i^{e_i}(A)$ correspond exactement à la dimension totale des blocs $C_{P_i^k}$, qui vaut $m_i d_i$. □

On en déduit directement l'algorithme 11.1 pour le calcul de la multiplicité m_i d'un facteur P_i.

Algorithme 11.1 : `Nullité`

Données : A, une matrice $n \times n$; P_i un facteur irréductible de P^A_{min} et e_i sa multiplicité dans P^A_{min}
Résultat : m_i, la multiplicité de P_i dans P^A_{char}
début
  $r = \text{rang}(P^{e_i}(A))$;
  retourner $m_i = (n - r)/\text{degree}(P_i)$;
fin

Propriété 11.1. *L'algorithme 11.1 calcule la multiplicité de P_i dans le polynôme caractéristique de A en $\mathcal{O}(e_i d_i n E(n))$ opérations dans le corps.*

Démonstration. En utilisant le schéma de Horner, on peut créer la matrice *boîte noire* $P^{e_i}(A)$ dont l'application à un vecteur ne coûte que $E'(n) = d_i e_i E(n)$ opérations sur le corps. Le calcul du rang de cette matrice peut être effectué en $\mathcal{O}(nE'(n))$ opérations, en utilisant l'algorithme de Wiedemann combiné avec des préconditionneurs [42]. □

Cet algorithme, adapté pour des petites valeurs de $e_i d_i$, est beaucoup trop coûteux quand $e_i d_i$ devient grand. Nous présentons comment déterminer *partiellement* la multiplicité quand de telles situations ont lieu.

Sur la forme primaire de A, la multiplicité m_i d'un facteur P_i est formée par la contribution de plusieurs blocs compagnons $C_{P_i^j}$ pour $j \in [1 \dots e_i]$. Si e_i est grand, seuls quelques blocs avec des j grands peuvent exister, à cause de la limitation de la dimension totale. A l'inverse, les blocs pour un j petit peuvent être nombreux, mais leur nullité est facile à calculer.

On pose $n_{i,j}$, le nombre d'occurrences du bloc $C_{P_i^j}$ dans la forme primaire de A. La multiplicité m_i s'obtient directement à partir de $n_{i,j}$ par la relation

$$m_i = \sum_{j=1}^{e_i} j n_{i,j}. \tag{11.1}$$

Nous allons montrer comment calculer $n_{i,j}$, pour des petites valeurs de j, à partir de l'algorithme 11.1. Cette donnée *partielle* de la multiplicité, réduit le nombre d'affectations possibles pour les $n_{i,j}$ restants. Ceux-ci pourront être déterminés par exemple par une recherche combinatoire (voir partie 11.1.2).

Commençons par ce résultat :

Theorème 11.1. *Soit P un polynôme irréductible de degré d sur un corps parfait K et k et $e \geq 1$ deux entiers. Alors,*

$$\nu(P^k(C_{P^e})) = \textit{min}(k, e)d.$$

Démonstration. Soit $A = C_{P^e}$ et $B = P^k(A)$.

Si $k \geq e$, alors P^k est un multiple du polynôme minimal de A. Donc B est la matrice nulle et sa nullité égale sa dimension.

Considérons maintenant que $k < e$. Soit $\overline{K}$ une extension de K telle que le polynôme P soit scindé sur $\overline{K}$: $P = \prod_{i=1}^{d} P_i$. Ces facteurs sont tous différents car K est un corps parfait.

Le polynôme minimal de A dans $\overline{K}$ est toujours P^e (en appliquant [50, equation (54)] soit sur K soit sur $\overline{K}$). Donc la forme normale de Frobenius de A sur $\overline{K}$ est C_{P^e} et sa forme primaire est $F = \text{Diag}(C_{P_i^e})$. Plus précisément, il existe $U \in M_n(\overline{K})$ tel que $A = U^{-1}FU$. Ainsi, $B = U^{-1}P^k(F)U = U^{-1}\text{Diag}(P^k(C_{P_i^e}))U$.

Si $k = 1$: le polynôme minimal de chaque $P(C_{P_i^e})$ est X^e donc leur forme de Frobenius est C_{X^e}. Il existe donc $V \in M_n(\overline{K})$ tel que

$$B = U^{-1}V^{-1}\text{Diag}(\underbrace{C_{X^e}, \ldots, C_{X^e}}_{d \text{ fois}})VU.$$

Enfin, le rang de C_{X^e} étant $e-1$, on en déduit que $\text{rang}(B) = d(e-1)$. La nullité de B vaut donc $\nu(B) = d$.

Pour k quelconque

$$B = U^{-1}V^{-1}\text{Diag}(\underbrace{(C_{X^e})^k, \ldots, (C_{X^e})^k}_{d \text{ fois}})VU.$$

Or C_{X^e} est la matrice nilpotente de d'ordre e dont la sous-diagonale est formée de 1. Sa puissance $k^{\text{ième}}$ est de rang $e-k$. Ainsi $\text{rang}(B) = (e-k)d$ et la nullité de B est $\nu(B) = kd$. □

On applique ensuite ce résultat à une matrice A sur K, dont le polynôme minimal admet la factorisation en polynômes irréductibles $P_{\text{min}}^A = \prod_{i=1}^{k} P_i^{e_i}$. On note $\nu_{i,j} = \nu(P_i^j(A))$ la nullité de $P_i^j(A)$.

La nullité de $P_i(A)$, vue sur sa forme primaire, se décompose en la somme des nullités de chaque $P_i(C_{P_i^k})$:

$$\nu_{i,1} = \sum_{k=1}^{e_i} n_{i,k} d_i. \tag{11.2}$$

Si on applique P_i^j à A, chaque $P_i^j(C_{P_i^k})$ pour $k \leq j$ sera un bloc nul et contribuera donc de kd_i à la nullité. Et $k > j$, la contribution à la nullité est de jd_i. On obtient ainsi :

$$\nu_{i,j} = \sum_{k=1}^{j} n_{i,k} k d_i + \sum_{k=j+1}^{e_i} n_{i,k} j d_i. \tag{11.3}$$

Calculons

$$\frac{1}{j-1}\nu_{i,j-1} = \frac{1}{j-1}\sum_{k=1}^{j-1} n_{i,k} k d_i + n_{i,j} d_i + \sum_{k=j+1}^{e_i} n_{i,k} d_i.$$

Et comme

$$\nu_{i,j+1} - \nu_{i,j} = \sum_{k=j+1}^{e_i} n_{i,k} d_i,$$

on en déduit que

$$n_{i,j} = \frac{1}{d_i}\left(\frac{1}{j-1}\nu_{i,j-1} + \nu_{i,j} - \nu_{i,j+1}\right) - \frac{1}{j-1}\sum_{k=1}^{j-1} n_{i,k} k.$$

On peut alors exprimer les formules permettant de calculer les $n_{i,j}$ de proche en proche dans le corollaire 11.2.

Corollaire 11.2.

$$\begin{aligned}
n_{i,1} &= (2\nu_{i,1} - \nu_{i,2})/d_i \\
n_{i,j} &= \frac{1}{d_i}\left(\frac{1}{j-1}\nu_{i,j-1} + \nu_{i,j} - \nu_{i,j+1}\right) - \frac{1}{j-1}\sum_{k=1}^{j-1} n_{i,k} k \;\; \forall j \in [1 \dots e_i] \\
n_{i,e_i} &= \frac{\nu_{i,e_i}}{e_i d_i} - \frac{1}{e_i}\sum_{k=1}^{e_i-1} n_{i,k} k
\end{aligned}$$

La dernière formule pour n_{i,e_i} est donnée ici par souci d'exhaustivité. En pratique, le calcul de la nullité de $P_i^{e_i}$ suffit à déterminer la multiplicité m_i (lemme 11.1) et il est alors inutile de calculer les occurrences $n_{i,j}$. Celles-ci sont utiles dans le cas où l'on n'en calcule qu'une partie. Nous allons maintenant décrire la recherche combinatoire permettant de déterminer les $n_{i,j}$ restants.

11.1.2 Recherche combinatoire

Il s'agit d'une stratégie de type *Branch and Bound* : toutes les affectations possibles pour les occurrences inconnues sont testées et les branches pour lesquelles le degré total dépasse la dimension n sont coupées.

Pour tout polynôme irréductible P_i, on définit k_i tel que, les occurrences $n_{i,j}$ ont été calculées pour tout $j \leq k_i$.

Les $n_{i,j}$ vérifient l'équation du degré total : $\sum_{i,j} j d_i n_{i,j} = n$. On définit alors l'objectif Ω comme le degré total que doivent former les occurrences inconnues :

$$\Omega = n - \sum_i \sum_{j=1}^{k_i} j d_i n_{i,j}. \tag{11.4}$$

Encore une fois, le fait que jd_i soit grand assure que les $n_{i,j}$ seront petits et réduit donc le nombre de candidats possibles.

Un autre test facilement réalisable est celui de la trace : le coefficient du monôme de degré $n-1$ du polynôme caractéristique doit être égal à la trace de la matrice. Or, si $\forall i \; P_i = X^{d_i} + t_i X^{d_i - 1} + \dots$, le coefficient du monôme de degré $n-1$ de $\prod_i P_i^{m_i}$ vaut

$$\mathrm{Tr}(A) = -\sum_i t_i m_i. \tag{11.5}$$

On peut donc facilement discriminer les candidats obtenus par la recherche combinatoire par ce test.

L'algorithme 11.2 résume cette méthode.

Algorithme 11.2 : `recherche combinatoire`

Données : Ω : l'objectif, $N = (N_{i,j})$: un vecteur de bornes inférieures sur les occurrences inconnues $n_{i,j}$
Résultat : sol est une liste de candidats vérifiant les équations (11.4) et (11.5)
début
  si $\Omega < 0$ **alors**
    exit
  sinon si $\Omega = 0$ **alors**
    si $Tr(A) = -\sum_i t_i \left(\sum_{j=1}^{e_i} j N_{i,j}\right)$ **alors**
      sol.empile(N)
  sinon
    pour tout $N_{i,j}$ **faire**
      $N_{i,j}$++
      `recherche combinatoire`(sol, $\Omega - jd_i, N$)
      $N_{i,j}$- -
fin

Enfin, dans le cas où plusieurs candidats subsistent, on les discriminera par autant d'évaluations aléatoires du polynôme caractéristique que nécessaire. Pour une valeur aléatoire λ, il s'agit de calculer le déterminant $\det(\lambda I - A)$ et de le comparer avec la valeur du polynôme reconstruit en λ. L'algorithme 11.3 résume la stratégie de calcul des multiplicités des grands facteurs par recherche combinatoire.

