Algorithmes parallèles efficaces pour le calcul formel : algèbre linéaire creuse et extensions algébriques

Jean-Guillaume Dumas

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE

pour obtenir le titre de

DOCTEUR DE L'INPG

***Spécialité :* «MATHÉMATIQUES APPLIQUÉES»**

préparée au **LABORATOIRE INFORMATIQUE ET DISTRIBUTION**

dans le cadre de l'**ÉCOLE DOCTORALE «MATHÉMATIQUES ET INFORMATIQUE»**

présentée et soutenue publiquement

par

JEAN-GUILLAUME DUMAS

le 20 décembre 2000

ALGORITHMES PARALLÈLES EFFICACES POUR LE CALCUL FORMEL : ALGÈBRE LINÉAIRE CREUSE ET EXTENSIONS ALGÉBRIQUES

DIRECTRICE DE THÈSE

Mme. Brigitte PLATEAU

JURY

M. Jean DELLA DORA,	Président
M. Mark GIESBRECHT,	Rapporteur
M. Paul ZIMMERMANN,	Rapporteur
Mme. Brigitte PLATEAU,	Directrice de thèse
M. Thierry GAUTIER,	Responsable de thèse
M. David SAUNDERS,	Examinateur
M. Gilles VILLARD,	Examinateur

TABLE DES MATIÈRES

TABLE DES FIGURES

TABLE DES TABLEAUX

TABLE DES ALGORITHMES

TABLE DES CODES

INTRODUCTION

1

THÈSE

« Cataleptique
Élan cyclopéen
Zone inhospitalière
S'en sortira demain
Qu'il est loin le soleil »

Bertrand Cantat

Le mot le plus important, peut-être, du titre de cette thèse est le mot *efficace* : il a deux aspects, un aspect temporel, il faut aller le plus vite possible, et un aspect spatial, il faut consommer le moins possible de mémoire. Ces déclarations d'intention peuvent sembler ambitieuses, mais c'est en gardant cette idée à l'esprit que nous avons développé et mené des expériences sur différents algorithmes de calcul formel.

Si le calcul numérique s'attache à résoudre des problèmes scientifiques en calculant des solutions approchées, le calcul formel développe des réponses exactes.

Seulement, les architectures des processeurs actuels sont construites pour le calcul numérique ; et même si une machine possède, en général, deux types de représentations pour les nombres, les nombres à virgule flottante et les nombres *entiers*, ceux-ci n'en restent pas moins des outils d'approximation. En effet, les nombres flottants, d'une part, sont les chiffres significatifs, en base 2, d'un nombre décimal alors que les nombres entiers, d'autre part, sont les chiffres les moins significatifs, généralement dans l'écriture binaire, d'un entier relatif. Néanmoins, il est possible, à l'aide de ces outils, de construire des représentations exactes, pour les entiers ou les décimaux au moins, les réels étant en général transcendants. Par exemple, il est possible d'obtenir une précision, a priori *infinie*, en représentant un entier relatif par une liste d'entiers machine ; une autre idée est d'adopter une représentation modulaire, où un entier est représenté par ses restes modulo un ensemble d'entiers plus petits.

Depuis quelques années, l'extension de l'utilisation de l'informatique dans tous les domaines de recherche scientifique et technique se traduit par un besoin croissant de puissance de calcul. L'augmentation de la vitesse des microprocesseurs ainsi que de leur capacité mémoire n'est plus suffisante pour couvrir ces besoins. La mise en commun des ressources de plusieurs processeurs, dans une même machine, ou à l'aide de réseaux à haut débit, est à même de fournir l'aleph nécessaire. Il est donc vital d'employer ces ressources en *parallèle*, et de bien les employer ; en particulier, il s'agit de réduire les communications entre processeurs pour favoriser le calcul, et de réagir dynamiquement aux différences de caractéristiques et de charge des processeurs.

En pratique, le problème principal que nous cherchons à résoudre est le calcul d'une forme canonique de très grandes matrices creuses à coefficients entiers, la forme normale de Smith, pour être à même de résoudre ou de mieux comprendre les systèmes d'*équations linéaires* qu'elles représentent. Par « très grandes », nous entendons un million d'inconnues et un million d'équations, c'est-à-dire mille milliards de termes. De tels systèmes sont même, en général, impossibles à stocker actuellement. Cependant, nous nous intéressons à des systèmes dans lesquels beaucoup de ces termes sont identiques et valent zéro ; on parle dans ce cas de système *creux*, par opposition à un système dense. Il suffit, alors, de stocker les autres variables. La limite actuelle sur le nombre de ces variables, permettant de calculer avec ces systèmes, semble être de l'ordre de quelques dizaines de millions d'éléments. Enfin, nous travaillons avec des nombres entiers. Cela induit un calcul exact mais implique deux problèmes principaux : premièrement, la taille des entiers n'est pas bornée a priori puisque l'ensemble des entiers est infini et, deuxièmement, les opérations entières sont limitées (la division n'est pas toujours définie par exemple). Pour résoudre ces problèmes, nous nous placerons, dans un

premier temps, dans une structure algébrique plus petite et autorisant toutes les opérations classiques, un *corps fini*. Un corps fini est simplement une extension d'un ensemble d'entiers modulo un nombre premier. La reconstruction de la solution entière à partir des solutions plus petites est ensuite relativement aisée.

Dans ce cadre, la thèse que nous défendons peut donc se résumer ainsi :
Associée à une arithmétique rapide et un parallélisme portable à ordonnancement dynamique, la résolution exacte de très grands systèmes linéaires creux peut être envisagée en pratique.

La première étape pour réaliser ce projet est donc d'avoir une implémentation efficace des corps finis. Dans la première partie, nous proposons une méthode pour construire et utiliser les corps finis (§3) et la comparons ensuite à différentes méthodes existantes (§4).

Ensuite, la deuxième partie s'attache à décrire différentes méthodes de calcul du rang (le nombre d'équations indépendantes) de matrices creuses à coefficients dans un corps fini. En effet, nous verrons que le calcul du rang est une brique essentielle au calcul de la forme de Smith. Le premier chapitre étudie des variantes de la méthode d'élimination de Gauß (§5). Le deuxième chapitre étudie des algorithmes itératifs pour calculer le rang : les méthodes de Krylov (§6). Le dernier chapitre de cette partie définit précisément des domaines privilégiés de ces deux types de méthodes (§7).

Dans la troisième partie de ce document, nous développons un nouvel algorithme fortement parallèle pour le calcul de la forme normale de Smith, adapté au cas des très grandes matrices creuses. Après un état des algorithmes actuels pour le calcul de cette forme canonique (§8), nous proposons un nouvel algorithme pour le calcul parallèle du polynôme minimal entier d'une matrice creuse symétrique (§9) puis son application au calcul de la forme normale de Smith (§10).

La dernière partie, enfin, justifie les choix des structures de données employées (§11) et décrit les modules, formés des algorithmes présentés, que nous avons implémentés pour différents logiciels (§12).

Pour conclure cette introduction, nous présentons le cadre logiciel du calcul formel dans lequel nous travaillons, Linbox, puis le modèle de programmation parallèle que nous avons choisi, Athapascan-1 (§2).

2

BOÎTE À OUTILS : BIBLIOTHÈQUES DE CALCUL FORMEL ET MODÈLE DE PROGRAMMATION PARALLÈLE

« Puisque ces mystères me dépassent, feignons d'en être l'organisateur »

Jean Cocteau

Sommaire

L'objectif de ce chapitre est double. D'une part, nous présentons plusieurs bibliothèques de calcul formel spécialisées dans différents domaines. Notre but est de pouvoir utiliser, pour chaque partie de nos algorithmes, la bibliothèque la plus performante sur ce point. Dans cette optique, le projet Linbox, commun au CNRS à Grenoble et à la National Science Foundation aux États Unis et au Canada, s'intéresse à la définition d'interfaces communes pour bibliothèques de calcul formel. Nous le présentons dans la prochaine section.

D'autre part, après une brève étude des langages de programmation parallèle, nous choisissons Athapascan-1 comme interface. Nous montrons dans la suite que ce modèle nous paraît le plus adapté, à la fois en terme d'expression du parallélisme pour les algorithmes irréguliers du calcul formel (l'ordonnancement est dynamique) et en terme de modèle de coût (dans le modèle d'Athapascan-1, le temps *et* l'espace mémoire sont bornés).

2.1 LINBOX

De nombreuses bibliothèques de calcul formel spécialisées existent actuellement. Par exemple, les bibliothèques GMP [75] pour l'arithmétique entière à précision arbitraire, ou encore NTL [153] pour l'arithmétique polynomiale, sont quasiment des standards. Ces bibliothèques sont écrites en C ou C++ pour des raisons d'efficacité. Java est une autre piste pour écrire les algorithmes les plus efficaces, mais ne semble pas encore compétitif [14 - Bernardin et al. (1999)]. À l'opposé des bibliothèques pointues, on trouve des systèmes complets de calcul tels que Maple [36, 114, 35], Mathematica [175], Axiom [173] ou, plus spécifiquement, GAP [67]. Ces plates-formes sont très répandues et intéressantes par leur simplicité d'utilisation et le vaste répertoire de leurs possibilités. Toutefois, les différents algorithmes qui composent leur noyau ne sont pas en général les meilleurs du moment. En outre, ces systèmes sont en général interprétés. Nous nous situons exactement au milieu de ces deux approches : nous voulons tirer parti des implémentations de base les plus performantes pour écrire les algorithmes spécifiques les plus performants, puis les compiler et les utiliser dans les systèmes de calculs les plus simples. De plus, ce niveau paraît adéquat pour l'introduction de parallélisme dans le cas d'algorithmes irréguliers : paralléliser à grain trop fin est inefficace dans notre cas, les communications étant alors largement prépondérantes. À ce niveau, en effet, les algorithmes de calcul formel manipulant des matrices, des polynômes, par exemple, possèdent un grain suffisant. Enfin, l'écriture d'algorithmes dans ce modèle se fait donc de manière générique pour

pouvoir inter-changer les bibliothèques, afin de tirer le mieux parti de leurs possibilités pour différents cas de figure. Non seulement les meilleurs algorithmes de base sont choisis et garantissent les meilleures performances à l'algorithme, mais, en outre, un même code peut alors être réutilisé dans tous les cas, induisant un confort de programmation majeur.

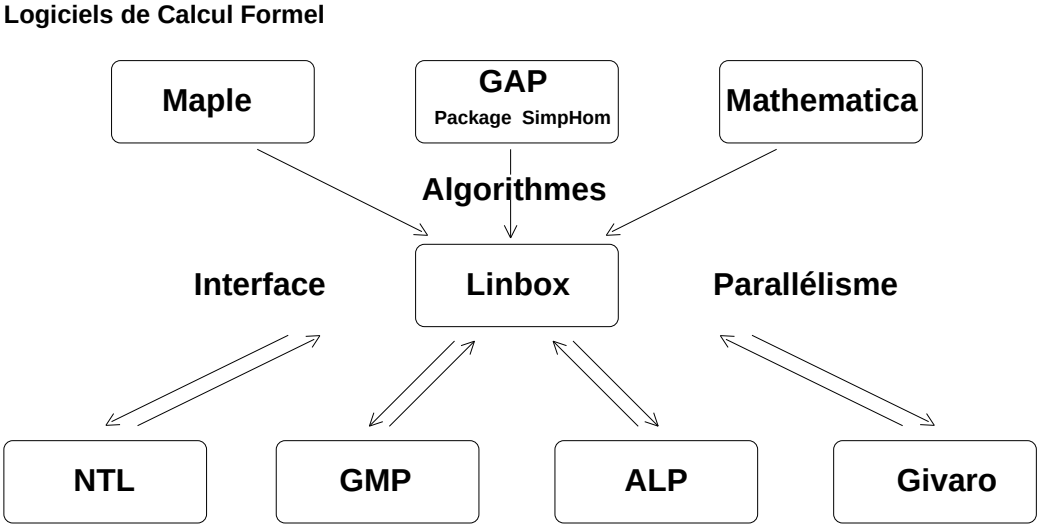

FIGURE 2.1 – Le projet Linbox

Le projet CNRS-NSF Linbox [108 - Linbox Group (2000)], regroupant l'Université de Calgary, le Washington College dans le Maryland, le Laboratoire de Modélisation et Calcul à Grenoble, le Laboratoire Informatique et Distribution à Grenoble, l'Université de l'Ontario de l'Ouest, l'Université du Delaware, et l'Université d'état de Caroline du Nord, s'inscrit dans ce cadre pour développer des algorithmes de calcul de polynômes minimaux, de formes normales de matrices, de résolution de systèmes linéaires et d'équations diophantiennes, par exemple. L'organigramme 2.1 représente ce cadre de travail.

Par ailleurs, il existe de nombreuses autres approches pour le calcul formel parallèle, J-L. Roch et G. Villard en font une étude exhaustive dans [144 - Roch et Villard (1997)]. Nous pensons que l'approche de Linbox est la plus efficace et la plus évolutive car elle est construite de manière générique pour pouvoir utiliser, puis changer, les bibliothèques spécialisées les plus performantes. C'est donc dans ce cadre que nous développons nos algorithmes pour la construction des corps finis, l'algèbre linéaire creuse et le calcul de formes normales de matrices.

Dans les sections suivantes, nous présentons brièvement les bibliothèques spécialisées que nous utilisons.