Cet algorithme pourrait être utilisé seul, pour déterminer toutes les multiplicités. Mais sa complexité exponentielle (due à l'explosion combinatoire de la recherche de l'algorithme 11.2) peut le rendre impraticable. Il n'est donc à utiliser qu'en complément d'un autre algorithme, comme celui des nullités, afin de calculer les occurrences des blocs de grande dimension.

11.1.3 Résolution du système logarithmique

Nous présentons ici une méthode alternative pour déterminer les multiplicités, par la résolution d'un système linéaire.

Principe

On se place dans un corps fini GF(q) où q est suffisamment grand devant n : $q - 1$ possède un facteur premier supérieur à n. Une évaluation du polynôme caractéristique en la valeur λ_i forme l'équation

$$\prod_{j=1}^{k} P_j^{m_j}(\lambda_i) = \det(\lambda_i I - A)$$

Algorithme 11.3 : `Multiplicité des grands facteurs`

Données : A, une matrice $n \times n$; $d = (d_i)_i$, les degrés des facteurs irréductibles P_i de P^A_{char} et $k = (k_i)_i$ tels que $n_{i,j}$ est calculée $\forall j \leq k_i$.
Résultat : $N = (n_{i,j}) \forall j > k_i$
début
  $N = (0, \ldots, 0)$
  sol = []
  $\Omega = n - \sum_{i=1}^{k_i} j d_i n_{i,j}$
  `recherche combinatoire`(sol, Ω, N)
  tant que # *sol* > 1 **faire**
    Tirer $\lambda \in K$ uniformément
    $d = \det(\lambda I - A)$
    pour tout $N \in$ *sol* **faire**
      si $\prod (P_i^j)^{n_{i,j}}(\lambda) \neq d$ **alors**
        rejeter N
  retourner $N = sol.$*dépiler*()
fin

dont les inconnues sont les multiplicités m_j. Si λ_i n'est pas une racine du polynôme caractéristique, les logarithmes discrets des membres de cette équation conduisent à l'équation

$$\sum_{j=1}^{k} m_j \log(P_j(\lambda_i)) = \log(\det(\lambda_i I - A)) \mod q - 1 \tag{11.6}$$

qui est linéaire en les m_j modulo $q - 1$.

On peut alors former un système linéaire de dimensions $l \times k$ en choisissant l valeurs λ_i aléatoirement, en dehors des racines de P^A_{min}. Ce système est consistant car les multiplicités m_i en sont une solution. Si il est régulier, avec $k \leq l$, l'unique solution modulo $q-1$ plongée dans $\mathbb{Z}$ correspondra au vecteur des multiplicités m_i.

La résolution du système pourra être effectuée soit par une élimination de Gauss dans l'anneau $\mathbb{Z}_{q-1}$ ou, de façon plus efficace, dans le corps fini $\mathbb{Z}_p$ où p est un grand facteur de $q - 1$ (supérieur à n). Dans ce dernier cas, le résultat sera correct tant que le système restera régulier modulo p.

L'algorithme

L'algorithme 11.4 précise le principe décrit précédemment.

k est le nombre de multiplicités m_i à calculer. La première étape consiste à choisir k valeurs λ_i afin de former une matrice B inversible. En première approche, on pourrait se contenter de tirer aléatoirement ces k valeurs. Mais comme nous le verrons dans la preuve de la proposition 11.2, le coût de la construction d'une ligne de cette matrice est négligeable devant les autres opérations de l'algorithme. On choisit donc d'en calculer $l \geq k$, afin d'augmenter la probabilité de former une matrice de rang k. L'algorithme calcule ainsi l lignes de G en choisissant l λ_i

Algorithme 11.4 : `système logarithmique discret`

Données : A, une matrice $n \times n$ sur un corps fini GF(q) ; $(P_i)_{i=1...k}$, les facteurs irréductibles de son polynôme minimal ; $l \geq k$ un entier
Résultat : $(m_i)_{i=1...k}$, les multiplicités des P_i dans P^A_{car}
début
  Choisir uniformément l $\lambda_i \in$ GF(q) distincts, non racines de P^A_{min}
  Former la matrice $l \times k$ $G = [g_{i,j}]$ où $g_{i,j} = \log(P_j(\lambda_i)) \mod q-1$
  Soit p un facteur premier de $q-1$ tel que $p > n$
  $(L, Q, U, P) = \texttt{LQUP}(G) \mod p$
  $B = \texttt{FRL}(G)$; `/* Forme Réduite en Ligne de G :` $B = [I_k 0]Q^T G$ `*/`
  Former le vecteur $b = [b_i]$ où $b_i = \log(det(\lambda_i I - A)) \mod p$
  Résoudre $Bx = b \mod p$
  retourner (x_i)
fin

aléatoirement. Puis B s'obtient en calculant la forme réduite par ligne de G (voir partie 4.1). Une fois les k λ_i sélectionnés, on peut former le vecteur b en calculant les k déterminants $\det(\lambda_i I - A)$. La résolution du système utilise ensuite la décomposition `LUP` de B, lisible sur la décomposition `LQUP` de G.

Propriété 11.2. *Si la matrice G est de rang k, l'algorithme 11.4 calcule les multiplicités en $\mathcal{O}(knE(n))$ opérations arithmétiques, en choisisant l tel que $l = \mathcal{O}(nE(n)k^{2-\omega})$ et $l = \mathcal{O}(kE(n))$.*

Démonstration. La construction de chaque ligne de la matrice se fait par une évaluation des polynômes suivant un schéma de Horner. La somme des degrés des facteurs P_i étant inférieure à n, la construction d'une ligne de la matrice G coûte $\mathcal{O}(n)$ opérations dans Z_q. Le calcul des logarithmes discrets des coefficients peut se faire de deux façons :

- en tabulant la correspondance entre les éléments de GF(q) et leur logarithme discret [31], ce qui nécessite $\mathcal{O}(q)$ allocations mémoire.
- en triant d'abord les coefficients de G et b. Puis en calculant itérativement toutes les puissances d'un générateur et en établissant les correspondances avec les coefficients triés. Le surcoût mémoire n'est alors que de $\mathcal{O}(lk)$ pour un $\tilde{\mathcal{O}}(lk+q)$ opérations supplémentaires dans GF$_q$.

Chaque calcul de déterminant nécessite $\mathcal{O}(n)$ produits matrice-vecteur avec la matrice A dans Z_p [110, §3.1]. La décomposition `LQUP` nécessite $\mathcal{O}(lk^{\omega-1})$ opérations dans Z_p et la résolution du système, $\mathcal{O}(k^2)$.

La complexité totale est donc $\mathcal{O}(knE(n) + nl + lk^{\omega-1})$ soit, $\mathcal{O}(knE(n))$ en choisissant l tel que $l = \mathcal{O}(kE(n))$ et $l = \mathcal{O}(nE(n)k^{2-\omega})$. □

L'analyse de cet algorithme est incomplète. Si nous pouvons donner sa complexité, nous n'avons pas pu fournir une borne sur la probabilité que la matrice G soit de rang k. Les expérimentations pratiques confirment l'intuition selon laquelle, seulement k valeurs λ_i choisies aléatoirement suffisent à former une matrice de rang k, et la marge permise sur le paramètre

l devrait permettre d'assurer une probabilité de succès grande. Cependant, nous laissons cette question ouverte, pour l'instant.

Remarque 11.1. La construction des lignes de la matrice peut être intégrée dans le processus d'élimination `LQUP`. Ceci permet de terminer leur calcul de façon anticipée, lorsque le rang k est atteint.

Même si chacune de ces trois techniques permet la détermination de toutes les multiplicités, leur complexité est trop élevé. En revanche, leurs domaines de prédilection étant disjoints, des algorithmes adaptatifs permettent de les combiner afin de tirer parti de chacun de leurs avantages.

11.2 Algorithmes adaptatifs

Nous présentons ici comment combiner les algorithmes des parties précédentes, entre elles et avec la méthode du $i^{\text{ème}}$ facteur invariant de Villard [115]. Les algorithmes hybrides obtenus sont adaptatifs, dans le sens où ils tiennent compte des propriétés des données pour choisir le meilleur algorithme. On se reportera à [21] pour plus de détails sur la définition des algorithmes hybrides, adaptatifs, ...

Nous montrons d'abord comment combiner l'algorithme de résolution du système logarithmique avec le calcul des nullités ou une recherche combinatoire. Nous présentons ensuite un premier algorithme pour l'étude asymptotique. Enfin nous donnerons certaines améliorations à ce dernier pour une meilleure efficacité pratique.

11.2.1 Méthode des nullités et recherche combinatoire

Une première idée est de combiner les algorithmes 11.1, calculant des nullités, et 11.2, complétant les occurrences manquantes par une recherche combinatoire.

On suppose connu le polynôme minimal P^A_{min} et sa décomposition en facteurs irréductibles $P^A_{\text{min}} = \prod_{i=1}^{k} P_i^{e_i}$. On rappelle que le $n_{i,j}$ est le nombre d'occurrences du bloc compagnon $C_{P_i^j}$ dans la forme primaire de A. Ces deux algorithmes sont complémentaires puisque le premier est efficace dans la recherche des occurrences des facteurs de petit degré élevés à une petite puissance (jd_i petit), alors que la recherche combinatoire admet d'autant moins de branches que le degré est grand et que le facteur est élevé à une puissance grande (jd_i grand).

Ainsi, l'algorithme hybride consiste à dresser la liste des occurrences $n_{i,j}$ à déterminer, en les classant par jd_i croissant. On calcule ensuite successivement les nullités des matrices $P_i^j(A)$ jusqu'à ce qu'il reste un certain nombre de *grands* facteurs. Ceux-ci sont traités par la recherche combinatoire et on calcule les $n_{i,j}$ ainsi que les multiplicités m_i par le corollaire 11.2.

Cependant la recherche combinatoire dévient rapidement très coûteuse. Il faut donc limiter son utilisation à la détermination d'un nombre arbitraire et fixe d'occurrences. Expérimentalement, cinq facteurs semblait correspondre au meilleur compromis sur les matrices que nous avons manipulées (voir partie 12.3.2). Il faut donc essayer de combiner cet algorithme avec d'autres, pour traiter le cas de facteurs de grand degré trop nombreux.