2.1.1 GMP

GNUmp, *GNU Multiprecision Package* [75 - Granlund (2000)], est une bibliothèque implémentant des nombres entiers signés, des nombres rationnels, et des nombres à virgule flottante en précision arbitraire. Toutes les fonctions ont une interface normalisée. GNUmp est conçu pour être aussi rapide que possible en utilisant les mots machine comme type arithmétique de base, en utilisant des algorithmes rapides, en optimisant soigneusement le code assembleur pour les boucles intérieures les plus communes, et par une attention générale sur la vitesse (par opposition à la simplicité ou à l'élégance). Nous utilisons principalement les fonctionnalités entières de GMP (mpz), à savoir les opérations de base (addition, multiplication, division) ainsi que le test de primalité probabiliste de Miller et Rabin [139 - Rabin (1980)].

2.1.2 NTL

NTL, *A Library for doing Number Theory* [153 - Shoup (2000)], est une bibliothèque C++ fournissant des structures de données et des algorithmes pour des nombres entiers signés de longueur arbitraire. Les structures implémentées sont des vecteurs, des matrices, et des polynômes sur $\mathbb{Z}$ ou sur des corps finis. L'arithmétique entière utilise GMP. Nous utilisons NTL pour comparer notre arithmétique sur les corps finis.

2.1.3 ALP

ALP, *Algèbre Linéaire pour les Polynômes* [116 - Mourrain et Prieto (2000)], est une bibliothèque de classes C++ pour le calcul scientifique et symbolique consacrée à l'algèbre linéaire et polynomiale. Elle contient des classes de vecteurs, de matrices, de monômes, de polynômes et d'algorithmes de résolution d'équations. Elle utilise plusieurs bibliothèques externes et spécialisées, comme GMP, MPFR, umfpack, Lapack, RS, MPSolve. Nous l'utilisons pour comparer notre arithmétique sur les corps finis. Des algorithmes de calcul de rang sont aussi implémentés dans ALP, mais seulement pour matrices denses ou numériques.

2.1.4 GIVARO

Givaro, [70 - Gautier (1996)], est une bibliothèque C++ pour le calcul formel. Elle utilise GMP pour définir une extension des entiers machine en précision arbitraire. Elle définit des structures vectorielles, matricielles et polynomiales ainsi qu'une structure de nombres algébriques. Elle est développée en commun aux laboratoires LMC et LID. Ainsi, nous utilisons ses fonctionnalités vectorielles et matricielles de base pour implémenter nos algorithmes.

Nous avons aussi développé un module de théorie des nombres pour nos calculs de construction des corps finis (voir chapitres 3 et 12) ainsi que des modules d'arithmétique sur les corps finis.

2.1.5 SPARSELIB++

SparseLib++, [137 - Pozo et al. (1996)], est une bibliothèque C++ pour les calculs creux efficaces. Le progiciel se compose des classes de matrices pour plusieurs formats de mémoire creux (par exemple ligne-compressé, colonne-compressé et formats de coordonnées) et fournit les fonctionnalités de base pour manipuler ces matrices creuses. Les BLAS [46 - Dongarra et al. (1990)] sont utilisés pour les exécutions numériques (par exemple le produit matrice-vecteur creux). Divers pré-conditionneurs numériques généralement utilisés dans les résolutions itératives de systèmes linéaires sont inclus dans la bibliothèque. Nous l'utilisons pour ses formats de matrices creuses.

2.1.6 LEDA

LEDA, *A Library of Efficient Data Types and Algorithms* [112 - Mehlhorn et Näher (1999)], est une bibliothèque C++ de structures de données et d'algorithmes de combinatoire. Elle fournit une collection considérable de structures de données et d'algorithmes sous une forme qui leur permet d'être employés par des non-experts. Dans la version en cours, cette collection inclut la plupart des structures et algorithmes classiques du domaine. LEDA contient des implantations efficaces pour chacun de ces types de données, par exemple, piles de Fibonacci pour des files d'attente prioritaires, tables dynamiques d'adressage dispersé parfait (*dynamic perfect hashing*) pour les dictionnaires... Un atout majeur de LEDA est son implémentation des graphes. Elle offre les itérations standards telles que "pour tous les nœuds v d'un graphe G" ou encore "pour tous les voisins W de v" ; elle

permet d'ajouter et d'effacer des sommets et des arêtes, d'en manipuler les matrices d'incidence, etc.

2.2 QUEL LANGAGE DE PROGRAMMATION PARALLÈLE ?

L'utilité des machines parallèles est aujourd'hui unanimement reconnue par la communauté scientifique. De plus, l'évolution rapide de la technologie en ce qui concerne l'architecture matérielle et les réseaux est telle que les machines symétriques multiprocesseurs*, tout comme les interconnexions « rapides » de stations de travail†, sont de plus en plus performantes et répandues.

Ainsi, la programmation de telles machines commence à devenir populaire, comme en témoigne la multiplicité des langages parallèles : parallélisation automatique dans certains langages de programmation logique (Prolog), annotations pour une extraction implicite du parallélisme (HPF [79], Jade [143]), créations de fils d'exécution concurrents dans les langages à base de processus légers (bibliothèque Pthreads [81], Java [129], OpenMP [43]), créations explicites de processus communicants par échanges de messages (PVM [160], MPI [117, 47], PM^2 [121], Athapascan-0 [23]), créations explicites de tâches concurrentes (Cilk [19], NESL [18], Athapascan-1 [32, 65, 49, 33]), réalisation d'une mémoire virtuelle partagée permettant une programmation parallèle des machines à mémoire distribuée (Linda [71, 30]), etc.

Dans cette section, nous tirons parti de l'étude de F. Galilée sur ces langages pour déterminer le plus à même de répondre à nos besoins.

Ces langages facilitent la description du parallélisme contenu dans l'application, en général par une transposition sous forme de graphe. Ensuite, l'obtention de performances résulte de l'ordonnancement (placement et date d'exécution) des tâches qui sera effectué ; certains de ces langages garantissent leurs performances tant temporelles que spatiales par un modèle de coût associé. Enfin, la portabilité de ces systèmes résulte de la capacité de l'ordonnancement à s'adapter aux conditions particulières de l'exécution, c'est-à-dire essentiellement à la machine hôte. F. Galilée a réalisé une étude exhaustive des langages parallèles de haut niveau dans sa thèse [65 - Galilée (1999), Chapitre 2]. Nous en reprenons le tableau ré-

* SMP, *Symmetric Multi-Processors*.

† "Réseaux" (*NOW, Network of Workstations*), ou "grappes" (*COW, Cluster of Workstations*) de stations de travail.

capitulatif.

Nous entendons par langages de « haut niveau » les langages offrant un ordonnancement automatique des tâches et dans lesquels les applications sont exprimées en ne faisant aucune hypothèse sur la machine sur laquelle aura lieu l'exécution. Nous remarquons que la modélisation de l'exécution d'une application par un graphe permet de définir la sémantique des accès aux données et d'associer un modèle de coût au langage. Afin de régler les accès concurrents à la mémoire, les tâches de calcul doivent être synchronisées. Ces synchronisations sont, soit **explicites**, comme par exemple pour les langages à base de processus légers ou pour Cilk, soit **implicites** et déterminées à partir de la déclaration par les tâches des accès à la mémoire qu'elles effectuent, comme dans Jade ou Athapascan-1. On parle alors de langages effectuant une analyse du **flot de données**.

Langage	**Type de graphe**	**Ordonnancement**	**Modèle de coût**	**Architecture visée**
à base de processus légers	×	quelques choix	×	SMP
Jade	flot de données emboîté	fixé	×	SMP et distribuée
Modèle BSP	×	×	durée	SMP et distribuée
NESL	flot de données série-parallèle	fixé	durée et mémoire	SMP et distribuée
Cilk	précédence série-parallèle	fixé	durée et mémoire	SMP
Athapascan-1	flot de données emboîté	entièrement adaptable	durée et mémoire	SMP et distribuée

TABLEAU 2.1 – Comparaison de différents langages de programmation parallèle de « haut niveau » [65 - Galilée (1999)]

Enfin, le parallélisme peut être de type **« série-parallèle »** si les synchronisations entre les tâches sont effectuées par fratrie : la tâche mère[‡] est seule capable de synchroniser ses filles, et cette synchronisation est globale sur l'ensemble des filles créées. Par exemple, une tâche créant du parallélisme à l'aide d'une séquence est bloquée jusqu'à la terminaison du calcul de tous les éléments de la séquence. Le parallélisme peut, d'autre part, être de type **« emboîté »** lorsque les accès aux

[‡]La notion de tâche mère/tâche fi lle correspond au contexte de création de la tâche : la mère de la tâche créée est la tâche qui exécute l'instruction de création.

données partagées effectués par les filles constituent un sous-ensemble des accès effectués par la tâche mère. Par exemple, si deux tâches n'accèdent à aucune donnée en commun, alors ces deux tâches n'ont aucune contrainte de précédence et peuvent être exécutées dans un ordre quelconque l'une par rapport à l'autre et donc, en particulier, en parallèle. Si, par contre, elles accèdent en commun à une même donnée et si l'un des accès est une écriture ou une libération, alors il y a contrainte de précédence entre les tâches : elles doivent s'exécuter séquentiellement et une synchronisation doit être insérée entre ces deux tâches.

Nos applications sont des calculs matriciels sur de grandes matrices. Un modèle de coût garantissant une durée et surtout une taille mémoire bornées est indispensable. En outre, nos algorithmes doivent pouvoir s'exécuter sur différents types de machines, afin de toujours profiter des meilleurs matériels. Ensuite, nos matrices sont creuses mais elles ne sont pas forcément très structurées. Le comportement précis des algorithmes peut n'être connu qu'à l'exécution, du fait de cette **irrégularité** des données. Un ordonnancement dynamique et spécialisé est alors nécessaire pour augmenter les performances. Nous avons donc choisi Athapascan-1 comme langage de programmation parallèle : en effet, il permet d'adapter l'ordonnancement, tout en garantissant une exécution bornée en temps et en mémoire et, d'autre part, il est développé au laboratoire ID, permettant ainsi un suivi aisé. Enfin, son utilisation est simple puisqu'il s'agit d'une bibliothèque C++ augmentée de seulement deux mots clés : `Fork` et `Shared`. Athapascan-1 est étudié plus en détails dans la section suivante.

2.3 ATHAPASCAN-1

2.3.1 INTERFACE DE PROGRAMMATION

Tout d'abord, dans Athapascan-1, le parallélisme est explicite. L'utilisateur définit ses tâches de calcul et les données partagées. Pour réaliser cela, deux mots clés seulement sont nécessaires.

`Fork` permet de lancer l'exécution en parallèle des tâches de calcul. Les paramètres de construction d'un objet `Fork` permettent de spécifier l'algorithme d'ordonnancement choisi pour cette tâche. Toute classe C++ possédant un opérateur `void operator()` peut être lancée en parallèle par `Fork` (c'est cet opérateur

qui sera exécuté en parallèle).

`Shared` permet de spécifier les accès en *lecture* (`.read()`) ou *écriture* (`.write()`) ou *modification* (lectures et/ou écritures, `.access()`) effectués par la procédure. Cette spécification est réalisée par l'adjonction d'un suffixe au mot clé `Shared` :

`Shared_r` pour un accès en lecture,
`Shared_w` pour un accès en écriture,
`Shared_r_w` pour un accès en modification,
`Shared_cw` pour une écriture cumulative par plusieurs procédures.

Pour expliquer ces fonctionnalités, nous proposons un exemple, l'élimination de Gauß sur matrice dense, stockée par lignes [§].

2.3.2 ÉLIMINATION DE GAUSS AVEC ATHAPASCAN-1

Nous donnons ici à titre d'exemple le code Athapascan-1 pour l'élimination de Gauß exacte sur une matrice dense, a priori non inversible.