Algorithme 11.5 : `Nullité et recherche combinatoire`

Données : A, une matrice $n \times n$ sur une corps fini ; $(P_i)_{i=1\dots k}$, les facteurs irréductibles de son polynôme minimal
Résultat : $(m_i)_{i=1\dots k}$, les multiplicités des P_i dans P^A_{car}
début
  Soit $\mathcal{E} = \{(i,j), i \in [1\dots k], j \in [1\dots e_i]\}$
  Trier $\mathcal{E}$ selon les valeurs croissantes des jd_i
  tant que *(*$\#\mathcal{E} > 5$*)* **faire**
    Dépiler (i,j), le premier élément de $\mathcal{E}$
    Calculer $\nu_{i,j} = n - \text{rang}(P_i^j(A))$
  pour tout $i \in [1\dots k]$ **faire**
    Soit j_i le plus grand indice tel que ν_{i,j_i} a été calculé
    si $j_i < e_i$ **alors**
      Calculer $\nu_{i,j_i+1} = n - \text{rang}(P_i^{j_i+1}(A))$
    Calculer les $n_{i,k} \forall k \in [1\dots j_i]$ d'après le corollaire 11.2
  `Recherche des grands facteurs`$(A, (d_i)_i, (j_i)_i)$
  retourner $(m_i = \sum_{j=1}^{e_i} j n_{i,j})_i$
fin

11.2.2 Réduction de la taille du système logarithmique

On peut restreindre l'algorithme 11.4 au calcul d'un sous-ensemble de multiplicités : soit $\mathcal{C}$ l'ensemble des indices des multiplicités déjà connues, alors l'équation (11.6) dévient

$$\sum_{j \notin \mathcal{C}} m_j \log(P_j(\lambda_i)) = \log(\det(\lambda_i I - A)) - \sum_{j \in \mathcal{C}} m_j \log(P_j(\lambda_i)) \mod q-1.$$

Il suffit alors de prendre $\forall i\ b_i = \log(\det(\lambda_i I - A)) - \sum_{j\in\mathcal{C}} m_j \log(P_j(\lambda_i)) \mod q-1$ dans l'algorithme.

On peut ainsi combiner cet algorithme avec celui des nullités : les facteurs de degré 1 et de multiplicité 1 dans le polynôme minimal ($d_i e_i = 1$) sont traités par l'algorithme des nullités, et le restant est laissé à la résolution du système logarithmique. Cette approche est toujours bénéfique car elle remplace le calcul d'un déterminant par celui d'un rang, qui est moins coûteux, et réduit la dimension du système à résoudre.

On peut aussi combiner cet algorithme avec la recherche combinatoire : on définit $\mathcal{C}$ comme l'ensemble des s facteurs P_i de plus grand degré (pour un s convenablement choisi). On dresse ensuite par recherche combinatoire la liste de toutes les affectations possibles pour les multiplicités des $(P_i)_{i\in\mathcal{C}}$. Pour chaque affectation possible $(m_j^{(k)})_{j\in\mathcal{C}}$, indicée par k, les multiplicités restantes sont déterminées par la résolution du système logarithmique de la forme :

$$\forall i\ \sum_{j \notin \mathcal{C}} m_j \log(g_j(\lambda_i)) = \log(\det(\lambda_i I - A)) - \sum_{j \in \mathcal{C}} m_j^{(k)} \log(h_j(\lambda_i)) \mod q-1.$$

Il s'agit d'une résolution d'un système linéaire multiple (avec une matrice inconnue). La construction de la matrice et le calcul des déterminants sont communs à tous les systèmes. Il

suffit donc de remplacer les résolutions de systèmes triangulaires (avec L et U de la décomposition `LUP` de B) par des résolutions de systèmes triangulaires multiples. Ainsi on peut réduire la dimension du système logarithmique en utilisant la recherche combinatoire. En contre partie, des opérations supplémentaires sont nécessaires pour les résolutions de systèmes triangulaires multiples mais un choix judicieux de s assure que celles-ci seront amorties dans le coût total.

Soit k le nombre total de facteurs et soit $h = \min_{P_i \in \mathcal{C}} (\text{degré}(P_i))$. Le nombre total d'affectations possibles pour les $(m_i)_{i \in \mathcal{C}}$ est borné par $\frac{(n/h)^s}{s!}$. Cette méthode économise, $snE(n) + 2/3(k^3 - (k-s)^3$ opérations (s déterminants et l'élimination d'une matrice de dimension $k - s$ au lieu de k). Elle engendre par ailleurs un surcoût de $\frac{(n/h))^s}{s!}k^2$ pour les résolutions de systèmes triangulaires.

On peut donc imaginer un algorithme *introspectif*, déterminant s de façon à maximiser la fonction

$$snE(n) + 2/3(k^3 - (k-s)^3) - \frac{(n/h))^s}{s!}k^2.$$

La valeur de s étant petite, la maximisation se fait par le calcul explicite des premières valeurs de cette fonction pour $s = 1, 2, \ldots$

Par exemple, soit une matrice d'ordre 2000, dont le polynôme minimal se décompose en 2 facteurs de degré 100, 2 facteurs de degré 50 et 6 facteurs de dégré 10. La valeur optimale pour s sera 3, ce qui revient à calculer toutes les affectations possibles pour les 2 facteurs de degré 100 et pour un des facteurs de degré 50.

11.2.3 Un algorithme asymptotiquement performant

Le premier algorithme *boîte noire* de Villard [115] est le premier à avoir atteint une complexité sous-cubique pour le calcul de la forme normale de Frobenius sans faire appel au produit matriciel rapide. Il repose sur le calcul de seulement $\mu \log(n)$ facteurs invariants où μ est le nombre de facteurs invariants distincts de la matrice. Comme $\mu < 3\sqrt{n}/2$ et qu'un facteur invariant peut être calculé en $\mathcal{O}(n)$ produits matrice-vecteur, l'algorithme nécessite donc $\mathcal{O}\left(n^{3/2}\log(n)E(n)\right)$ opérations sur le corps, ainsi que $\mathcal{O}\left(n^{5/2}\log^2(n)\text{loglog}(n)\right)$ opérations supplémentaires pour les préconditionnements.

Nous proposons l'algorithme adaptatif 11.6 qui utilise la technique de l'algorithme de Villard pour calculer le $k^{\text{ième}}$ facteur invariant et la résolution du système logarithmique présentée dans la partie 11.1.3. Le principe est de limiter l'utilisation de l'algorithme de Villard au calcul des $\sqrt{n}$ premiers facteurs invariants. Une fois ce calcul effectué, il y a moins de $\sqrt{n}$ multiplicités inconnues, et l'algorithme 11.4 peut alors s'appliquer efficacement.

Propriété 11.3. *L'algorithme 11.6 calcule le polynôme caractéristique P^A_{car} en $\mathcal{O}(n\sqrt{n})$ produits matrice-vecteur avec la matrice A et $\mathcal{O}\left(n^{5/2}\log(n)\log\log(n)\right)$ opérations supplémentaires sur le corps.*

A nouveau, il manque une borne sur la probabilité de réussite de cet algorithme. Si elle peut être fournie, alors cet algorithme améliore d'un facteur $\log(n)$ le nombre de produits matrice-vecteurs dans la complexité de l'algorithme de Villard [115, Theorem 3].

Démonstration. La boucle `tant que` est exécutée au plus $\sqrt{n}$ fois. En effet, sinon il y aurait plus de $\sqrt{n}$ facteurs invariants ayant plus de $\sqrt{n}$ facteurs irréductibles. Il y a donc $\mathcal{O}(\sqrt{n})$ appels à la fonction `InvFact`.

Algorithme 11.6 : `Boîte Noire Adaptatif`

Données : A, une matrice $n \times n$ sur un corps fini GF(q)
Résultat : P_{car}^A le polynôme caractéristique de A
début
  $f_1 = \texttt{InvFact}(1)$
  Factoriser $f_1 = \prod_{i=1}^{k} P_i^{e_i}$ par l'algorithme de Cantor-Zassenhaus
  Soit $S = \{P_i, i = 1 \dots k\}$, $j = 2$
  tant que $(\#S > \sqrt{n})$ **faire**
    $f_j = \texttt{InvFact}(j)$
    pour tout $P_i \in S$ **faire**
      Calculer α tel que $\gcd(P_i^{e_i}, f_j) = P_i^{\alpha}$
      si $\alpha = 0$ **alors**
        $S = S \backslash \{P_i\}$
      sinon
        m_i += α
    j++
  Tirer uniformément $k = \#S$ différents $\lambda_i \in \text{GF}(q)$ tels que $f_1(\lambda_i) \neq 0$
  Construire la matrice $G = [g_{i,j}]$ où $g_{i,j} = \log(P_j(\lambda_i))$ for $P_j \in S$
  Construire le vecteur $b = [b_i]$ où $b_i = \log(\det(\lambda_i I - A)) - \sum_{P_j \notin S} m_j \log(P_j(\lambda_i))$
  Résoudre $Ax = b$
  retourner $P = \prod_{j \in S} P_j^{x_j} \prod_{j \notin S} P_j^{m_j}$
fin

La condition de sortie de la boucle assure par ailleurs que la dimension du système logarithmique sera inférieure à $\sqrt{n}$. En prenant $k = \sqrt{n}$ dans la proposition 11.2 on obtient la complexité annoncée. □

11.2.4 Améliorations pour la mise en pratique

Dans l'algorithme de Villard, [115], une recherche récursive permet à l'algorithme d'éviter le calcul de certains facteurs invariants répétés. L'idée est de calculer le premier et le dernier facteur invariant, puis de calculer les invariants intermédiaires en suivant une découpe dichotomique. Lorsque deux invariants d'ordre i et j sont égaux, la propriété de divisibilité assure que tous les facteurs invariants intermédiaires leur seront aussi égaux. Ainsi, l'algorithme ne calcule que $\sqrt{n} \log n$ au lieu de n dans le pire cas. On peut adapter cette recherche pour le calcul des $\sqrt{n}$ premiers facteurs invariants dans l'algorithme 11.6 : en pratique cela évite dans de nombreux cas des calculs inutiles, mais n'améliore pas la complexité asymptotique. En effet, dans le pire cas, il peut y avoir $\mathcal{O}(\sqrt{n})$ facteurs invariants distincts dans les $\sqrt{n}$ premiers.

Enfin, on peut aussi incorporer dans l'algorithme 11.6 les techniques de la partie 11.2.2. A la fin de la boucle `tant que`, et avant la construction du système linéaire, on peut calculer les nullités des petits facteurs ($e_i d_i = 1$) et tenter une résolution par recherche combinatoire.

En effet, le calcul de la nullité d'un facteur de degré 1 est un simple calcul de rang, plus

Algorithme 11.7 : `Facteurs de degré 1`

pour tout $P_i \in S$ *tel que* $d_i e_i = 1$ **faire**
  $m_i = \nu(P_i(A))$
  $S = S \backslash \{P_i\}$
si *(*$\#S < T$*)* **alors**
  `Multiplicité des grands facteurs`$(A, (d_i), (0))$

économique qu'un déterminant ($r = \text{rang}(P_i(A)) < n$ applications de la *boîte noire*, contre n pour le déterminant). Par ailleurs cela fait décroître la dimension du système logarithmique à résoudre. La recherche combinatoire est effectuée pour un nombre de facteurs inférieur à un seuil T qui pourra être déterminé soit par une estimation introspective de la complexité de la recherche, soit fixé empiriquement. En pratique, nous avons vérifié que $T = 5$ convenait dans la plupart des situations. Ces modifications ne modifient pas la complexité asymptotique mais rendent l'algorithme plus efficace en pratique.