Code 2.3.1 Élimination de Gauß avec Athapascan-1

```
1   struct SearchPivot {
    // pivot_row   : la ligne de recherche
    // i           : le numéro de ligne courante
    // rank        : le rang actuel
5   // permutation : indique l'indice du prochain pivot
      void operator() (Shared_r<vector<int> > pivot_row,
                       int i, int n_col,
                       Shared_cw<add_int, int> rank,
                       Shared_w<int> permutation) {
10   // recherche un élément non nul dans la ligne
     // ajoute 1 au rang si un élément non nul est trouvé
     // indique l'emplacement de ce pivot.
         for(int k = 0; (k<n_col) &&
                         (pivot_row.read()[k] == 0); ++k)
15       if (k < n_col) {
           permutation.write( new int(k) );
           rank.cumul( 1 ) ;
         } else
```

[§] Pour plus de détails, le manuel de programmation Athapascan-1 est consultable à cette adresse : http ://www-apache.imag.fr/software/ath1/manual

```
          permutation.write( new int(-1));
20      }
    };

    struct EliminateRow {
      void operator() (Shared_r<vector<int> > pivot_row,
25                     Shared_r_w< vector<int> > elim_row,
                       int i, int n_col,
                       Shared_r< int > permutation);
    // La ligne elim_row est éliminée à l'aide de la ligne
    // de pivot. Ne fait rien si permutation == -1.
30  };

    struct ComputeRank {
    // rank         : le rang en sortie
    // n_row, n_col : les dimensions de la matrice
35  // Mat          : une matrice dense stockée
    //                par lignes partagées
    // Les opérations sont effectuées en place sur Mat.
      void operator() (Shared<int >& rank,
                       vector<Shared<vector<int> > >& Mat,
40                     int n_row, int n_col) {
          vector< Shared<int> > permutation(n_row);
          int last = n_row - 1;
          for (int i=0; i<last;++i) {
            permutation[i] = Shared<int>(0);
45            // Recherche d'un pivot non nul
            Fork<SearchPivot> () ( Mat[i], i, n_col,
                                   rank, permutation[i]) ;
            for(int k=i+1; k<n_row; ++k)
                    // Élimination des lignes suivantes
50            Fork< EliminateRow >() ( Mat[i],
                                       Mat[k], i, n_col,
                                       permutation[i] );
          }
          permutation[last] = Shared<int>(0);
55        Fork<SearchPivot>() ( Mat[last], last, n_col,
                                rank, permutation[last]);
      }
    };
```

Nous n'avons pas choisi d'ordonnancement particulier, donc les tâches `Fork` sont créées par le constructeur par défaut (`Fork<task>()`) qui utilise ici l'or-

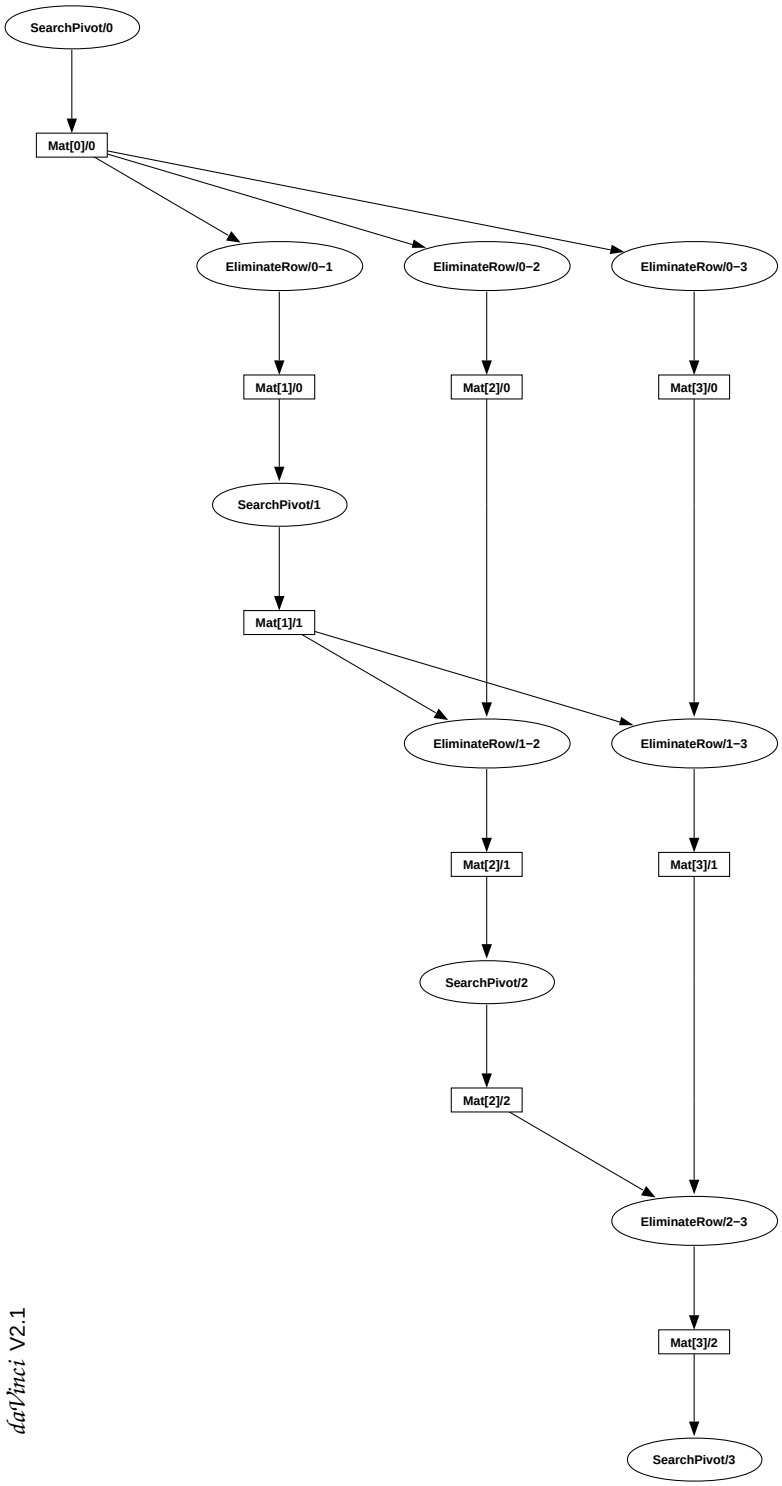

FIGURE 2.2 – Graphe de dépendance : élimination 4×4

donnancement par défaut (algorithme de liste, de type « glouton »). Nous voyons aussi l'utilisation d'une variable en écriture cumulative : à chaque fois qu'un pivot non nul est découvert, 1 est accumulé dans la variable `rank` par la fonction `add_int`.

À l'aide de la seule description des accès des procédures à leurs données, Athapascan-1 est capable de tirer le graphe de flot de données du programme et ainsi le parallélisme maximal de l'algorithme. Le graphe 2.2, page 34 donne une partie du graphe généré par Athapascan-1 pour l'exécution de cet algorithme sur une matrice 4×4. Les ellipses sont les tâches de calculs, les rectangles sont les données partagées. Une flèche provenant d'un rectangle signifie qu'une lecture est faite sur cette donnée ; une flèche provenant d'une ellipse signifie que la procédure va écrire sur cette donnée.

2.3.3 MODÈLE DE COÛT

Le graphe de flot de données ainsi généré par Athapascan-1 permet d'une part de fournir, aux politiques d'ordonnancement, une information complète sur l'exécution (précédences et localités des tâches et des données). Il permet d'autre part d'associer un modèle de coût à l'interface de programmation. Chaque programme est caractérisé par les grandeurs suivantes :

- T_1, le travail, qui correspond au nombre d'instructions exécutées, identique au nombre de nœuds du graphe et à la durée sur un processeur : les tâches sont considérées de durée unitaire et peuvent allouer au plus une quantité bornée de mémoire.
- T_∞, la longueur du chemin critique du graphe, correspond à la durée de l'exécution sur un nombre infini de processeurs.
- T_p, la durée de l'exécution sur un nombre infini de$p$ processeurs.
- S_1, l'espace mémoire requis par une exécution séquentielle, cette exécution correspondant à un parcours du graphe en profondeur d'abord (en parcourant les tâches filles selon leur ordre de création, c'est-à-dire de gauche à droite dans les représentations classiques du graphe).
- S_p, l'espace mémoire total requis par une exécution sur p processeurs.
- q, le nombre de processeurs virtuels émulés sur les p processeurs réels de la machine (afin de pouvoir majorer la latence des communications soit par du calcul soit par d'autres communications [29 - Carissimi (1999), Section 2.2.3]).
- h, le délai d'accès à distance d'un bit de donnée. Ce délai peut être borné en utilisant des processeurs virtuels, comme présenté dans [93 - Karp et al. (1996)] (h dépend de p et de q).

- C_1, le volume total d'accès distants. Cette valeur représente la somme sur tout le graphe d'exécution G des tailles des données accédées en lecture directe.
- C_∞, le volume d'accès distant effectué par un plus long chemin dans ce graphe (selon ce critère d'accès).
- σ, la taille du graphe G, c'est-à-dire le nombre de nœuds et d'arêtes.

Il est possible, à l'aide de ce modèle de coût, de garantir la durée et la consommation mémoire de toute exécution en fonction de la stratégie d'ordonnancement utilisée [65 - Galilée (1999), Section 6.4] et [49 - Doreille (1999), Section 5.4] : par exemple, une politique d'ordonnancement de type « glouton » (vol de travail) donne les bornes suivantes :

$$\begin{aligned} T_p &\leq \frac{T_1 + hC_1}{p} + \frac{q}{p}(T_\infty + hC_\infty) + h\frac{q}{p}\mathcal{O}\left(\sigma\right) \\ S_p &\leq S_1 + q\mathcal{O}(T_\infty + hC_\infty) \end{aligned}$$

Il est possible, en utilisant une autre politique d'ordonnancement, d'obtenir les bornes suivantes dans le cas d'un graphe série-parallèle :

$$\begin{aligned} T_p &\leq \frac{T_1 + hC_1}{p} + \frac{q}{p}\mathcal{O}(T_\infty + hC_\infty) \\ S_p &\leq qS_1 \end{aligned}$$

2.3.4 CONCLUSIONS

Les techniques d'ordonnancement en ligne de type « glouton », où les processeurs sont maintenus au maximum en activité, permettent d'atteindre des efficacités en temps quasi-optimales, de l'ordre de $\frac{T_1}{p} + T_\infty$. Ensuite, différentes variantes des algorithmes gloutons permettent de garantir une majoration de la consommation mémoire. Dans le cas de la résolution de très grandes matrices creuses, ces deux aspects sont primordiaux et, dans la suite, nous nous servirons de ces bornes pour nos algorithmes parallèles.

Enfin, la bibliothèque Athapascan-1 est principalement destinée à la programmation des machines à mémoire distribuée, mais l'implantation étant entièrement portable, les machines à mémoire partagée de type SMP peuvent également être exploitées (la portabilité repose sur la couche Athapascan-0 qui repose sur MPI et une bibliothèque de processus légers de type POSIX).

Première partie

Corps finis

Outre les ensembles infinis, mais dénombrables, le calcul formel permet aussi de calculer dans des ensembles finis abstraits tels que les sous-groupes de $\mathbb{Z}$ ou encore les corps finis et leurs extensions algébriques.

En effet, il est souvent intéressant, pour résoudre un problème de calcul exact, de le résoudre modulo un ou plusieurs entiers au lieu de le résoudre directement sur les entiers en précision arbitraire. De nombreux avantages résultent de cette approche. Par exemple, elle permet de limiter le temps et l'espace mémoire nécessaires lorsque des expressions intermédiaires utilisent de trop grands coefficients entiers. En particulier, cela est le cas pour les calculs de forme normale de matrice que nous verrons dans la troisième partie. Un autre point est que si les moduli ne dépassent pas la valeur maximale des entiers machine, les calculs ne nécessitent plus que quelques cycles. Il est alors plus avantageux d'effectuer plusieurs calculs modulaires plutôt qu'un seul calcul avec une représentation lourde des entiers en précision arbitraire. Enfin, les calculs modulaires sont souvent indépendants les uns des autres et permettent donc une parallélisation simple et efficace. Dans tous les cas, il est de la plus grande importance de pouvoir implémenter de manière efficace une arithmétique modulaire. En outre, il est utile que cette arithmétique soit efficace pour des entiers les plus grands possible, afin de limiter le nombre de moduli différents.

Enfin, certains algorithmes travaillent sur des corps finis. La factorisation d'entiers, par exemple, peut nécessiter de travailler modulo 2. Une partie des calculs de forme normale de matrices nécessite aussi de travailler modulo des petits nombres premiers avec des algorithmes probabilistes. Le problème est que la probabilité de réussite est souvent liée à la taille du corps. Il est alors possible de continuer à travailler avec ces petits nombres premiers tout en plongeant le corps dans une de ses extensions, de plus grande taille. Là encore, il est vital de posséder des implémentations efficaces de ces extensions.

C'est pourquoi nous nous intéressons dans cette partie à définir des représentations des entiers modulaires et des extensions de corps finis qui soient les plus efficaces possibles.

3

CONSTRUCTION DES CORPS FINIS

« Quand quelqu'un vous dit : "Je me tue à vous le dire", laissez-le mourir. »

Jacques Prévert

Sommaire

Rappelons tout d'abord quelques définitions et propriétés des corps finis que nous utiliserons par la suite. Nous renvoyons le lecteur à [44 - Demazure (1997)], par exemple, pour plus de détails :

Définitions 3.0.1

- *Un* **corps** *est un anneau non vide dans lequel tout élément non nul possède un inverse. L'ensemble des inversibles d'un ensemble F est noté F^*.*
- *Le corps des nombres rationnels $\mathbb{Q}$ et les corps $\mathbb{Z}/_p\mathbb{Z}$, pour p premier, sont appelés* **corps premiers***.*
- *Les corps finis sont appelés* **corps de Galois***. Ils sont notés $\mathtt{GF}(q)$, où q est le cardinal du corps.*

Définitions 3.0.2

- *Le* **cardinal** *d'un corps fini est son nombre d'éléments.*
- *Soit un corps $\mathbb{F}$; le noyau de l'homomorphisme de groupe $n \to n.1_{\mathbb{F}}$ de $\mathbb{Z}$ dans $\mathbb{F}$ est l'ensemble des multiples d'un entier $p \geq 0$. Cet entier est appelé la* **caractéristique** *du corps $\mathbb{F}$.*
- *Dans un groupe multiplicatif fini, pour chaque élément f, il existe un plus petit entier non nul m, tel que $f^m = 1$. Cet entier est appelé l'***ordre** *de l'élément f. Par extension, l'ordre d'un élément inversible d'un corps fini est son ordre dans le groupe des inversibles.*
- *Un* **générateur** *d'un groupe multiplicatif fini de cardinal h est un élément d'ordre h.*
- *Un groupe multiplicatif est* **cyclique** *si il possède au moins un générateur.*

Un exemple de corps fini est l'ensemble $\mathbb{Z}/_p\mathbb{Z}$, des entiers modulo un nombre premier p ; il est de caractéristique et de cardinal p. Considérons $\mathbb{Z}/_7\mathbb{Z}$. Alors, par exemple, 3 est un générateur car $3^1 = 3$, $3^2 = 2$, $3^3 = 6$, $3^4 = 4$, $3^5 = 5$, $3^6 = 1$.

3.1 CONSTRUCTION DES CORPS DE GALOIS

Construire un corps fini, c'est différencier ses éléments et calculer avec eux. Nous montrons tout d'abord qu'il suffit d'étudier les corps premiers et de considérer les autres comme des extensions de ceux-ci. Ensuite, nous montrons des méthodes effectives pour calculer dans ces corps.

Propriétés 3.1.1

- *Si la caractéristique d'un corps est non nulle, c'est un nombre premier.*
- *Si la caractéristique d'un corps fini est non nulle, le cardinal du corps est une puissance de la caractéristique.*
- *Tous les corps finis de même cardinal sont isomorphes.*
- *Le cardinal de tout sous-corps d'un corps fini est un diviseur du cardinal du corps.*
- *L'ordre de tout élément d'un corps fini est un diviseur du cardinal du groupe des inversibles du corps.*
- *Le groupe des inversibles, $\mathbb{F}^*$, d'un corps fini $\mathbb{F} = \mathtt{GF}(q)$ est cyclique de cardinal $q-1$.*

Ainsi, comme les corps finis de même cardinal $q = p^n$ sont tous isomorphes, il suffit d'en connaître un pour les connaître tous. Par exemple, soit P un polynôme irréductible de degré n dans $\mathbb{Z}/_{\mathrm{p}}\mathbb{Z}[X]$, l'ensemble $\mathbb{Z}/_{\mathrm{p}}\mathbb{Z}\,[\mathrm{X}]/\,\mathrm{P}$ peut être muni d'une structure de corps et est de cardinal p^n. Une construction classique de l'arithmétique dans un corps fini est donc d'implémenter $\mathbb{Z}/_{\mathrm{p}}\mathbb{Z}$, de chercher un polynôme irréductible, P, dans $\mathbb{Z}/_{\mathrm{p}}\mathbb{Z}[X]$ de degré n, puis de représenter les éléments de $\mathtt{GF}(p^n)$ par des polynômes, ou des vecteurs, et d'implémenter les opérations arithmétiques comme des opérations modulo p et P. Cette méthode est utilisée dans la bibliothèque NTL [153 - Shoup (2000)], par exemple.

Une autre idée [154 - Sibert et al. (1990)] consiste à utiliser la dernière propriété 3.1.1 : le groupe multiplicatif des inversibles d'un corps fini est cyclique, c'est-à-dire qu'il existe au moins un générateur et que le corps est généré par les puissances de ce générateur. Il s'agit de représenter les éléments inversibles par un indice, leur puissance, et zéro par un indice spécial. Ainsi, si g est un générateur du groupe multiplicatif d'un corps fini $\mathtt{GF}(q)$, tous les inversibles peuvent s'écrire g^i. De plus, comme ce groupe est cyclique, il suffit de considérer, pour i, un intervalle

de taille q pour générer l'ensemble des inversibles. Les opérations arithmétiques classiques sont alors grandement simplifiées, en utilisant la proposition suivante :

Proposition 3.1.2

Soit $\mathtt{GF}(q)$ un corps fini. Soit g un générateur de $\mathtt{GF}(q)^*$. Alors $g^{q-1} = 1_{\mathtt{GF}(q)}$. En outre, si la caractéristique de $\mathtt{GF}(q)$ est impaire, alors $g^{\frac{q-1}{2}} = -1_{\mathtt{GF}(q)}$ et dans le cas contraire, $1_{\mathtt{GF}(2^n)} = -1_{\mathtt{GF}(2^n)}$.

Preuve de 3.1.2 : L'ordre de g divise $q-1$ donc $g^{q-1} = 1_{\mathtt{GF}(q)}$. Si le corps est de caractéristique 2, alors, comme dans $\mathbb{Z}/2\mathbb{Z}[X]$, $1 = -1$. Sinon $\frac{q-1}{2} \in \mathbb{Z}$ et donc $g^{\frac{q-1}{2}} \in \mathtt{GF}(q)$. Or comme on est dans un corps, l'équation $X^2 = 1$ possède au plus deux racines, 1 et -1. Or g est un générateur donc l'ordre de g est $q-1$ et non pas $\frac{q-1}{2}$. La seule possibilité restante est $g^{\frac{q-1}{2}} = -1$. □

Cela donne le codage suivant pour un élément $x \in \mathtt{GF}(q)$ si $\mathtt{GF}(q)$ est généré par g :

$$\begin{cases} 0 & \text{si } x = 0 \\ q-1 & \text{si } x = 1 \\ i & \text{si } x = g^i \end{cases}$$

En particulier, dans notre codage, notons $\overline{q} = q-1$ le représentant de $1_{\mathtt{GF}(q)}$. Nous notons aussi i_{-1} l'indice de $-1_{\mathtt{GF}(q)}$; il vaut $\frac{q-1}{2}$ si $\mathtt{GF}(q)$ est de caractéristique impaire et $q-1$ sinon. Ce qui donne la possibilité d'écrire simplement les opérations arithmétiques.

- La multiplication et la division d'inversibles sont respectivement une addition et une soustraction d'indices modulo $q-1$.
- La négation est donc simplement l'identité en caractéristique 2 et l'addition de $\frac{q-1}{2}$ modulo $q-1$ en caractéristique impaire.
- L'addition est l'opération la plus complexe. Il faut l'implémenter en utilisant les autres opérations, par exemple de cette manière : si g^i et g^j sont deux éléments non nuls d'un corps fini, $g^i + g^j = g^i(1 + g^{j-i})$. En construisant une table, $t\_plus1[]$, de taille q, des successeurs de chaque élément du corps, l'addition est implémentée par une soustraction d'indice, un accès à une table et une addition d'indices.

Dans le tableau 3.1, page 45, nous montrons le calcul de ces opérations sur les indices, en considérant une seule table de taille q, celle des successeurs. Au chapitre suivant, nous proposons une autre implémentation, avec plusieurs tables de plus grandes tailles (section 4.1.3, page 63), qui permettent de réduire le nombre de tests et de calculs. Nous nous intéressons ici à la complexité du calcul en utili-

Opération	Éléments	Indices	Coût		
			+/-	Tests	Accès
Multiplication	$g^i * g^j$	$i + j\ (-\overline{q})$	1.5	1	0
Division	g^i / g^j	$i - j\ (+\overline{q})$	1.5	1	0
Négation	$-g^i$	$i - i_{-1}\ (+\overline{q})$	1.5	1	0
Addition	$g^i + g^j$	$k = j - i\ (+\overline{q})$ $i + t\_plus1[k]\ (-\overline{q})$	3	2	1
Soustraction	$g^i - g^j$	$k = j - i + i_{-1}\ (\pm\overline{q})$ $i + t\_plus1[k]\ (-\overline{q})$	3.75	2.75	1

TABLEAU 3.1 – Opérations sur les inversibles avec générateur en caractéristique impaire

sant le moins de mémoire possible, en considérant des éléments aléatoires. Nous indiquons le coût des calculs en nombre d'additions et de soustractions (+/-), en nombre de tests et en nombre d'accès dans une table.

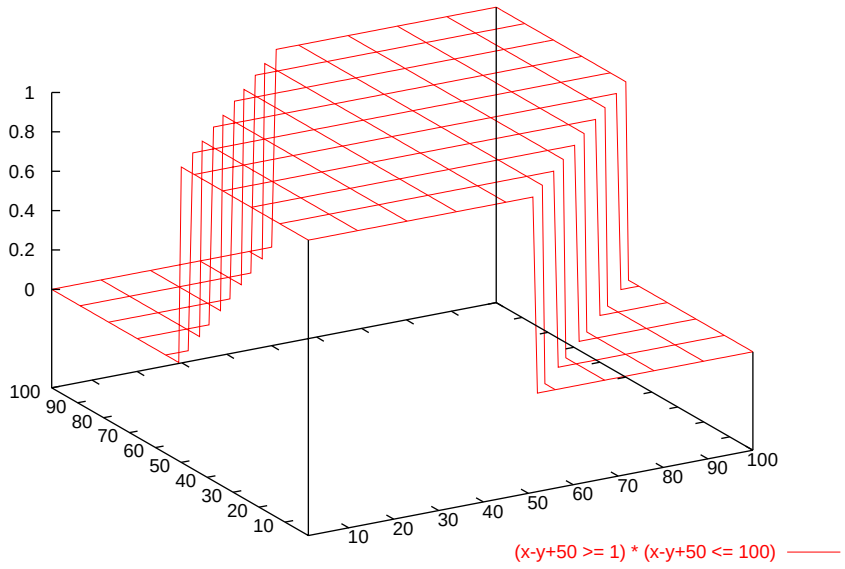

FIGURE 3.1 – Répartition de $j - i + \frac{q-1}{2}$ lorsque i et j varient de 1 à $q - 1$, pour $q = 101$

Nous avons compté 0.5 additions ou soustractions pour un modulo $\overline{q}$ dans le cas de la multiplication, la division, la négation et l'addition dans le corps fini ; en effet, dans la moitié des cas, il n'est pas nécessaire d'ajouter ni de retirer $\overline{q}$ pour que l'indice résultant soit compris entre 0 et $\overline{q}$. Le cas de la soustraction dans le corps fini nécessite une étude plus fine. Dans ce cas, $j - i + i_{-1}$ est compris entre $-\frac{\overline{q}}{2}$ et $\frac{3\overline{q}}{2}$ et ne nécessite un ajout ou une soustraction de $\overline{q}$ que dans 2 cas sur 8 comme le montre la figure 3.1, page 45. On en déduit un total de $2 + 0.25 + 1 + 0.5 = 3.75$ additions ou soustractions et $1 + 0.75 + 1 = 2.75$ tests. Il est possible de réduire le nombre d'additions et soustractions, par exemple en remplaçant $i + x$ par $i + x - \frac{q-1}{2}$ et en ajoutant un test. Cependant, pour de nombreuses architectures, un test du type $a > q$? peut être aussi coûteux que l'opération $a - q$. Nous proposons donc l'implémentation qui minimise le nombre total d'opérations (additions, soustractions, tests, accès), tout en n'utilisant qu'une seule table.

Ces opérations sont signées et restent valides tant que les indices sont plus petits que la moitié de la valeur maximale d'un entier machine. Alors, si un mot machine est codé sur 32 bits, la mantisse signée est codée sur 31 bits et ces opérations sont donc valables pour un corps de taille jusqu'à 30 bits, c'est-à-dire jusqu'à $2^{30} - 1$ éléments. Notons qu'il est possible de travailler avec des corps de taille 31 bits en utilisant l'arithmétique non signée et en ajoutant des tests de dépassement de capacité. En fait, la nécessité d'une table des successeurs implique souvent une plus petite limite du fait de l'espace mémoire disponible. Si l'on travaille sur des entiers codés sur 32 bits, c'est-à-dire 4 octets, un corps de taille 2^{26} nécessitera déjà au moins 256 Méga octets.

Nous montrons l'implémentation en C++ de ces opérations dans le prochain chapitre, section 4.1.2, page 61. De cette manière, si un générateur est connu, les opérations et le stockage dans un corps fini peuvent être simples. Pour cela, il faudra générer le corps fini ($\mathbb{Z}/p\mathbb{Z}$ ou $\mathbb{Z}/p\mathbb{Z}[X]/P$) puis calculer un générateur. Il est alors possible de précalculer les puissances successives de ce générateur pour identifier les indices de chaque élément et stocker ainsi les correspondances entre éléments et indices à l'aide de deux autres tables de taille q. Une autre possibilité, si la mémoire disponible est restreinte, ou si il y a peu de conversions entre représentation interne (les indices) et représentation externe (les éléments), est de faire ces calculs à chaque fois qu'une conversion est nécessaire.

Le point qu'il reste donc à élucider est la construction d'un générateur. On ne sait pas, en général, trouver explicitement de générateur d'un corps fini donné. Néanmoins, il est possible d'en trouver par essais successifs, comme nous allons le voir dans les sections suivantes.

3.2 RACINES PRIMITIVES DANS LES SOUS-GROUPES DE $\mathbb{Z}$

Un générateur du groupe des inversibles de $\mathbb{Z}/_{m}\mathbb{Z}$ est appelé racine primitive de m. Nous allons voir qu'il existe des racines primitives pour tous les nombres premiers, ainsi que pour quelques autres nombres particuliers, même si dans ce cas $\mathbb{Z}/_{m}\mathbb{Z}$ n'est plus un corps. Nous allons avoir besoin de l'indicateur d'Euler, en particulier. Nous renvoyons le lecteur à [26 - Burton (1998)], par exemple, pour plus de détails.