12

CALCUL BOÎTE NOIRE DANS L'ANNEAU DES ENTIERS

A nouveau, notre propos n'est pas de fournir de nouvelles méthodes pour le calcul entier, mais d'adapter des techniques existantes pour pouvoir mettre en pratique nos algorithmes pour les corps finis. Nous présentons donc deux méthodes basées sur le calcul des multiplicités : l'une utilisant directement l'algorithme 11.6 et la remontée multifacteurs de Hensel, proposée par Storjohann, l'autre appliquant la méthode des multiplicités directement sur $\mathbb{Z}$. Enfin, nous donnons les résultats pratiques obtenus dans l'application de ces algorithmes à un problème concret de théorie des graphes.

12.1 Par le calcul des multiplicités et la remontée de Storjohann

Comme nous l'avons mentionné dans la partie 9.2.2, Storjohann propose dans [105] une méthode pour le calcul de la forme normale de Frobenius d'une matrice creuse dans $\mathbb{Z}$. Elle suppose donnés le polynôme minimal sur $\mathbb{Z}$ et la forme normale de Frobenius sur un corps fini $\mathbb{Z}_p$. L'idée est de calculer la partie sans carré s du polynôme minimal dans $\mathbb{Z}$, et son image $\overline{s}$ dans $\mathbb{Z}_p$. Une base pgcd-libre est alors calculée en utilisant [5, §4.8] pour représenter ce polynôme et les facteurs invariants de la matrice. On applique alors une remontée de Hensel mutlifacteurs (voir par exemple [51, §15.5]) à cette base pour reconstruire le polynôme s dans $\mathbb{Z}$. Ceci est valable si chaque facteur invariant de A dans $\mathbb{Z}$ réduit modulo p égale le facteur invariant correspondant dans $\mathbb{Z}_p$. On en déduit alors les facteurs invariants dans $\mathbb{Z}$. Cette note technique n'étant pas publiée, on pourra aussi se référer à [78, §7.2] où l'algorithme est rappelé.

Nous appliquons cette idée au calcul du polynôme caractéristique directement, sans passer par le calcul des facteurs invariants. On utilise donc la base pgcd-libre de s avec $P^A_{\text{min}} \mod p$ et $P^A_{\text{car}} \mod p$. Ainsi, la condition de validité est plus faible, puisqu'il s'agit simplement de vérifier que l'image de P^A_{min} modulo p soit égale au polynôme minimal de A dans $\mathbb{Z}_p$.

Nous décrivons cette méthode dans l'algorithme 12.1.

Pour le calcul du polynôme minimal sur $\mathbb{Z}$, on utilise à nouveau l'algorithme probabiliste de Wiedemann combiné avec le Théorème des Restes Chinois [40, Theorem 3.3] pour un

Algorithme 12.1 : `Remontée multifacteur et base pgcd-libre`

Données : A, une matrice $n \times n$ dans $\mathbb{Z}$; P^A_{min}, son polynôme minimal dans $\mathbb{Z}$; p, un nombre premier ; $\overline{P^A_{\text{car}}}$, le polynôme carctéristique de A dans $\mathbb{Z}_p$; β, une borne sur les coefficients des facteurs de P^A_{car}
Résultat : P^A_{car}, le polynôme caractéristique de A dans $\mathbb{Z}$
début
Calculer s la partie sans carré de P^A_{min}
$\overline{s} = s \mod p$, $\overline{P^A_{\text{min}}} = P^A_{\text{min}} \mod p$
Calculer une base pgcd-libre $(\overline{g_1}, \ldots, \overline{g_\chi})$ de $(\overline{s}, \overline{P^A_{\text{min}}}, \overline{P^A_{\text{car}}})$, ainsi que les exposants $(f_1, \ldots, f_\chi)$ tels que $\overline{P^A_{\text{car}}} = \prod \overline{g_i}^{f_i}$
Faire la remontée multifacteurs de Hensel pour obtenir $(g_1, \ldots, g_\chi)$ tels que $s \equiv g_1 \ldots g_\chi \mod p^k$ et $(g_1, \ldots, g_\chi) \equiv (\bar{g}_1, \ldots, \bar{g}_\chi) \mod p$ où $p^k > \beta$
retourner $P^A_{car} = \prod g_i^{f_i}$
fin

coût de $\mathcal{O}(sdE(n))$ opérations binaires, où s est la taille en bits des coefficients du polynôme minimal et d son degré. Le polynôme caractéristique dans $\mathbb{Z}_p$ est calculé par l'algorithme 11.6 en $\mathcal{O}(n^{1,5}E(n)) + \mathcal{O}^\sim(n^{2,5})$ opérations dans $\mathbb{Z}_p$. La remontée de Hensel de la base pgcd-libre nécessite $\mathcal{O}^\sim(nk)$ (où $k \approx \log_p \beta$) opérations de mots avec l'arithmétique rapide des entiers et des polynômes [51, Theorem 15.18]. D'après le lemme 9.1, les coefficients du polynôme caractéristique et par conséquent ceux du polynôme minimal sont bornés par $s = \mathcal{O}(n(\log n + \log \|A\|))$. Comme d est borné par n, le coût dominant est celui du calcul du polynôme minimal, en $\mathcal{O}^\sim(n^2 \log \|A\| E(n))$.

En pratique, les matrices creuses ou structurées peuvent avoir des polynômes minimaux avec des coefficients de petite taille, ou de faible degré. C'est le cas par exemple des matrices d'homologie utilisées dans [40]. Dans ce cas, c'est le calcul du polynôme caractéristique dans $\mathbb{Z}_p$ qui peut devenir dominant, et notre approche améliore alors ce coût d'un facteur $\sqrt{n}$, en comparaison avec l'algorithme dense de calcul de la forme de Frobenius utilisé dans [105].

12.2 Mise en pratique

Comme nous l'avons expliqué dans la partie 9.2.2, la factorisation de P^A_{min} en facteurs irréductibles dans $\mathbb{Z}$ est trop coûteuse en théorie. C'est la raison pour laquelle nous avons utilisé dans l'algorithme précédent, la factoristion de Cantor-Zassenhaus dans un corps fini, puis une remontée de Hensel sur la base pgcd-libre uniquement. Mais les excellentes implémentations de l'algorithme de Hensel, comme celles de la bibliothèque `NTL` le rendent envisageable en pratique dans de très nombreuses situations. Ainsi, une démarche alternative consiste à décomposer le polynôme minimal en facteurs irréductibles dans $\mathbb{Z}$, puis à chercher les multiplicités de ces facteurs dans le polynôme caractéristique en appliquant les méthodes du chapitre 11 dans un corps fini $\mathbb{Z}_p$ où p est tiré aléatoirement. Cela revient à faire faire effectuer la remontée de Hensel de l'algorithme 12.1 par l'algorithme de factorisation de la bibliothèque `NTL`.

12.3 Application et expérimentations

Pour illustrer l'efficacité en pratique de cette méthode, nous l'avons appliquée à un problème de la théorie des graphes que nous rappelons dans la partie 12.3.1. Nous donnons ensuite les résultats expérimentaux dans la partie 12.3.2.

12.3.1 La conjecture des cubes symétriques cospectraux

Ce problème qui nous a été soumis par G. Royle est décrit dans [4] ainsi que sur sa page internet[1].

Nous rappelons d'abord quelques définitions :

Définition 12.1. Soit X un graphe à n sommets dont les ensembles de sommets et d'arêtes sont respectivement $V(X)$ et $E(X)$.

- On définit la $k^{\text{ième}}$ puissance symétrique $X^{\{k\}}$ de X, comme le graphe dont les sommets correspondent aux $\binom{n}{k}$ k-sous-ensembles de $V(X)$. Deux sommets sont reliés par une arête si et seulement si la différence symétrique des sous-ensembles correspondant appartient à $E(X)$.
- On définit le spectre du graphe X comme le spectre de sa matrice d'adjacence.
- X est un graphe fortement régulier de paramètres (n, k, a, c) si il n'est ni complet ni vide, et le nombre de voisins de deux sommets x et y est k, a ou c, selon que x et y sont égaux, adjacents ou distinct et non adjacents.

La définition de la $k^{\text{ième}}$ puissance symétrique permet de généraliser la notion de marche aléatoire dans un graphe où k jetons sont placés sur k sommets distinct du graphe. Une k-marche est une séquence alternée de sous-ensembles V_i de k sommets de $V(X)$ et d'arêtes $e_i \in E(X) : (V_0, e_1, V_1, e_2, \ldots, e_n, V_n)$, tels que la différence symétrique entre V_i et V_{i-1} est l'arête e_i. Ainsi une k-marche dans X est plus simplement vue comme une 1-marche dans $X^{\{k\}}$. Les puissances symétriques des graphes apparaissent dans l'étude des systèmes Hamiltoniens d'interaction en mécanique quantique [4, §12] : les superpositions d'états sont représentées par les k-sous-ensembles de $V(X)$. L'évolution d'un système quantique correspond à une k-marche dans X et peut être représentée par une 1-marche dans $X^{\{k\}}$.

Par ailleurs, les puissances symétriques de graphes ouvrent aussi des perspectives dans la résolution du problème d'isomorphisme de graphe.

Problème 12.1. *Trouver un algorithme polynomial déterminant si deux graphes sont isomorphes.*

Les propriétés spectrales d'un graphe ne suffisent pas à déterminer sa classe d'isomorphisme, mais celles de ses puissances symétriques contiennent plus d'information. Si l'on trouve un degré k tel que le spectre de la $k$ième puissance symétrique décrive la classe d'isomorphisme du graphe, alors on dispose d'un algorithme polynomial pour résoudre le problème de l'isomorphisme.

A l'inverse, si tel n'est pas le cas, cela signifie que pour tout k, il existe une infinité de paires de graphes X et Y non-isomorphes, telles que $X^{\{k\}}$ et $Y^{\{k\}}$ sont cospectraux.

[1] http ://www.csse.uwa.edu.au/ gordon/sympower.html

L'équipe de Audenaert, Godsil, Royle et Rudolph ont abordé cette question en testant cette hypothèse sur une classe de graphes non isomorphes mais partageant de nombreuse propriétés spectrales : les graphes fortement réguliers. Dans [4], ils montrent qu'il existe une infinité de paires de graphes ayant des carrés symétriques cospectraux, en montrant que toute paire de graphes fortement réguliers ayant mêmes paramètres sont non seulement cospectraux (résultat classique) mais ont des carrés symétriques cospectraux.