Définitions 3.2.1

- *Le nombre d'entiers positifs non nuls premiers avec un entier m et plus petits que m est appelé l'***indicateur d'Euler** *de m et est noté $\phi(m)$.*
- *Une* **racine primitive** *d'un entier m est un entier premier avec m et d'ordre $\phi(m)$ modulo m. La* **plus petite racine primitive** *de m est notée $\chi(m)$.*

Propriétés 3.2.2

[26 - Burton (1998), Théorèmes 7.1 & 7.2]
- *Si p est premier et $k > 0$ est un entier, alors $\phi(p^k) = p^{k-1}(p-1)$.*
- *Si $m = pq$ et p et q sont premiers entre eux, alors $\phi(m) = \phi(p)\phi(q)$.*

Propriétés 3.2.3

[26 - Burton (1998), Théorèmes 8.4 & 8.10]
- *Si m possède une racine primitive, alors il en a exactement $\phi(\phi(m))$.*
- *Un entier $m > 1$ possède une racine primitive si et seulement si $m = 2, 4, p^k$ ou $2p^k$ avec p un nombre premier impair et $k > 0$ entier.*

Les théorèmes de Burton nous indiquent donc dans quels cas nous pouvons espérer avoir une racine primitive. Il faut maintenant en calculer au moins une. Pour trouver une racine primitive d'un nombre, nous avons tout d'abord besoin de tester si un entier donné satisfait cette propriété. Pour ce faire, il suffit (!) de posséder des algorithmes de test de primalité et de factorisation d'entiers, pour

calculer $\phi(m)$ puis tester si l'ordre de l'élément est bien $\phi(m)$:

Algorithme 3.2.4 *Test-Racine-Primitive*

Entrées : – Un entier $m > 0$.
– Un entier $a > 0$.

Sorties : – Oui, si a est une racine primitive de m ; Non dans le cas contraire.

1 : **Si** a et m ne sont pas premiers entre eux **Alors** Renvoyer "Non".
2 : $\phi_m = \phi(m)$ {Factorisation de m et calcul par les propriétés 3.2.2, page 47}
3 : **Pour tout** p, premier et divisant ϕ_m, **Faire** {Factorisation de $\phi(m)$}
4 : **Si** $a^{\frac{\phi_m}{p}} \equiv 1[m]$ **Alors** Renvoyer "Non". {Calcul récursif par carrés}
5 : Renvoyer "Oui".

Théorème 3.2.5

L'algorithme Test-Racine-Primitive *est correct.*

Preuve de 3.2.5 : On utilise [26 - Burton (1998), Théorème 8.1] : soit un entier a, d'ordre k modulo m. Alors $a^h \equiv 1[m]$ si et seulement si $k|h$. On en déduit que si l'ordre de a est plus petit que $\phi(m)$, comme il doit diviser $\phi(m)$, nécessairement l'une des valeurs $\frac{\phi_m}{p}$ sera un multiple de l'ordre de a. Dans le cas contraire, la seule valeur possible pour l'ordre de a est $\phi(m)$. □

Une première méthode de calcul est alors d'essayer un à un tous les entiers plus petits que m, qui ne soient ni 1, ni -1, ni une puissance sur les entiers, et de trouver ainsi la plus petite racine primitive de m. De nombreux résultats théoriques existent [120 - Murata (1991), 60 - Elliott et Murata (1997)] prouvant qu'en général, il ne faut pas trop d'essais pour la trouver, de l'ordre de

$$\chi(m) = \mathcal{O}\left(r^4(\ln(r)+1)^4 \ln^2(m)\right)$$

avec r le nombre de facteurs premiers distincts de m [151 - Shoup (1992)]. En pratique, $\chi(m)$ semble être encore plus petit ; d'après Tomás Oliveira e Silva [132 - Oliveira e Silva (2000)], il apparaît qu'environ 80% des nombres premiers inférieurs à 8910000000000 ont une racine primitive plus petite que 6 et, même, 306841261647 des plus petites racines primitives, sur ces 309582581120 premiers nombres premiers, sont plus petites que 23. Ainsi, dans plus de 99% de ces cas, il suffira de 18 tests pour découvrir une racine primitive d'un nombre premier.

Une autre méthode est de tirer aléatoirement des entiers plus petits que m et de tester si ceux-ci sont une racine primitive ou non. Étant donné qu'il y a $\phi(p-1)$ racines primitives dans $\mathbb{Z}/_{\mathrm{p}}\mathbb{Z}$ pour p premier, la probabilité d'en trouver une est de $\frac{\phi(p-1)}{p-1}$ et donc l'espérance du nombre de tirages pour tomber sur une racine primitive est de $\frac{p-1}{\phi(p-1)}$. Ce qui nous donne une meilleure chance que la force brute puisque Rosser et Schoenfeld [145 - Rosser et Schoenfeld (1962), Théorème 15] ont montré l'inégalité suivante où C est la constante d'Euler, $C \approx 0.5772156649\ldots$:

$$\frac{m}{\phi(m)} < e^C \ln(\ln(m)) + \frac{5}{2\ln(\ln(m))}, \forall m \geq 3, m \neq 223092870 \qquad (3.1)$$

En outre, comme $223092871 = 317 \times 703763$ n'est pas premier, nous pouvons utiliser cette inégalité pour toutes les racines primitives de nombres premiers. Il est de plus conjecturé que $e^C \ln(\ln(m)) < \frac{m}{\phi(m)}$ pour un nombre infini de m, cette borne semble donc très bonne. Ainsi, pour $7 \leq m \leq 8910000000000$, elle donne une valeur maximale d'environ 6.78330. En pratique c'est encore mieux, puisqu'il y a seulement 36 nombres premiers inférieurs à 10000000000 avec $\frac{p-1}{\phi(p-1)} > 6.0$.

En conclusion, l'algorithme le plus rapide nous semble donc être une combinaison de ces deux méthodes. Pour 80% des nombres premiers inférieurs à 8910000000000 , il fera moins de 4 tests et dans les autres cas, il fera en moyenne moins de 11 tests.

Enfin, le cas des entiers non premiers se ramène au cas premier à l'aide des théorèmes suivants :

Théorème 3.2.6

[26 - Burton (1998), Théorème 8.9 et Corollaire]
- *Si a est une racine primitive de p premier impair, alors a ou $a+p$ est une racine primitive de p^k pour $k \geq 2$.*
- *Si a est une racine primitive de p^k avec p premier impair, alors celui de a et $a+p^k$ qui est impair est une racine primitive de $2p^k$.*

Nous en déduisons l'algorithme général.

Algorithme 3.2.7 *Racine-Primitive*

Entrées : – Un entier $m = 2$, 4, p^k ou $2p^k$ avec p premier impair.

Sorties : – a une racine primitive de m.

1 : **Si** $m == 2$ **Alors** Renvoyer 1.
2 : **Si** $m == 4$ **Alors** Renvoyer 3.
3 : **Pour** $b = 2, 3, 5, 6$ **Faire**
4 : **Si** *Test-Racine-Primitive* (p, b) == "Oui" **Alors** $a = b$
5 : **Tant que** *Test-Racine-Primitive* (p, a) == "Non" **Faire**
6 : Sélectionner a au hasard entre 7 et $p - 1$.
7 : **Si** *Test-Racine-Primitive* (p^k, a) == "Non" **Alors** $a += p$
8 : **Si** $m == p^k$ **ou si** a est impair **Alors**
9 : Renvoyer a.
10 : **Sinon**
11 : Renvoyer $a + p^k$.

En pratique, cet algorithme, exécuté sur un pentium cadencé à 333 MHz, et implémenté en utilisant *Gnu Multiprecision Package*[*] comme arithmétique, n'a jamais mis plus d'un dixième de seconde pour trouver une racine primitive d'entiers aléatoires bornés par 2^{64} !

3.3 GÉNÉRATEURS DES EXTENSIONS

Nous savons donc maintenant trouver un générateur pour un corps premier, et plus généralement pour tout sous-groupe $\mathbb{Z}/_{p^k}\mathbb{Z}$ de $\mathbb{Z}$: il faut trouver une racine primitive. Mais si $k > 1$ un tel sous-groupe n'est pas un corps. Nous nous intéressons maintenant aux corps finis : les $\mathtt{GF}(p^k)$ extensions des corps $\mathtt{GF}(p) = \mathbb{Z}/_{\mathrm{p}}\mathbb{Z}$. Pour construire une telle extension, il faut tout d'abord construire le corps fini $\mathtt{GF}(p)$, par la méthode de la section précédente, par exemple. Ensuite, il est possible d'utiliser un polynôme de degré k, irréductible sur ce corps, qui définit une extension de degré k, et donc de taille p^k, comme nous l'avons vu au début du chapitre. Nous commençons donc par construire l'extension avec des polynômes puis cherchons un *polynôme* générateur de cette extension pour pouvoir coder les éléments non plus par des polynômes, mais par leurs indices. Le codage et les opérations sont alors les *mêmes* que ceux des sous-groupes de $\mathbb{Z}$. Nous étudions ici les algorithmes permettant de trouver un polynôme irréductible possédant un générateur simple. En premier lieu, nous avons besoin de la notion de polynôme cyclotomique.

[*] http ://www.gnu.org/gnulist/production/gnump.html

3.3.1 POLYNÔMES CYCLOTOMIQUES

Le lecteur trouvera plus de détails dans [44 - Demazure (1997), 68 - Gathen et Gerhard (1999)], par exemple.

Définition 3.3.1

Soit un entier $n > 0$. Le $n^{\text{ième}}$ **polynôme cyclotomique** *est noté $\Theta_n(X)$ et vaut : $\Theta_1(X) = X - 1$ et*

$$\Theta_n(X) = \prod_{k \in \mathbb{Z}/n\mathbb{Z}^*} (X - e^{2i\pi \frac{k}{n}}) \text{ pour } n \geq 2$$

Propriétés 3.3.2

- *Θ_n est un polynôme unitaire à coefficients entiers, de degré $\phi(n)$.*
- *$X^n - 1 = \prod_{d|n} \Theta_d(X)$.*

Cette dernière propriété permet de calculer les polynômes cyclotomiques de manière récursive, avec de nombreux raccourcis, comme par exemple : $\Theta_p(X) = \sum_{i=0}^{p-1} X^i$, pour p premier ; $\Theta_{p^r}(X) = \Theta_p(X^{p^{r-1}})$, pour p premier et $r > 0$; $\Theta_{4n}(X) = \Theta_{2n}(X^2)$; $\Theta_{4n+2}(X) = \Theta_{2n+1}(-X)$; $\Theta_{pq}(X)\Theta_q(X) = \Theta_q(X^p)$, si p et q sont premiers entre eux ; etc.

Ensuite, les propriétés suivantes sont utiles pour déterminer les polynômes irréductibles, à partir des polynômes cyclotomiques.

Propriétés 3.3.3

- *$\forall n > 0, \Theta_n(X)$ est irréductible dans $\mathbb{Q}[X]$.*
- *Soient q et n premiers entre eux. Si r est l'ordre de q dans $\mathbb{Z}/n\mathbb{Z}$ alors, dans $\mathtt{GF}(q)$, $\Theta_n(X)$ est un produit de polynômes unitaires irréductibles distincts de degré r.*

On déduit directement de cette dernière propriété une méthode pour calculer un polynôme irréductible de degré r dans $\mathbb{Z}/p\mathbb{Z}$: il suffit (!) de trouver un facteur de Θ_{p^r-1}, puisque l'ordre de p dans $\mathbb{Z}/(p^r-1)\mathbb{Z}$ est r. La factorisation est ici, par exemple, une simplification de l'algorithme de Cantor et Zassenhaus [28 - Can-

tor et Zassenhaus (1981)], puisque tous les facteurs sont de même degré [92 - Kaltofen et Shoup (1997), Algorithme E]. Une variante de cette méthode est étudiée dans [152 - Shoup (1994), Section 4]. Malheureusement, cette méthode a deux désavantages importants. D'une part, la factorisation peut s'avérer difficile et d'autre part, cette méthode produit des polynômes quelconques. En effet, pour faciliter les calculs avec ces polynômes, il est intéressant d'obtenir des polynômes creux, c'est-à-dire avec très peu de coefficients non nuls. Dans ce cas, de même que pour les racines primitives, la recherche systématique peut s'avérer plus efficace en pratique, comme nous allons le voir.

3.3.2 POLYNÔMES IRRÉDUCTIBLES CREUX DE $\mathtt{GF}(q^r)$

Nous commençons par présenter un test d'irréductibilité, dû à Ben-Or [10 - Ben-Or (1981)], qui est simplement une variante de l'algorithme de factorisation de polynômes de Cantor et Zassenhaus [28 - Cantor et Zassenhaus (1981)], simplifiée pour nos besoins.

Algorithme 3.3.4 *Test-Irréductibilité*

Entrées : – Un polynôme, $P \in \mathtt{GF}(q)[X]$.

Sorties : – "Oui" si P est irréductible, "Non" sinon.

1 : **Si** pgcd $(P, P') \neq 1$ **Alors** Renvoyer "Non". {P est-il sans carré ?}

2 : $W = X$

3 : **Pour** $d = 1$ **jusqu'à** $\frac{\mathrm{d}^\circ P}{2}$ **Faire**

4 : $\quad W \equiv W^q[P]$

5 : $\quad$ **Si** pgcd $(W - X, P) \neq 1$ **Alors** Renvoyer "Non".

Le test de V. Shoup [152 - Shoup (1994)], par exemple, est plus efficace quand le polynôme est irréductible. En effet, il n'est pas nécessaire de calculer toutes les puissances successives W^q, il suffit de vérifier les $X^{q^{\frac{n}{p_i}}}$ pour tous les p_i facteurs premiers distincts de n. Néanmoins, si il faut tester plusieurs polynômes avant de tomber sur un polynôme irréductible, cet algorithme sera, en général, plus performant puisqu'il découvrira que les précédents polynômes sont réductibles avec des puissances moins grandes, et donc avec moins de calculs.