En revanche, pour $k = 3$, on s'attendrait à trouver une infinité de paires de graphes ayant des cubes symétriques cospectraux, mais les auteurs n'en ont trouvé aucune pour tous les graphes fortement réguliers ayant jusqu'à 29 sommets. Pour ce faire, ils ont calculé les spectres de chacun de ces graphes en comparant ceux des graphes ayant les mêmes paramètres. Le tableau 12.1 décrit le nombre de chacun de ces graphes et la dimension de la matrice d'adjacence de leur cube symétrique. A partir de 35 sommets, la dimension devenait trop importante, et le

Paramètres	Nombre de graphes	$\binom{n}{3}$
(5, 2, 0, 1)	1	10
(9, 4, 1, 2)	1	84
(10, 3, 0, 1)	1	120
(13, 6, 2, 3)	1	286
(15, 6, 1, 3)	1	455
(16, 5, 0, 2)	1	560
(16, 6, 2, 2)	2	560
(17, 8, 3, 4)	1	680
(21, 10, 3, 6)	1	1330
(21, 10, 5, 4)	1	1330
(25, 8, 3, 2)	1	2300
(25, 12, 5, 6)	15	2300
(26, 10, 3, 4)	10	2600
(27, 10, 1, 5)	1	2925
(28, 12, 6, 4)	4	3276
(29, 14, 6, 7)	41	3654
(35, 16, 6, 8)	3854	6545
(36, 10, 4, 2)	1	7140
(36, 14, 4, 6)	108	7140
(36, 14, 7, 4)	1	7140
(36, 15, 6, 6)	32548	7140

TAB. 12.1 – *Nombre de graphes fortement réguliers ayant moins de* 36 *sommets*

calcul trop lent pour effectuer les 3854 calculs avec les outils logiciels existants. C'est à ce niveau que nous avons pu mettre en œuvre nos algorithmes implémentés dans la bibliothèque LINBOX, comme nous le décrivons dans la partie 12.3.2. Ceux-ci nous ont permis de vérifier qu'aucune paire de graphe fortement régulier ayant jusqu'à 36 sommets n'ont des cubes symétriques cospéctraux. Cette constatation prend plus de force dans la mesure ou elle porte $36\,582$ graphes, au lieu des 72 étudiés jusque là.

12.3.2 Résultats expérimentaux

Banc d'essai pour les algorithmes creux

Les matrices d'adjacence des cubes symétriques de graphes fortement réguliers nous ont d'abord servi de banc d'essai pour expérimenter les algorithmes *boîte noire* présentés dans cette partie. Les matrices `EX1, EX3, EX5` sont les matrices d'adjacence des cubes symétriques de graphes de paramètres (16,6,2,2), (26,10,3,4) et (35,16,6,8). Leurs dimensions sont respectivement $560 = \binom{16}{3}, 2600 = \binom{26}{3}$ et $6545 = \binom{35}{3}$.

Nous avons implémenté les algorithmes en utilisant la bibliothèque LINBOX pour la manipulation des *boîtes noires* et les calculs d'algèbre linéaire. La factorisation des polynômes est effectuée par la bibliothèque `NTL`. Pour ces matrices particulières nous avons indiqué aux algorithmes de LINBOX que la matrice représentée est symétrique, permettant ainsi d'économiser la moitié des produits matrice-vecteur dans la plupart des algorithmes (rang, déterminant, polynôme minimal).

Matrix	EX1	EX2	EX3	EX4	EX5
n : dimension	560	560	2600	2600	6545
d : degré du polynôme minimal	54	103	1036	1552	2874
ω : # nombre d'éléments non nuls par ligne	15,6	15,6	27,6	27,6	45,2
`Polynôme minimal` sur $\mathbb{Z}$	0,11s	0,26s	117s	260s	5002s
Factorisation dans $\mathbb{Z}$	0,02s	0,07s	9,4	18,15	74,09s
Nullité et recherche combinatoire	**3,37s**	5,33s	**33,2s**	**30,15s**	**289s**
Total	**3,51s**	5,66s	**159,4s**	**308,11s**	**5366s**
Système logarithmique	3,46s	**4,31s**	64,0s	57,0s	647s
Total	3,59s	**4,64s**	190,4s	336,4s	5641s

TAB. 12.2 – *Temps de calculs des différents modules de l'algorithme boîte noire sur un P4-3,2GHz-1Go*

Nous donnons dans le tableau 12.2 les temps de calcul des différents modules décrits dans le chapitre 11. Pour chaque matrice, deux calculs différents sont comparés : ils ont en commun le calcul du polynôme minimal et sa factorisation dans $\mathbb{Z}$. Pour déterminer les multiplicités des facteurs irréductibles, l'un utilise l'algorithme 11.5 et l'autre l'algorithme 11.4. On constate

que la détermination des multiplicités peut être l'opération dominante, comme c'est le cas avec la matrice `EX1` dont le polynôme minimal est de faible degré. Selon la structure des facteurs irréductibles, la méthode des nullités combinée avec la recherche combinatoire peut être plus rapide que la résolution du système logarithmique. Ces remarques justifient l'approche de la partie 11.2 qui permet de tirer parti des avantages des deux méthodes en les combinant.

Nous comparons maintenant les performances de ces implémentations totalement *boîte noire* avec celles présentées dans la partie 9.3 où la variante `HPC-BN` avait introduit l'intérêt de l'approche *boîte noire* sur des matrices creuses. Le tableau 12.3 montre le gain apporté par les

Matrice	n	ω	`HPC-BN`	`nullité-rec-comb.`	`système-log.`
$A =$ 08blocks	300	1,9	0,32s	0,08s	**0,07s**
$A\,{}^tA$	300	2,95	0,81s	**0,12s**	**0,12s**
$B =$ ch5-5.b3	600	4	4,4s	**1,52s**	1,97s
$B\,{}^tB$	600	13	**2,15s**	3,96	7,48s
`TF12`	552	7,6	6,8s	**5,53s**	5,75s
`mk9b3`	1260	3	31,25s	**10,51s**	177s
`Tref500`	500	16,9	65,14s	**25,14s**	25,17s

TAB. 12.3 – *Comparaison entre `HPC-BN` et les algorithmes boîte noire sur un Athlon-1,8-GHz-2Go*

implémentations *boite noire* en comparaison avec `HPC-BN`, pourtant déjà la meilleure dans les tableaux 9.2 et 9.3. A nouveau, la structure des facteurs irréductibles du polynôme minimal de chaque matrice implique des comportements différents. Ainsi, l'algorithme `système-log` est parfois comparable à celui des nullités, voire plus rapide, mais il peut être aussi beaucoup plus lent comme sur la matrice $B\,{}^tB$ et surtout `mk9b3`.

Recherche des cubes symétriques cospectraux

Ces premières expérimentations ont laissé entrevoir que la recherche de cubes symétriques cospectraux parmi les graphes fortement réguliers de 35 ou 36 sommets était réalisable. Nous avons donc mis en œuvre un calcul parallèle permettant d'effectuer une recherche exhaustive parmi les 3854 graphes à 35 sommets. La démarche adoptée consistait d'abord à sélectionner les graphes remplissant une condition nécessaire pour la cospectralité : leur polynôme caractéristique évalué en une valeur aléatoire dans un corps fini doivent être égaux. Ainsi la première phase consistait à tirer un nombre premier p de 20 bits aléatoirement et une valeur $\lambda \in \mathbb{Z}_p$, et de calculer les $v_i = \det(\lambda I - A_i)$ pour toutes les matrices A_i considérées. Le parallélisme ne consiste alors qu'à distribuer chacun de ces calcul de déterminant sur les différents processeurs utilisés. Parmi les paires restantes, on peut à nouveau appliquer cette méthode avec une autre valeur pour λ. Ce n'est que sur les éventuels derniers candidats que l'on calculera le polynôme caractéristique afin de vérifier qu'ils ont les mêmes spectres.

Pour le calcul des déterminants, nous avons utilisé l'algorithme de Wiedemann [119] combiné avec un préconditionnement cyclique rendant les polynômes minimaux et caractéristiques

égaux (voir [12] à ce sujet). En testant si le degré du polynôme minimal calculé est égal à n, on a un certificat du résultat. Ainsi, l'algorithme est probabiliste du type Las-Vegas : le résultat retourné est toujours correct, mais plusieurs calculs peuvent être nécessaires. L'implémentation de l'algorithme de Wiedemann est spécialisée pour les matrices symétriques, ce qui permet d'économiser la moitié des produits matrice-vecteurs. De plus les matrices sont représentées par des boîtes noires dont le produit matrice-vecteur tient compte du fait que les coefficients sont dans l'ensemble $\{0, 1\}$.

Le test des 3854 matrices de paramètre (35,16,6,8) a été effectué en 131,35 heures CPU d'un Itanium2 cadencé à 1,5Ghz (soit 16h25 effectives sur les 8 processeurs). Pour toutes les matrices A_i, nous avons calculé $\det(A + 26\,2139I) \mod 1\,048\,573$. Parmi les 3854 résultats, 14 paires étaient identiques. Pour ces matrices, nous avons ensuite calculé $\det(A_i + 26\,2147I) \mod 1\,048\,571$ et aucune paire ne restait identique. Il n'a donc fallu que 3866 calculs de déterminants pour vérifier qu'aucun polynôme caractéristique n'était égal.

De la même façon, le test des $32\,548$ matrices de paramètres (36, 15, 6, 6) a été effectué en 588 heures : après un premier calcul des $\det(A + 1234547I) \mod 67\,108\,859$, il restait 12 paires identiques et le calcul de $\det(A + 1234543I) \mod 67\,108\,819$ les a toutes distinguées. Ainsi nous avons vérifié qu'il n'y a pas de graphes fortement réguliers ayant au plus 36 sommets, dont les cubes symétriques sont cospectraux.

Conclusion

Nous avons présenté dans ce mémoire la mise en œuvre du calcul efficace du polynôme caractéristique dans $\mathbb{Z}$.

Elle a d'abord consisté en l'introduction d'un ensemble de routines de base pour l'algèbre linéaire dense dans un corps fini. Nous avons proposé une démarche de conception basée sur une prise en compte d'optimisations tant algorithmiques que matérielles, pour un ensemble restreint de routines, ainsi qu'une utilisation intensive de réductions à ces routines. Cette structuration débouche sur des algorithmes adaptatifs utilisant des seuils de différentes natures dont la détermination précise est essentielle.

L'utilisation des routines de base du calcul numérique, couplée à une analyse fine des bornes sur les calculs intermédiaires des algorithmes, permet d'approcher sur des corps finis les performances des routines numériques. Par la réduction des constantes dans les complexités et l'utilisation de l'arithmétique matricielle rapide, elles peuvent même être dépassées.

Au delà de ses applications propres en calcul discret (théorie des graphes, cryptographie, etc), le calcul exact est aussi complémentaire au calcul approché multiprécision dans la résolution de certains problèmes numériques mal conditionnés dans la mesure où il permet l'utilisation d'une algèbre linéaire efficace.

Pour le calcul du polynôme caractéristique de matrices denses, nous avons comparé plusieurs algorithme de type Krylov sur des corps finis, tirant leur efficacité de ces routines de base. Grâce à une estimation fine de la taille des coefficients du polynôme caractéristique dans $\mathbb{Z}$, nous en avons déduit un premier algorithme déterministe. Les performances dépassent celles des meilleures implémentations disponibles actuellement. De plus l'approche probabiliste, certifiant une probabilité d'erreur négligeable en pratique, améliore ces performances, en particulier en tirant parti des spécificités des données.

Enfin, pour le calcul avec des matrices *boîte noire*, un algorithme adaptatif est proposé, combinant les avantages de différentes méthodes basées sur des calculs de polynômes minimaux, de rangs et de déterminants. En particulier, une méthode basée sur le calcul d'index est proposée et permet d'ores et déjà d'atteindre une grande efficacité en pratique. Par exemple, ces algorithmes ont été mis en application pour un problème de théorie des graphes, sur des classes de matrices qu'aucun logiciel ne pouvait manipuler jusque là. L'étude de la probabilité de réussite de cet algorithme *boîte noire* reste cependant à finir.

Les points essentiels que nous avons dégagés dans cette étude sont les suivants :

1. Le produit matriciel sur des corps finis, ainsi que les résolutions de systèmes triangulaires multiples, la décomposition `LQUP` et l'inversion matricielle peuvent être effectués avec des performances avoisinant celles des BLAS numériques et de LAPACK.
2. L'algorithme de Winograd peut être employé pour améliorer ces performances. Nous précisons le domaine de validité de son utilisation sans réduction modulaire.
3. Des réductions au produit matriciel permettent de faire bénéficier de multiples autres calculs (profil de rang, base du noyau, multiplications triangulaires, inversions, ...) de ces performances.
4. Il est réaliste d'envisager la résolution exacte de certains problèmes numériques mal conditionnés.
5. Dans un corps fini, l'algorithme `LU-Krylov`, basé sur une méthode de Krylov et des

opérations par blocs est plus efficace que l'algorithme de Keller-Gehrig par branchements.

6. La forme de Kalman d'un problème de contrôle d'un système différentiel linéaire peut être calculée en $\mathcal{O}(n^\omega \log n)$ opérations algébriques.
7. L'algorithme rapide de Keller-Gehrig est intéressant en pratique. Nous présentons une relaxation des conditions de généricité et une approche adaptative pour traiter les situations de blocage.
8. La taille du plus grand coefficient du polynôme caractéristique d'une matrice entière dépasse d'au plus $0.21163175\frac{n}{2}$ bits celle de son déterminant.
9. La factorisation du polynôme minimal entier par la remontée de Hensel, permet d'obtenir des algorithmes efficaces en pratique pour de nombreux cas.
10. Sous l'hypothèse de la généralisation des projections de blocs efficaces au cas des transformation de similitude, la complexité algébrique du calcul du polynôme caractéristique d'une matrice *boîte noire* est réduite à $\tilde{\mathcal{O}}(n^{2,1365})$.
11. Des algorithmes adaptatifs, pour la recherche des multiplicités des facteurs polynôme minimal, permettent d'atteindre une grande efficacité pour le calcul du polynôme caractéristique de matrice *boîte noire* en pratique.

En outre, ces travaux ont permis de dégager un certain nombre de questions qui constituent autant de perspectives.

1. La résolution exacte de problèmes numériques mal conditionnés est possible lorsque les coefficients du problème sont du même ordre de grandeur (par exemple entre 10^{-20} et 10^{20}). Dans le cas contraire, les coefficients entiers du problème exact pourraient être trop grands ce qui pénaliserait la méthode exacte. Il faudrait alors étudier des techniques de raffinement en séparant le problème par ordres de grandeur, comme le suggère Schatzman. De manière plus générale, il manque une étude approfondie de la relation entre le nombre de conditionnement et la complexité des méthodes exactes.
2. Nous avons pu esquisser des techniques pour relâcher la contrainte de généricité dans l'algorithme rapide de Keller-Gehrig. Supprimer ces contraintes est d'un intérêt multiple : au niveau pratique, on obtiendrait l'algorithme le plus efficace sur un corps fini, et au niveau théorique, on pourrait obtenir la complexité $\mathcal{O}(n^\omega)$ dans tous les cas, ce qui résoudrait le dernier problème canonique de réduction au produit matriciel, en complexité algébrique.
3. Les algorithmes probabilistes que nous avons présentés reposent sur un calcul probabiliste de type *Monte Carlo* du polynôme minimal, puis sur le calcul de multiplicités. L'information apportée par ces dernières permet de détecter de nombreux cas d'échec de l'algorithme de calcul du polynôme minimal. Si l'on parvenait à détecter tous ces cas, on pourrait alors certifier le polynôme minimal grâce au calcul du polynôme caractéristique et l'algorithme serait alors du type *Las Vegas*.
4. La généralisation de l'utilisation des projections de blocs efficaces au cas des transformation de similitude permettrait de réduire l'exposant de la complexité algébrique du calcul du polynôme caractéristique de matrices *boîte noire*.

5. Dans nos algorithmes *boîte noire*, nous avons utilisé la remontée de Hensel pour la factorisation en polynômes irréductibles du polynôme minimal, pour des raisons de simplicité de mise en œuvre. Il est pourtant préférable de cantonner cette remontée à une base pgcd-libre décrivant le polynôme minimal et le polynôme caractéristique, comme l'a montré Storjohann. Cette idée doit être validée en pratique et doit permettre d'améliorer encore l'efficacité des implémentations.
6. La méthode du système logarithmique pour le calcul des multiplicités, permet d'atteindre à la fois une complexité intéressante, et une efficacité en pratique d'autant plus grande qu'elle peut être combinée avec d'autres méthodes complémentaires. Il reste à compléter l'analyse de sa probabilité de réussite.

BIBLIOGRAPHIE

[1] ABDELJAOUED, J., ET MALASCHONOK, G. I. Efficient algorithms for computing the characteristic polynomial in a domain. *Journal of Pure and Applied Algebra 156* (2001), 127–145. 108

[2] AHO, A. V., HOPCROFT, J. E., ET ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. 27, 74

[3] ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., ET SORENSEN, D. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999. 78

[4] AUDENART, K., GODSIL, C., ROYLE, G., ET RUDOLPH, T. Symmetric squares of graphs. Rapport Technique math.CO/0507251, arXiv, Juil. 2005. 179, 180

[5] BACH, E., ET SHALLIT, J. *Algorithmic Number Theory — Efficient Algorithms*, vol. I. MIT Press, Cambridge, USA, 1996. ISBN : 0-262-02405-5. 177

[6] BERKOWITZ, S. J. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters 18*, 3 (1984), 147–150. 107, 108

[7] BINI, D., CAPOVANI, M., LOTTI, G., ET ROMANI, F. $O(n^{2.7799})$ complexity for matrix multiplication. *Information Processing Letters 8* (1979), 234–235. 33

[8] BINI, D., ET LOTTI, G. Stability of fast algorithms for matrix multiplication. *Numerische Mathematik 36*, 1 (march 1980), 63–72. 41

[9] BINI, D., ET PAN, V. *Polynomial and Matrix Computations, Volume 1 : Fundamental Algorithms.* Birkhauser, Boston, 1994. 76

[10] BUNCH, J. R., ET HOPCROFT, J. E. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation 28* (1974), 231–236. 34, 61, 74

[11] CHAR, B. W., GEDDES, K. O., GONNET, G. H., LEONG, B., MONAGAN, M. B., ET WATT, S. M. *First Leaves : A Tutorial Introduction to Maple V.* Springer Verlag, New York, 1992. 22

[12] CHEN, L., EBERLY, W., KALTOFEN, E., SAUNDERS, B. D., TURNER, W. J., ET VILLARD, G. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and its Applications 343-344* (2002), 119–146. 21, 183

[13] CLAUSEN, M., BÜRGISSER, P., ET SHOKROLLAHI, M. A. *Algebraic Complexity Theory*. Springer, 1997. 25, 34, 84

[14] COHN, H., ET UMANS, C. A group-theoretic approach to fast matrix multiplication. Dans *Proceedings of the 44th Annual Symposium on Fondations of Computer Science (FOCS)* (oct 2003), pp. 438–449. 33

[15] COPPERSMITH, D. Solving homogeneous linear equations over GF[2] via block Wiedemann algorithm. *Mathematics of Computation 62*, 205 (Jan. 1994), 333–350. 108, 155

[16] COPPERSMITH, D. Rectangular matrix multiplication revisited. *Journal of Symbolic Computation 13* (1997), 42–49. 160

[17] COPPERSMITH, D., ET WINOGRAD, S. Matrix multiplication via arithmetic progression. Dans *Proceedings of the 19th ACM STOC* (1987), pp. 1–6. 33

[18] COPPERSMITH, D., ET WINOGRAD, S. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation 9*, 3 (1990), 251–280. 33

[19] COURRIEU, P. Fast computation of Moore-Penrose inverse matrices. *Neural Information Processing - Letters and Reviews 8*, 2 (Août 2005), 25–29. 94

[20] CSANKY, L. Fast parallel inversion algorithms. *SIAM Journal on Computing 5*, 4 (1976), 618–623. 107

[21] CUNG, V.-D., DANJEAN, V., DUMAS, J.-G., GAUTIER, T., HUARD, G., RAFFIN, B., RAPINE, C., ROCH, J.-L., ET TRYSTRAM, D. Adaptive and hybrid algorithms : classification and illustration on triangular system solving. Dans Dumas [32]. 171

[22] D., F., ET I., S. *Collected Problems in Higher Algebra, Problem n°979*. 1949. 107

[23] DANILEVSKIĬ, A. Sur la solution numérique de l'équation caractéristique (en russe). *Math. Sbornik 44*, 2 (1937), 169–172. 107, 127

[24] DEMMEL, J., HIDA, Y., KAHAN, W., XIAOYE, S. L., MUKHERJEE, S., ET RIEDY, E. J. Error bounds from extra precise iterative refinement. LAPACK working note 165. 100

[25] DEMMEL, J. W. The probability that a numerical analysis problem is difficult. *Mathematics of Computation 50*, 182 (Avr. 1988), 449–480. 100