Ainsi, en utilisant ce test, il est possible de tirer des polynômes au hasard et d'avoir une bonne chance de tomber sur un polynôme irréductible. En effet, si $m_r(p)$ est le nombre de polynômes unitaires irréductibles de degré r dans $\mathbb{Z}/\mathrm{p}\mathbb{Z}[X]$, alors comme $X^{p^r} - X$ est le produit de tous les polynômes irréductibles

de degré divisant r, on obtient d'après [44 - Demazure (1997), Corollaire 8.19] et [96 - Koblitz (1987), Proposition II.1.8] :

$$\frac{1}{r}(p^r - p^{\lfloor \frac{r}{2} \rfloor + 1}) \leq m_r(p) = \frac{1}{r}\big(p^r - \sum_{d|r;d<r} d m_d(p)\big) \leq \frac{1}{r}p^r \qquad (3.2)$$

ce qui montre que parmi les polynômes, tirés au hasard, de degré r, environ un sur r est irréductible.

Nous proposons donc l'algorithme hybride suivant produisant de préférence un polynôme irréductible creux, un facteur de polynôme cyclotomique ou un polynôme irréductible au hasard, si les deux précédents s'avèrent trop difficile à calculer. Par exemple, Coppersmith propose de prendre des polynômes du type $X^r + g(X)$ avec $g(X)$ au hasard de faible degré par rapport à r [39 - Coppersmith (1984)].

Algorithme 3.3.5 *Polynôme-Irréductible*

Entrées : – Un corps fini $\mathtt{GF}(q)$.
– Un entier $r > 0$.

Sorties : – Un polynôme irréductible de $\mathtt{GF}(q)[X]$, de degré r.

1 : **Pour tout** $a \in \mathtt{GF}(q), a \neq 0$ **Faire**
2 : **Si** *Test-Irréductibilité* $(X^r + a)$ == "Oui" **Alors** Renvoyer $X^r + a$
3 : **Pour** $d = 2$ **jusqu'à** $r - 1$ **Faire**
4 : **Pour tout** $a \in \mathtt{GF}(q), a \neq 0, b \in \mathtt{GF}(q), b \neq 0$ **Faire**
5 : **Si** *Test-Irréductibilité* $(X^r + bX^d + a)$ == "Oui" **Alors** Renvoyer $X^r + bX^d + a$
6 : Soit $n \leq p^r - 1$, avec $pgcd(n, p) = 1$ et $\phi(n)$ minimal tel que l'ordre de p dans $\mathbb{Z}/n\mathbb{Z}$ soit r.
7 : **Si** $\phi(n)$ est petit **Alors**
8 : Renvoyer un facteur de $\Theta_n(X)$ {Par [92 - Kaltofen et Shoup (1997), Algorithme E] }
9 : **Sinon**
10 : **Répéter**
11 : Sélectionner P, unitaire de degré r, au hasard dans $\mathtt{GF}(q)[X]$.
12 : **Jusqu'à ce que** *Test-Irréductibilité* (P) == "Oui"
13 : Renvoyer P.

3.3.3 RACINES PRIMITIVES CREUSES DE $\mathtt{GF}(q^r)$

Nous savons maintenant trouver un polynôme irréductible de degré r quelconque dans $\mathtt{GF}(q)$. Il reste à calculer un générateur de $\mathtt{GF}(q^r)$. Nous utilisons de nouveau un algorithme probabiliste. Nous montrons tout d'abord un algorithme testant si un polynôme est générateur dans $\mathtt{GF}(q)[X]$. Cet algorithme est similaire à celui développé pour les racines primitives dans $\mathbb{Z}$.

Algorithme 3.3.6 *Test-Polynôme-Générateur*

Entrées : – Un polynôme $A \in \mathtt{GF}(q)[X]$.

– Un polynôme F irréductible de degré d dans $\mathtt{GF}(q)[X]$.

Sorties : – Oui, si A est générateur de l'extension $\mathtt{GF}(q)[\mathrm{X}]/\mathrm{F}$; Non dans le cas contraire.

1 : **Si** A et F ne sont pas premiers entre eux **Alors** Renvoyer "Non".

2 : **Pour tout** p, premier et divisant $q^d - 1$ **Faire** {Factorisation de $q^d - 1$}

3 : **Si** $A^{\frac{q^d-1}{p}} \equiv 1[F]$ **Alors** Renvoyer "Non". {Calcul récursif par carrés}

4 : Renvoyer "Oui".

Un algorithme cherchant au hasard un générateur, une fois le corps construit, est alors aisé. En outre, V. Shoup [151 - Shoup (1992), Théorème 1] a montré que l'on peut restreindre l'ensemble de recherche à des polynômes de faible degré ($\mathcal{O}(\ln(n))$). Toutefois, toujours par souci d'obtenir des polynômes creux, nous montrons à l'aide de la proposition suivante qu'il est possible de rapidement trouver, sur un corps premier, un polynôme irréductible pour lequel X est une racine primitive. Un tel polynôme est appelé **X-Irréductible**. Ainsi, les calculs des indices pour notre représentation en seront grandement accélérés, puisqu'il suffira de multiplier par X !

Proposition 3.3.7

Soit un corps fini premier $\mathbb{Z}/_p\mathbb{Z}$. Le nombre de polynômes irréductibles, F, de degré $r \geq 2$ tels que X soit un générateur de $\mathbb{Z}/_p\mathbb{Z}[X]/_F$ est minoré par :

$$\left(\frac{p^r-1}{r}\right)\left(\frac{2\omega(p,r)}{2e^C\omega^2(p,r)+5}\right)$$

où $\omega(p,r) = \ln(\ln(p^r-1)) < \ln(r)+\ln(\ln(p))$ et C est la constante d'Euler.

Preuve de 3.3.7 : Tous les facteurs irréductibles de Θ_{p^r-1} dans $\mathbb{Z}/_p\mathbb{Z}$ sont de degré r. Soit F un tel facteur, et soit ξ une racine de F. Nous considérons deux corps. Le premier est $\mathbb{Z}/_p\mathbb{Z}[X]/_F$. Le second est une extension cyclotomique de $\mathbb{Z}/_p\mathbb{Z}$ de niveau p^r-1 ; c'est-à-dire une plus petite extension de $\mathbb{Z}/_p\mathbb{Z}$ contenant une racine primitive $(p^r-1)^{\text{ième}}$ de l'unité. Ce corps, $R_{p^r-1}(\mathbb{Z}/_p\mathbb{Z})$, est obtenu par l'adjonction de ξ à $\mathbb{Z}/_p\mathbb{Z}$. D'après [44 - Demazure (1997), Proposition 8.22], $R_{p^r-1}(\mathbb{Z}/_p\mathbb{Z})$ est un corps à p^r éléments. Il est donc isomorphe à $\mathbb{Z}/_p\mathbb{Z}[X]/_F$. Ainsi, comme le polynôme minimal de ξ est F, calculer les puissances successives de ξ sachant $F(\xi)=0$ dans $R_{p^r-1}(\mathbb{Z}/_p\mathbb{Z})$, revient à calculer les puissances successives de X dans $\mathbb{Z}/_p\mathbb{Z}[X]/_F$. Or, ξ est une racine primitive $(p^r-1)^{\text{ième}}$ de l'unité, comme racine de Θ_{p^r-1}. Donc, aucune puissance de ξ inférieure à p^r-1 ne peut valoir 1 dans $R_{p^r-1}(\mathbb{Z}/_p\mathbb{Z})$. Alors, aucune puissance de X inférieure à p^r-1 ne peut valoir 1 dans $\mathbb{Z}/_p\mathbb{Z}[X]/_F$. Ce qui veut dire que X est un générateur de $\mathbb{Z}/_p\mathbb{Z}[X]/_F$.

Donc, dans $\mathbb{Z}/_p\mathbb{Z}[X]$, les facteurs de Θ_{p^r-1} *sont* les polynômes irréductibles qui ont X comme racine primitive. Il reste à les dénombrer.

Nous savons par la propriété 3.3.2, page 51 que le degré de Θ_{p^r-1} est $\phi(p^r-1)$. En utilisant la borne 3.1, page 49, nous pouvons donc minorer cette valeur par

$$\big(p^r-1\big)/\big(e^C\ln(\ln(p^r-1))+\frac{5}{2\ln(\ln(p^r-1))}\big).$$

La relation annoncée s'en déduit par le fait qu'il y a $\frac{\phi(p^r-1)}{r}$ polynômes de degré r facteurs de Θ_{p^r-1}, tous distincts. □

D'une part, si $p^r=4$, alors $\Theta_{p^r-1}=\Theta_3=X^2+X+1$ est le polynôme X-irréductible de degré 2 de $\mathbb{Z}/_2\mathbb{Z}[X]$. Et, d'autre part, $\frac{2e^C\omega^2(p,r)+5}{2\omega(p,r)}$ est inférieur à 12 pour $5 \leq p^r \leq 2^{32}$. Donc, pour les corps finis de taille comprise entre 4 et 2^{32}, un algorithme recherchant un polynôme X-irréductible au hasard nécessite moins de $12r$ essais en moyenne. Nous donnons maintenant l'algorithme final pour trouver un polynôme irréductible ayant X comme racine. C'est une modification de l'algorithme 3.3.5, page 53 utilisant le test de racine primitive :

Algorithme 3.3.8 *Polynôme-X-Irréductible*

Entrées : – Un corps fini $\mathbb{Z}/p\mathbb{Z}$.
– Un entier $r \geq 2$.

Sorties : – Un polynôme irréductible F dans $\mathbb{Z}/p\mathbb{Z}[X]$, de degré r, ayant X comme racine primitive.

1 : **Pour tout** $a \in \mathbb{Z}/p\mathbb{Z}$ **Faire**
2 : $F = X^r + a$
3 : **Si** *Test-Irréductibilité* (F)==“Oui”
et *Test-Polynôme-Générateur* (X, F)==“Oui” **Alors**
4 : Renvoyer F.
5 : **Pour** $d = 2$ **jusqu'à** $r - 1$ **Faire**
6 : **Pour tout** $a \in \mathbb{Z}/p\mathbb{Z}, b \in \mathbb{Z}/p\mathbb{Z}, b \neq 0$ **Faire**
7 : $F = X^r + bX^d + a$
8 : **Si** *Test-Irréductibilité* (F) == “Oui”
et *Test-Polynôme-Générateur* (X, F)==“Oui” **Alors**
9 : Renvoyer $X^r + bX^d + a$.
10 : $\phi_{pr} = \phi(p^r - 1)$.
11 : Faire au plus $12r$ tentatives probabilistes de factorisation modulo p de $\Theta_{p^r-1}(X)$.
12 : **Si** un facteur F de $\Theta_{p^r-1}(X)$ a été trouvé **Alors**
13 : Renvoyer F.
14 : **Sinon**
15 : **Répéter**
16 : Sélectionner F, unitaire de degré r, au hasard dans $\mathbb{Z}/p\mathbb{Z}[X]$.
17 : **Jusqu'à ce que** *Test-Irréductibilité* (F)==“Oui”
et *Test-Polynôme-Générateur* (X, F)==“Oui”
18 : Renvoyer F.

3.4 EFFICACITÉ DE L'X-IRRÉDUCTIBILITÉ

Nous avons testé cet algorithme, sur un Pentium cadencé à 333 MHz, avec 128 Mo de mémoire, pour trouver un polynôme irréductible de degré $r \geq 2$ dans $\mathbb{Z}/p\mathbb{Z}$,

avec p et r tels que la caractéristique p est inférieure à 2^{13} et la taille de l'extension q est inférieure à 2^{26}. La limite de 2^{26} est justifiée par le fait qu'au-delà, la mémoire nécessaire pour stocker la table des successeurs (de taille q) aurait dépassé les 256 Mo. Les calculs ont été effectués plusieurs fois et la valeur moyenne du

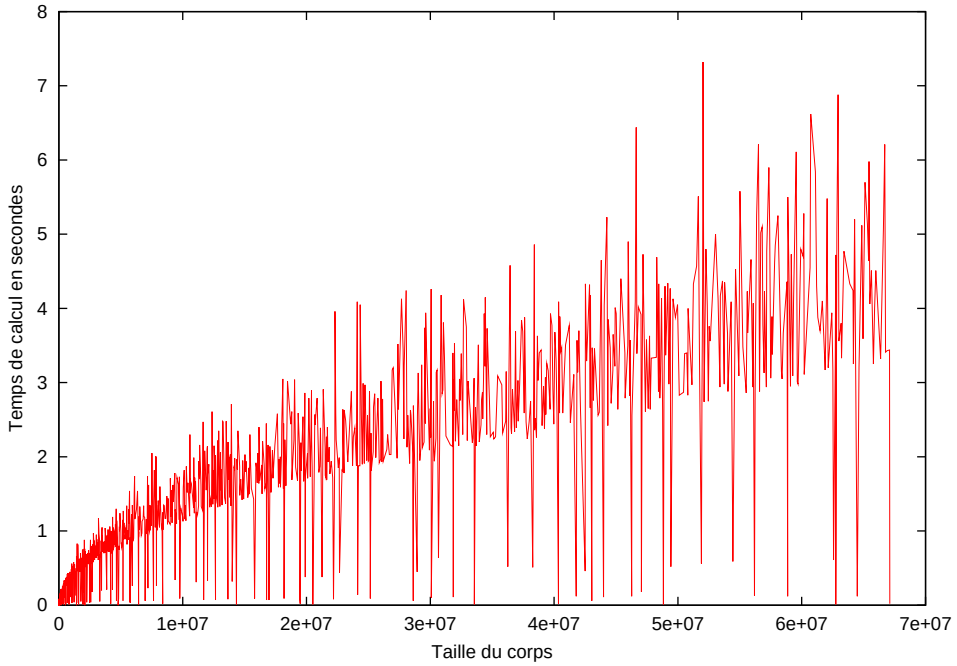

FIGURE 3.2 – Calcul d'un polynôme irréductible ayant X comme générateur

temps d'exécution a été retenue. Sur la figure 3.2, page 57 on peut voir que pour ces cas là, le temps de calcul n'a jamais dépassé 8 secondes.