[26] DIXON, J. D. Exact solution of linear equations using p-adic expansions. *Numerische Mathematik 40* (1982), 137–141. 17, 101

[27] DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., ET DUFF, I. A set of level 3 Basic Linear Algebra Subprograms. *Transactions on Mathematical Software 16*, 1 (Mar. 1990), 1–17. 33, 40

[28] DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., ET HANSON, R. J. An extended set of fortran basic linear algebra subprograms. *Transaction on Mathematical Software 14* (1988), 1–17. 33, 40

[29] DOUGLAS, C. C., HEROUX, M., SLISHMAN, G., ET SMITH, R. M. GEMMW : A portable level 3 BLAS winograd variant of Strassen's matrix-matrix multiply algorithm. *Journal of Computational Physics 110* (1994), 1–10. 41, 43

[30] DUMAS, J.-G. *Algorithmes parallèles efficaces pour le calcul formel : algèbre linéaire creuse et extensions algébriques*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, France, Dec. 2000. `ftp://ftp.imag.fr/pub/Mediatheque.IMAG/theses/2000/Dumas.Jean-Guillaume`. 21

[31] DUMAS, J.-G. Efficient dot product over finite fields. Dans *Proceedings of the seventh International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine*

(Juil. 2004), V. G. Ganzha, E. W. Mayr, et E. V. Vorozhtsov, Eds., Technische Universität München, Germany, pp. 139–154. 35, 36, 40, 170

[32] DUMAS, J.-G., Ed. *TC'2006. Proceedings of Transgressive Computing 2006, Granada, España* (Avr. 2006). 192, 193

[33] DUMAS, J.-G., EBERLY, W., GAUTIER, T., GIESBRECHT, M., GIORGI, P., HOVINEN, B., KALTOFEN, E., LOBO, A., PERNET, C., SAUNDERS, B. D., TURNER, W. J., VILLARD, G., ET WAN, Z. LinBox-1.0 : Exact computational linear algebra, Juil. 2005. `http://linalg.org`. 22

[34] DUMAS, J.-G., GAUTIER, T., ET PERNET, C. Finite field linear algebra subroutines. Dans Mora [87], pp. 63–74. 39, 46, 58

[35] DUMAS, J.-G., GIORGI, P., ET PERNET, C. FFPACK : Finite field linear algebra package. Dans *ISSAC'2004* (Juil. 2004), J. Gutierrez, Ed., ACM Press, New York, pp. 119–126. 61, 78, 82

[36] DUMAS, J.-G., PERNET, C., ET WAN, Z. Efficient computation of the characteristic polynomial. Dans Kauers [80], pp. 140–147. 108

[37] DUMAS, J.-G., ET ROCH, J.-L. On parallel block algorithms for exact triangularizations. *Parallel Computing 28*, 11 (Nov. 2002), 1531–1548. 80

[38] DUMAS, J.-G., ET RONDEPIERRE, A. Algorithms for symbolic/numeric control of affine dynamical system. Dans Kauers [80], pp. 277–284. 121, 122

[39] DUMAS, J.-G., SAUNDERS, B. D., ET VILLARD, G. Integer Smith form via the Valence : experience with large sparse matrices from Homology. Dans Traverso [109], pp. 95–105. 21

[40] DUMAS, J.-G., SAUNDERS, B. D., ET VILLARD, G. On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computation 32*, 1/2 (Juil.–Août 2001), 71–99. 143, 146, 148, 149, 151, 152, 177, 178

[41] DUMAS, J.-G., ET URBANSKA, A. An introspective algorithm for the determinant. Dans Dumas [32], pp. 185–202. 161

[42] DUMAS, J.-G., ET VILLARD, G. Computing the rank of sparse matrices over finite fields. Dans *Proceedings of the fifth International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine* (Sept. 2002), V. G. Ganzha, E. W. Mayr, et E. V. Vorozhtsov, Eds., Technische Universität München, Germany, pp. 47–62. 165

[43] EBERLY, W. Asymptotically efficient algorithms for the Frobenius form. Rapport technique, Dpt. of Computer Science, U. of Calgary Canada, Mai 2000. Extended abstract in [44]. 157

[44] EBERLY, W. Black box frobenius decomposition over small fields. Dans Traverso [109]. 21, 157, 163, 193

[45] EBERLY, W., GIESBRECHT, M., GIORGI, P., STORJOHANN, A., ET VILLARD, G. Solving sparse integer linear systems. Dans *Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation, Genova, Italy* (Juil. 2006), J.-G. Dumas, Ed., ACM Press, New York, pp. 63–70. 142, 157, 161

[46] EBERLY, W., GIESBRECHT, M., ET VILLARD, G. Computing the determinant and Smith form of an integer matrix. Dans *Proceedings of The* 41^{st} *Annual IEEE Symposium on Foundations of Computer Science, Redondo Beach, California* (Nov. 2000). 21

[47] EDELMAN, A. On the distribution of a scaled condition number. *Mathematics of Computation 58*, 197 (1992), 185–190. 100

[48] FADDEEVA, V. *Computational Methods of Linear Algebra*. Dover, New York, 1959. 107

[49] FRAME, J. A simple recurrent formula for inverting a matrix (abstract). *Bulletin of American Mathematical Society 55* (1949), 1045. 107

[50] GANTMACHER, F. R. *The Theory of Matrices*. Chelsea, New York, 1959. 25, 27, 107, 110, 164, 166

[51] VON ZUR GATHEN, J., ET GERHARD, J. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999. 25, 34, 41, 143, 144, 145, 149, 177, 178

[52] GAUTIER, T., VILLARD, G., ROCH, J.-L., DUMAS, J.-G., ET GIORGI, P. Givaro, a C++ library for computer algebra : exact arithmetic and data structures. Software, ciel-00000022, Oct. 2005. `www-lmc.imag.fr/Logiciels/givaro`. 22

[53] GIESBRECHT, M. Fast algorithms for rational forms of integer matrices. Dans *ISSAC'94 Oxford UK* (Juil. 1994), ACM press, New York, pp. 305–311. 143

[54] GIESBRECHT, M. Nearly optimal algorithms for canonical matrix form. *SIAM Journal on Computing 24*, 5 (1995), 948–969. 107

[55] GIESBRECHT, M., ET STORJOHANN, A. Computing rational forms of integer matrices. *J. Symb. Comput. 34*, 3 (2002), 157–172. 143, 145

[56] GIORGI, P. *Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque* LINBOX. Thèse de Doctorat, École normale supérieure de Lyon, Dec. 2004. 23, 37, 61

[57] GOLUB, G. H., ET VAN LOAN, C. F. *Matrix computations*, third ed. Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, MD, USA, 1996. 90, 93, 94

[58] GOTO, K., ET VAN DE GEIJN, R. On reducing TLB misses in matrix multiplication. Rapport technique, University of Texas, 2002. FLAME working note #9. 23

[59] GRABMEIER, J., KALTOFEN, E., ET WEISPFENNING, V., Eds. *Computer Algebra Handbook*. Springer Verlag, Heidelberg, Germany, 2002. 25

[60] GRANLUND, T. The GNU multiple precision arithmetic library, 2006. Version 4.2.1, `www.swox.com/gmp/manual`. 22

[61] GUSTAVSON, F., HENRIKSSON, A., JONSSON, I., ET KAAGSTROEM, B. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Lecture Notes in Computer Science 1541* (1998), 195–206. 55, 80

[62] HANROT, G., ET ZIMMERMANN, P. Seventh conference on real numbers and computers. `http://rnc7.loria.fr`. 102

[63] HIGHAM, N. J. Exploiting fast matrix multiplication within the level 3 BLAS. *Trans. on Mathematical Software 16*, 4 (Dec. 1990), 352–368. 40, 41

[64] HIGHAM, N. J. Stability of block alorithms with ast level 3 BLAS. *Trans. on Mathematical Software 18*, 3 (1992), 274–291. 41

[65] HIGHAM, N. J. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996. 99

[66] HOUSEHOLDER, A. *The Theory of Matrices in Numerical Analysis*. Blaisdell, Waltham, Mass., 1964. 107, 110, 127

[67] HUANG, X., ET PAN, V. Y. Fast rectangular matrix multiplications and improving parallel matrix computations. Dans *PASCO '97. Proceedings of the second international symposium on parallel symbolic computation, July 20–22, 1997, Maui, HI* (New York, NY 10036, USA, 1997), ACM, Ed., ACM Press, pp. 11–23. 64

[68] HUANG, X., ET PAN, V. Y. Fast rectangular matrix multiplication and applications. *J. Complexity 14*, 2 (Jun 1998), 257–299. 61

[69] HUSS-LEDERMAN, S., JACOBSON, E. M., JOHNSON, J. R., TSAO, A., ET TURNBULL, T. Implementation of Strassen's algorithm for matrix multiplication. Dans *Supercomputing '96 Conference Proceedings : November 17–22, Pittsburgh, PA* (1996), ACM, Ed., ACM Press and IEEE Computer Society Press. `www.supercomp.org/sc96/proceedings/SC96PROC/JACOBSON/`. 40, 42

[70] HUSS-LEDERMAN, S., JACOBSON, E. M., JOHNSON, J. R., TSAO, A., ET TURNBULL, T. Strassen's algorithm for matrix multiplication : Modeling analysis, and implementation. Rapport technique, Center for Computing Sciences, Nov. 1996. CCS-TR-96-17. 40, 41, 42, 45

[71] IBARRA, O. H., MORAN, S., ET HUI, R. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms 3*, 1 (Mar. 1982), 45–56. 76, 77, 110

[72] KALMAN, R. Canonical structure of linear dynamical systems. Dans *Proceedings of the National Academy of Sciences* (1961), pp. 596–600. 121

[73] KALTOFEN, E. On computing determinants of matrices without divisions. Dans *ISSAC'92* (Juil. 1992), P. S. Wang, Ed., ACM Press, New York. 108

[74] KALTOFEN, E. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation 64*, 210 (Avr. 1995), 777–806. 108, 155

[75] KALTOFEN, E. Challenges of symbolic computation : My favorite open problems. *Journal of Symbolic Computation 29*, 6 (Juin 2000), 891–919. 155

[76] KALTOFEN, E., ET SAUNDERS, B. D. On Wiedemann's method of solving sparse linear systems. Dans *Applied Algebra, Algebraic Algorithms and Error–Correcting Codes (AAECC '91)* (Oct. 1991), vol. 539 de *LNCS*, pp. 29–38. 19, 21, 155

[77] KALTOFEN, E., ET TRAGER, B. M. Computing with polynomials given by black boxes for their evaluations : greatest common divisors, factorization, separation of numerators and denominators. *Journal of Symbolic Computation 9*, 3 (1990), 301–320. 21