En fait, le gain est intéressant surtout au niveau de la génération des tables. C'est là que la différence entre polynômes irréductibles plus racines primitives aléatoires et polynômes creux X-Irréductibles est importante. Nous avons testé la génération des trois tables (une pour les successeurs, une pour chaque conversion), pour des corps de taille inférieure à 2^{23}, sachant que la mémoire nécessaire pour le stockage des tables de `int` est déjà de $4 * 3 * 2^{23} > 100$ Mo. Nous voyons sur les figures 3.4 et 3.3, page 58, qu'il y a gain pour chaque corps et que ce gain peut atteindre un facteur 8.5. En outre, le temps d'exécution de cette génération ne dépasse pas une minute pour chacun de ces exemples dans le cas du polynôme X-irréductible, et ne dépasse pas 2 secondes pour la moitié des cas.

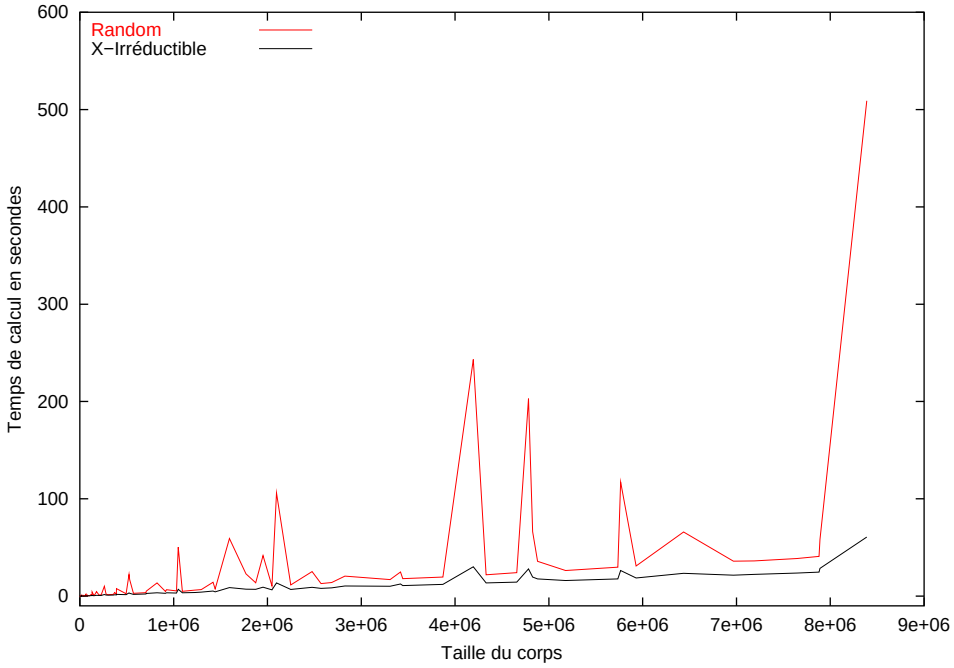

FIGURE 3.3 – Génération des tables de successeurs et de conversions

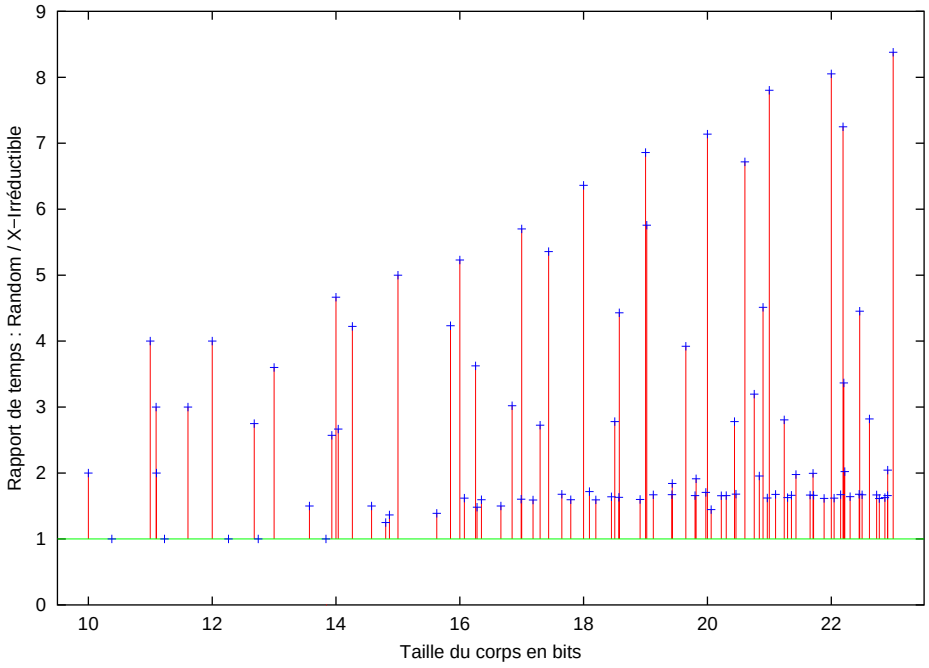

FIGURE 3.4 – Rapport des temps de génération des tables

4

ARITHMÉTIQUE DES CORPS PREMIERS

« Mathematik ist die Königin von Wissenschaften und Arithmetik die Königin von Mathematik »

Carl Friedrich Gauß

Sommaire

Nous étudions dans ce chapitre l'efficacité de l'utilisation de tables précalculées pour l'implémentation des corps premiers. Pour cela, nous avons implémenté quatre classes C++ : *Zpz*, *GFq*, *GFqTab* et *ZpzLong*. Toutes utilisent les entiers longs (`long int`). Dans la suite, p sera le modulo concerné dans l'implémentation de $\mathbb{Z}/_{p}\mathbb{Z}$. *Zpz* est l'implémentation classique avec division par p. *GFqTab* utilise les racines primitives et quinze tables de taille p. *GFq* utilise aussi les racines primitives et seulement trois tables de taille p. *Zpzlong* est simplement une arithmétique sur les entiers sans aucun rapport avec p ; les résultats obtenus avec cette arithmétique sont simplement des références. Les deux implémentations *GFqTab* et *GFq* fonctionnent aussi pour des corps de Galois de taille une puissance d'un nombre premier ; dans ce cas, seule la génération des tables diffère.

4.1 IMPLÉMENTATIONS

Nous présentons ici différentes méthodes pour implémenter sept des opérations de base : l'addition, la soustraction, la négation, la multiplication, la division, une multiplication suivie d'une addition ($r = a * x + y$) ou *AXPY*, une multiplication et une addition en place ($r = a * x + r$) ou *AXPYIN*. Dans le cadre de l'algèbre linéaire, il semble que ces deux dernières opérations soient les plus utilisées.

4.1.1 CLASSIQUE AVEC DIVISION

Zpz est une implémentation classique des entiers modulaires :

- L'addition est une addition signée sur les entiers suivie d'un test. Si le test montre que le résultat est supérieur au modulo, le modulo est soustrait du résultat.
- La soustraction est implémentée de la même façon.
- La négation est une simple soustraction du modulo.
- La multiplication est une multiplication machine suivie d'un modulo machine.
- La division est une inversion implémentée par l'algorithme d'Euclide du pgcd étendu puis une multiplication suivie d'un modulo ; en effet, si b et p sont premiers entre eux alors le théorème de Bézout donne l'existence de u et v tels que $1 = u * b + v * p$ et donc $u * b \equiv 1[p]$.
- *AXPY* est une multiplication et une addition suivies d'un seul modulo machine.

Code 4.1.1 Arithmétique *Zpz*

```
ZPZ_ADD(r,a,b,p) { r = (a+b); r = (r < p ? r : r-p);}
ZPZ_SUB(r,a,b,p) { r = (a-b); r = (r < 0 ? r+p : r);}
ZPZ_NEG(r,a,p)   { r = (a == 0 ? 0 : p-a); }
ZPZ_MUL(r,a,b,p) { r = (a*b); r= (r>=p ? r % p : r);}
ZPZ_DIVEXACT(r,a,b,p) { r = (b == 1 ? a : a/b);
ZPZ_DIV(r,a,b,p) { Rep g, u, v; Bezout(g, u, v, b, p);
                   ZPZ_DIVEXACT(r,a,g,p);
                   ZPZ_MUL(r,r,u,p);}
ZPZ_AXPY(r,a,b,c,p) { r = (a*b+c); r = (r<p ? r : r % p);}
ZPZ_AXPYIN(r,a,b,p) { r += (a*b); r = (r<p ? r : r % p);}
```

Pour que les résultats soient corrects, la valeur intermédiaire du AXPY ne doit pas dépasser la taille du mot machine. Si la taille de mot machine est 2^m, la taille de la mantisse signée est de $m-1$ et donc la taille du corps ne doit pas dépasser $2^{\frac{m-1}{2}}-1$. En particulier si le mot machine est codé sur 32 bits (respectivement 64 bits), alors la taille du corps ne peut pas dépasser 46339 (respectivement 3037000498). En outre, cette implémentation reste valide même si le modulo n'est pas premier, tant que les divisions se font par des éléments inversibles.

4.1.2 AVEC RACINES PRIMITIVES

GFq est une implémentation de la représentation par l'indice multiplicatif d'une racine primitive, comme décrit au chapitre précédent. Notons g une racine primitive du corps $\mathtt{GF}(q)$; l'indice de -1 est précalculé et noté i_{-1}, tel que $-1_{\mathtt{GF}(q)} = g^{i_{-1}}$; enfin, l'indice de 1 est codé par $\overline{q} = q-1$ et est donc noté $i_{\overline{q}}$. Cette implémentation utilise au moins une table prédéfinie, la table des successeurs, t_plus1 (il est possible d'utiliser deux autres tables, pour accélérer les conversions). La table des successeurs est précalculée de telle sorte que $\forall i, t\_plus1[i] = j$ si $1 + g^i = g^j$, avec un cas particulier pour l'indice de -1, $t\_plus1[i_{-1}] = 0$.

Code 4.1.2 Arithmétique *GFq*

```
GFQ_ADD(r,i,j) { // g^i + g^j = g^j(1 + g^{i-j})
  if (j==0) r=i;
  else if (i==0 ) r=j;
  else { r = i-j;
         r = t_plus1[ (r>0)? r : r + i_{\overline{q}}  ];
```

```
        GFQ_MUL(r,r,j);
} }

GFQ_SUB(r,i,j) { // g^i - g^j = g^i(1+g^{j-i}/(-1))
  if (i==0) GFQ_NEG(r,j);
  else if (j==0) r=i;
  else { r = j-i-i_{-1};
         r = (r>0)? r : r + i_{q};
         r = t_plus1[ (r>0)? r : r + i_{q} ];
         GFQ_MUL(r,r,i);
} }

GFQ_NEG(r,i) { // -g^i = g^i/(-1) = g^{i-i_{-1}}
  if ( i==0 ) r=0;
  else { r = i - i_{-1};
         r = (r>0)? r : r + i_{q};
} }

GFQ_MUL(r,i,j) {  // g^i * g^j = g^{i+j}
  if ( (i==0) || (j==0) ) r =0;
  else { r = i + j - i_{q};
         r = (r>0)? r : r + i_{q};
} }

GFQ_DIV(r,i,j) { // g^i / g^j = g^{i-j}
  if ( i==0 ) r=0;
  else { r = i - j;
         r = (r>0)? r : r + i_{q};
} }

GFQ_AXPY(r,i_1,i_2,j) { // g^{i_1} * g^{i_2} + g^j = g^j(1+g^{i_1+i_2-j})
  if (j==0) GFQ_MUL(r,i_1,i_2);
  else if ((i_1==0) || (i_2==0)) r=j;
  else {
    r = i_1 + i_2 - j - i_{q};
    r = (r<0)? r + i_{q} : r;
    r = t_plus1[(r>0)? r : r + i_{q}];
    if (r) { r = r + j - i_{q}
             r = (r>0)? r : r + i_{q};
} } }
```

En pratique, la table t_plus1 peut renvoyer un résultat auquel on a pré-soustrait

$i_{\overline{q}}$. Ainsi, on gagne une opération pour le calcul de GFQ_ADD et GFQ_AXPY. Dans ce cas, la taille du corps n'est plus limitée par la taille du mot machine, mais par l'espace mémoire disponible. En effet, nous avons vu qu'un corps de taille 2^{25} nécessite déjà 128 Mo. En outre, cette implémentation est valable non seulement pour les nombres premiers, mais pour tous les corps finis, pourvu qu'une représentation par indices ait été construite.

4.1.3 TOTALEMENT TABULÉE

GFqTab est une autre implémentation utilisant les racines primitives. La représentation est la même que pour *GFq*, mais plusieurs tables ont été précalculées pour éviter de faire des tests. De même que précédemment, on note $\overline{q} = q - 1$. La première table, t_mul, est une table des correspondances entre $i + j$ et $i + j \mod \overline{q}$. Comme $i + j$ peut prendre des valeurs entre 0 et $2\overline{q}$, nous avons choisi de coder 0 non plus par 0, mais par $2\overline{q}$. Ainsi, la table peut être construite pour toutes les valeurs de $k = i + j$ de la manière suivante :

- $t\_mul[k] = k$ pour $0 \leq k < \overline{q}$.
- $t\_mul[k] = k - \overline{q}$ pour $q - 1 \leq k < 2\overline{q}$.
- $t\_mul[k] = 2\overline{q}$ pour $2\overline{q} \leq k \leq 4\overline{q}$.