[78] KALTOFEN, E., ET VILLARD, G. On the complexity of computing determinants. *Computational Complexity 13* (2004), 91–130. 34, 108, 143, 149, 160, 177

[79] KAPORIN, I. The aggregation and cancellation techniques as a practical tool for faster matrix multiplication. *Theoretical Computer Science 315*, 2-3 (2004), 469–510. 41

[80] KAUERS, M., Ed. *ISSAC'2005. Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation, Beijing, China* (Juil. 2005), ACM Press, New York. 193

[81] KELLER-GEHRIG, W. Fast algorithms for the characteristic polynomial. *Theoretical computer science 36* (1985), 309–317. 76, 84, 107, 116, 117, 127, 128, 136

[82] LADERMAN, J., PAN, V., ET SHA, X.-H. On practical algorithms for accelerated matrix multiplication. *Linear Algebra Appl. 162–164* (1992), 557–588. 41

[83] LAWSON, C. L., HANSON, R. J., KINCAID, D., ET KROGH, F. T. Basic linear algebra subprograms for FORTRAN usage. *Transaction on Mathematical Software 5* (1979), 308–323. 33, 40

[84] LOMBARDI, H., ET ABDELJAOUED, J. *Méthodes matricielles - Introduction à la complexité algébrique*. Berlin, Heidelberg, New-York : Springer, 2004. 25, 107, 110

[85] MONAGAN, M. B., GEDDES, K. O., HEAL, K. M., LABAHN, G., ET VORKOETTER, S. M. *Maple V Programming Guide*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1998. With the assistance of J. S. Devitt, M. L. Hansen, D. Redfern, and K. M. Rickard. 22

[86] MONTGOMERY, P. L. Modular multiplication without trial division. *Mathematics of Computation 44*, 170 (Avr. 1985), 519–521. 36

[87] MORA, T., Ed. *ISSAC'2002. Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France* (Juil. 2002), ACM Press, New York. 193, 197

[88] MÜLLER, N. T. iRRAM - exact arithmetic in C++, 2005. Version 2005_03 `www.informatik.uni-trier.de/iRRAM/`. 101, 103

[89] NOBLE, B. A method for computing the generalized inverse of a matrix. *SIAM Journal on Numerical Analysis 3*, 4 (Dec. 1966), 582–584. 93

[90] O'GORMAN, T. J., ROSS, J. M., TABER, A. H., ZIEGLER, J. F., MUHLFELD, H. P., MONTROSE, C. J., CURTIS, H. W., ET WALSH, J. L. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development 40*, 1 (Jan. 1996), 41–50. 150

[91] PERNET, C. Implementation of Winograd's fast matrix multiplication over finite fields using ATLAS level 3 BLAS. Rapport technique, Laboratoire Informatique et Distribution, Juil. 2001. `www-id.imag.fr/Apache/RR/RR011122FFLAS.ps.gz`. 39

[92] PERNET, C. Calcul du polynôme caractéristique sur des corps finis. Mémoire de D.E.A., Université Joseph Fourrier, Juin 2003. `www-lmc.imag.fr/lmc-mosaic/Clement.Pernet/`. 108

[93] PERNET, C., ET WAN, Z. L U based algorithms for characteristic polynomial over a finite field. *SIGSAM Bull. 37*, 3 (2003), 83–84. Poster available at `www-lmc.imag.fr/lmc-mosaic/Clement.Pernet`. 108

[94] PREPARATA, F., ET SARWATE, D. An improved parallel processor bound in fast matrix inversion. *Information Processing Letters 7*, 3 (1978), 148–150. 107

[95] ROBBINS, H. A remark on Stirling's formula. *The American Mathematical Monthly 62*, 1 (Jan. 1955), 26–29. 144

[96] SAMUELSON, P. A. A method of determining explicitly the coefficients of the characteristic equation. *Annals of Mathematical Statistics 13*, 4 (dec 1942), 424–429. 107

[97] SAUNDERS, B. D. Black box methods for least squares problems. Dans *ISSAC 2001 : July 22–25, 2001, University of Western Ontario, London, Ontario, Canada : proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation* (2001), B. Mourrain, Ed., pp. 297–302. 92

[98] SCHATZMAN, M. Les bonnes matrices sont rares. *MATAPLI, bulletin de la SMAI 78* (Oct. 2005), 41–56. `http://atlas.mat.ub.es/personals/sombra/cours/uba/programas/schatz05.pdf`. 100

[99] SCHÖNAGE, A. Partial and total matrix multiplication. *SIAM Journal of Complexity 10* (1981), 434–455. 33

[100] SHOUP, V. NTL 5.4 : A library for doing number theory, 2006. `www.shoup.net/ntl`. 22, 36, 148, 149

[101] SMITH, H. J. S. On systems of linear indeterminate equations and congruences. *Philosophical Transactions of the Royal Society 151* (1861), 293–326. 27

[102] SOURIAU, J.-M. Une méthode pour la décomposition spectrale et l'inversion des matrices. *Comptes-Rendus de l'Académie des Sciences 227* (1948), 1010–1011. 107

[103] STEEL, A. Personal communication, 2005. 151

[104] STORJOHANN, A. *Algorithms for Matrix Canonical Forms*. Thèse de Doctorat, Institut für Wissenschaftliches Rechnen, ETH-Zentrum, Zürich, Switzerland, Nov. 2000. 83, 84, 113, 116

[105] STORJOHANN, A. Computing the Frobenius form of a sparse integer matrix. Manuscrit non publié, Avr. 2000. 143, 148, 149, 152, 177, 178

[106] STORJOHANN, A. High order lifting (extended abstract). Dans Mora [87], pp. 246–254. 197

[107] STORJOHANN, A. High–order lifting and integrality certification. *Journal of Symbolic Computation 36*, 3–4 (2003), 613–648. Extended abstract in [106]. 160

[108] STRASSEN, V. Gaussian elimination is not optimal. *Numerische Mathematik 13* (1969), 354–356. 18, 33, 34, 41

[109] TRAVERSO, C., Ed. *ISSAC'2000. Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation, Saint Andrews, Scotland* (Août 2000), ACM Press, New York. 193

[110] TURNER, W. J. *Blackbox linear algebra with the LinBox library*. Thèse de Doctorat, North Carolina State University, Mai 2002. 170

[111] VERRIER, U. L. Sur les variations séculaires des éléments elliptiques des sept plantètes principales. *Journal des Mathématiques Pures et Appliquées 5* (1840), 220–254. 107

[112] VILLARD, G. *Algorithmique en algèbre linéaire exacte*. Mémoire d'habilitation à diriger des recherches, Université Claude Bernard, Lyon I, mars 2003. 27, 155, 157, 160

[113] VILLARD, G. Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. Dans *ISSAC'97* (Juil. 1997), W. W. Küchlin, Ed., ACM Press, New York, pp. 32–39. 108, 155

[114] VILLARD, G. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Rapport Technique 975–IM, LMC/IMAG, Avr. 1997. 108

[115] VILLARD, G. Computing the Frobenius normal form of a sparse matrix. Dans *CASC'00* (Oct. 2000), V. G. Ganzha, E. W. Mayr, et E. V. Vorozhtsov, Eds. 21, 157, 158, 163, 171, 173, 174

[116] VILLARD, G. Personal communication, 2005. 121

[117] VILLARD, G. Personal communication, 2006. 161

[118] WHALEY, R. C., PETITET, A., ET DONGARRA, J. J. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing 27*, 1–2 (Jan. 2001), 3–35. 23, 40

[119] WIEDEMANN, D. H. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory 32*, 1 (Jan. 1986), 54–62. 18, 26, 111, 155, 182

[120] WILKINSON, J. *The Algebraic Eigenvalue Problem*. Oxford, Clarendon press, 1965. 99

[121] WOLFRAM, S. *The Mathematica book*, quatrième ed. Cambridge University Press and Wolfram Research, Inc., New York, NY, USA and 100 Trade Center Drive, Champaign, IL 61820-7237, USA, 1999. 22

Résumé : L'algèbre linéaire est une brique de base essentielle du calcul scientifique. Initialement dominée par le calcul numérique, elle connaît depuis les dix dernières années des progrès considérables en calcul exact. Ces avancées algorithmiques rendant l'approche exacte envisageable, il est devenu nécessaire de considérer leur mise en pratique. Nous présentons la mise en œuvre de routines de base en algèbre linéaire exacte dont l'efficacité sur les corps finis est comparable à celles des BLAS numériques. Au délà des applications propres au calcul exact, nous montrons qu'elles ouvrent une alternative au calcul numérique multiprécision pour la résolution de certains problèmes mal conditionnés.

Le calcul du polynôme caractéristique est l'un des problèmes classiques d'algèbre linéaire. Son calcul exact permet par exemple de déterminer la similitude entre deux matrices, par le calcul de la forme normale de Frobenius, ou la cospectralité de deux graphes. Si l'amélioration de sa complexité théorique reste un problème ouvert, tant pour les méthodes denses que *boîte noire*, nous abordons la question du point de vue de la praticabilité : des algorithmes adaptatifs pour les matrices denses ou *boîte noire* sont dérivés des meilleurs algorithmes existants pour assurer l'efficacité en pratique. Cela permet de traiter de façon exacte des problèmes de dimensions jusqu'alors inaccessibles.

Mots clés : Calcul exact, Algèbre linéaire, BLAS, Polynôme caractéristique, Algorithmes de Keller-Gehrig, Matrice denses, Matrices boîte noire.

Title : Efficient exact linear algebra : computing the characteristic polynomial

Abstract: Linear algebra is a building block in scientific computation. Initially dominated by the numerical computation, it has been the scene of major breakthrough in exact computation during the last decade. These algorithmic progresses making the exact computation approach feasible, it became necessary to consider these algorithms from the viewpoint of practicability. We present the building of a set of basic exact linear algebra subroutines. Their efficiency over a finite field near the numerical BLAS. Beyond the applications in exact computation, we show that they offer an alternative to the multiprecision numerical methods for the resolution of ill-conditioned problems.

The computation of the characteristic polynomial is part of the classic problem in linear algebra. Its exact computation, e.g. helps determine the similarity equivalence between two matrices, using the Frobenius normal form, or the cospectrality of two graphs. The improvement of its theoretical complexity remains an open problem, for both dense or black-box methods. We address this problem from the viewpoint of efficiency in practice: adaptive algorithms for dense or black-box matrices are derived from the best existing algorithms, to ensure high efficiency in practice. It makes it possible to handle problems whose dimensions was up to now unreachable.

Keywords: Exact computation, Linear algebra, BLAS, Characteristic polynomial, Keller-Gehrig algorithms, Dense matrices, Black Box.