De manière analogue, $i - j$ varie entre $-\overline{q}$ et $\overline{q}$; la précédente table peut aussi servir dans ce cas, il suffit de décaler les indices par $\overline{q}$: $t\_div = \&t\_mul[\overline{q}]$. Il faut aussi ajouter les valeurs entre $-2\overline{q}$ et $-\overline{q}$, puisque, comme i ou j peuvent valoir $2\overline{q}$ (notre codage du zéro), en fait $i - j$ varie entre $-2\overline{q}$ et $2\overline{q}$:

- $t\_mul[k] = 2\overline{q}$ pour $-\overline{q} \leq k < 0$.

On a donc, au total, une première table, de taille $5q$. Pour la négation, on utilise de même un décalage par l'indice de -1 : $t\_neg = \&t\_mul[i_{-1}]$. Enfin, on construit de la même manière une table des successeurs de taille $4q$ (t_plus1) et une table des prédécesseurs de taille $4q$ (t_moins1). Avec les deux tables de conversion, chacune de taille q, cela fait un espace mémoire total nécessaire de $15q$ entiers.

Code 4.1.3 Arithmétique *GFqTab*

```
TAB_ADD(r,i,j) {r = t_mul[i + t_plus1[j - i] ];}
TAB_SUB(r,i,j) {r = t_mul[i + t_moins1[j - i] ];}
TAB_NEG(r,i)   {r = t_neg[i];}
TAB_MUL(r,i,j) {r = t_mul[i + j];}
TAB_DIV(r,i,j) {r = t_div[i - j];}
TAB_AXPY(r,i,j,k) {r = t_mul[k + t_plus1[t_mul[i+j] - k]];}
```

De même que l'implémentation précédente, cette implémentation est valable pour tous les corps finis. La taille de corps est toujours limitée par l'espace mémoire disponible, mais la nécessité de 15 longueurs de table au lieu de 3, implique une taille maximale au moins divisée par 5, par rapport à l'implémentation précédente.

4.1.4 RÉFÉRENCE

ZpzLong est une implémentation de l'arithmétique avec les opérations de base uniquement. Les résultats ici ne veulent rien dire. Il s'agit seulement de fournir une référence externe des opérations sur un type machine de base.

Code 4.1.4 Arithmétique *ZpzLong*

```
LONG_ADD(r,a,b,p)    { r = a + b; }
LONG_SUB(r,a,b,p)    { r = a - b; }
LONG_NEG(r,a,p)      { r = - a; }
LONG_MUL(r,a,b,p)    { r = a * b; }
LONG_DIV(r,a,b,p)    { r = a / b; }
LONG_AXPY(r,a,b,c,p) { r = a * b + c; }
LONG_AXPYIN(r,a,b,p) { r += a * b; }
```

4.2 RÉSULTATS EXPÉRIMENTAUX

4.2.1 COMPARAISON AVEC ALP ET NTL

Notre but est de comparer différentes implémentations d'arithmétique modulaire. Nous présentons tout d'abord une comparaison générale avec l'arithmétique classique de deux logiciels pour montrer que notre implémentation est pertinente.

Le premier logiciel, ALP [116 - Mourrain et Prieto (2000)], implémente les opérations de la manière la plus simple, par l'utilisation d'une division (%) pour

chaque opération.

Le deuxième logiciel, NTL [153 - Shoup (2000)], utilise des méthodes particulières pour chaque opération. Dans le cas de l'addition et de la soustraction, il commence par calculer $r = a + b - p$ puis il ajoute ensuite p si r est négatif. Mais ce dernier test utilise le fait que les nombres négatifs sont codés en complément à la base, le signe étant stocké dans le bit de poids fort. Ainsi, r, décalé de $m - 1$ bits à droite (en remplaçant les bits manquants par le bit de signe), vaut 0 si r est positif et -1 ($111111\ldots111$) sinon. Ensuite, en effectuant un *ET* bit à bit entre 0 et p ou entre -1 et p, on obtient 0 si r est positif, et p si r est négatif. Il suffit ensuite d'ajouter cette dernière valeur à r pour obtenir une valeur modulaire comprise entre 0 et p.

Code 4.2.1 Addition modulaire dans NTL

```
inline long AddMod(long a, long b, long p) {
   long res = a + b;
#if (NTL_ARITH_RIGHT_SHIFT && defined(NTL_AVOID_BRANCHING))
   res -= p;
   res += (res >> (NTL_BITS_PER_LONG-1)) & p;
   return res;
#else
   if (res >= p) return res - p;
   else return res;
#endif
}
```

Il est possible de changer cela dans NTL pour effectuer un test classique à la place du décalage, comme dans notre implémentation ; toutefois, les performances entre ces deux méthodes ne sont que très légèrement différentes. D'autre part, pour la multiplication, les calculs se font non pas en utilisant les entiers machine, mais en utilisant l'arithmétique flottante. L'inverse de p est précalculé avec une double précision, le calcul modulaire de $a*b$ devient alors $a*b \mod p = a*b - \lfloor \frac{a*b}{p} \rfloor * p$, où les calculs intermédiaires se font avec l'arithmétique flottante et les valeurs entières sont calculées par conversion.

Ces deux bibliothèques ainsi que nos implémentations ont été compilées par le même compilateur, gcc version 2.95.1 19990816, et avec les mêmes options de compilations.

L'élément de comparaison choisi est le nombre d'opérations effectuées. Ainsi, *Mop/s* désigne un million d'opérations arithmétiques par seconde. Sur les figures

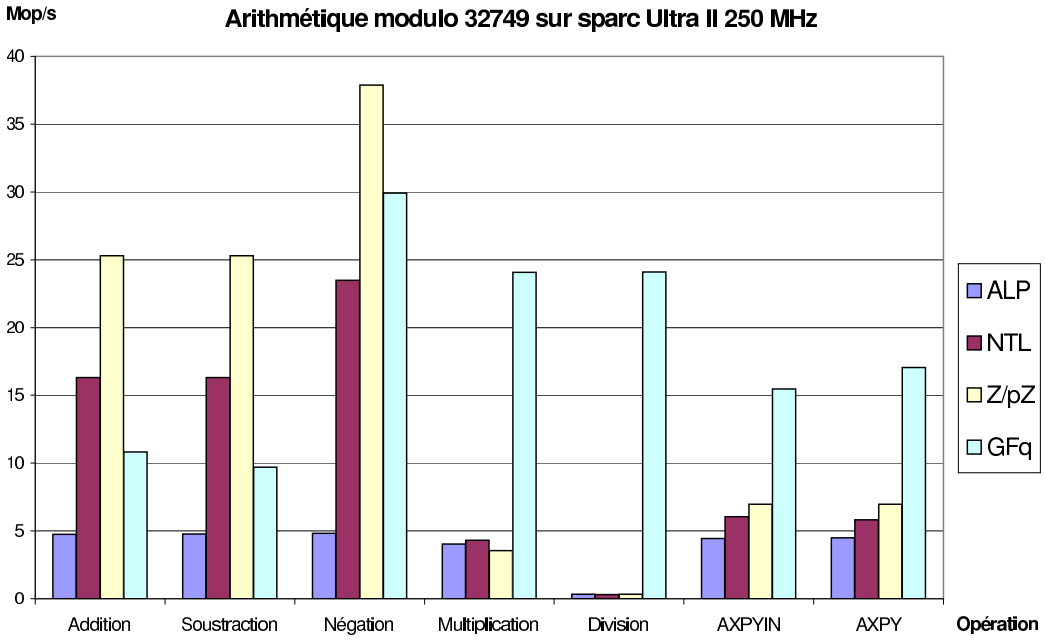

FIGURE 4.1 – Comparaison avec ALP et NTL, modulo 32479

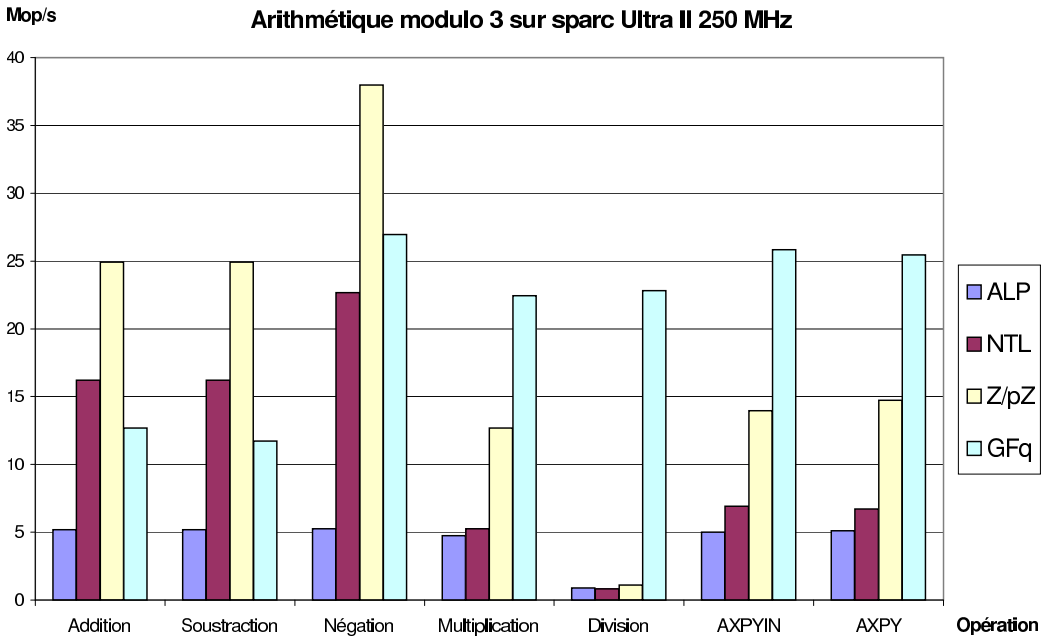

FIGURE 4.2 – Comparaison avec ALP et NTL, modulo 3

4.1, page 66 et 4.2, page 66, ALP et NTL désignent les implémentations de ces deux bibliothèques, $\mathbb{Z}/_{p}\mathbb{Z}$ désigne notre implémentation *Zpz*, de la section 4.1.1, page 60, et GFq désigne notre implémentation *GFq*, de la section 4.1.2, page 61. Nous voyons tout d'abord, sur la figure 4.1, page 66, avec un modulo de taille moyenne, que l'implémentation la plus simple donne environ les mêmes performances pour tous les types d'opérations (sauf la division). En effet, le coût dominant est celui de la division par le modulo. Ensuite, nous voyons que la multiplication de NTL est légèrement meilleure que ALP et *Zpz* : sur ultra sparc, l'arithmétique flottante est plus rapide que l'arithmétique entière. Pour toutes les autres opérations, *Zpz* est meilleure que NTL. Pour les deux *AXPY*, cela vient du fait que NTL ne propose pas cette méthode, donc deux appels de fonction sont effectués au lieu d'un seul dans notre cas. Enfin, pour l'addition, la soustraction et la négation, la seule différence entre les deux implémentations est que nous avons défini une macro alors que NTL fait un appel de fonction *inline*. Le compilateur ne doit pas réussir à passer outre l'appel de fonction supplémentaire dans le cas de NTL.

Finalement, en comparant ces trois bibliothèques avec l'implémentation par racines primitives, nous voyons que la multiplication et la division de cette dernière sont quasiment aussi rapides que l'addition et la soustraction dans le cas classique. Au contraire, l'addition et la soustraction par racines primitives sont deux fois plus rapides que la multiplication classique. C'est pourquoi la fonction *AXPY* est bien meilleure avec racines primitives. En outre, dans le cas de modulo très petits, on remarque, sur la figure 4.2, page 66, que ces différences sont encore plus marquées, grâce aux effets de mémoire cache. Nous étudions ceux-ci plus en détails dans la section suivante.

4.2.2 Quelle arithmétique modulaire ?

Dans cette section nous présentons le nombre de millions d'opérations par seconde obtenus pour différents corps, de taille 3 à 65521. Nous regardons les performances pour la fonction *AXPY*, c'est-à-dire que nous effectuons cette opérations plusieurs fois, pour tous les triplets possibles, et divisons le nombre total d'AXPY par la moitié du temps obtenu.

Il est toutefois utile de mesurer ces performances dans un contexte où plusieurs de ces opérations sont effectuées de concert. Le produit de matrices classique nous a semblé être un bon moyen de tester cette particularité. L'algorithme est la triple boucle classique sur deux matrices A et B, quelconques, mais de dimensions fixées :

Code 4.2.2 Produit de matrices

```
1   void matrix_product( Modulo* C, const Modulo* A,
                         const Modulo* B, int dim )
    {
       for (int i=0; i<dim; ++i)
5      {
          const Modulo* Ai = &A[i*dim];
          for (int j=0; j<dim; ++j)
          {
             const Modulo* Bj = &B[j];
10            Modulo sum =0;
              for (int k=0; k<dim; ++k, Bj += dim)
                 MODULO_AXPY(sum, Ai[k],*Bj, sum, P);
              C[i*dim+j] = sum;
           }
15     }
     }
```

Les résultats comparés des différentes implémentations des corps finis dépendant de l'architecture, nous présentons des mesures pour des matrices 64×64 sur un IBM rs6000 cadencé à 100 MHz, un Sun ultrasparc cadencé à 133 MHz, un Intel Pentium II cadencé à 333 MHz et un Dec alpha cadencé à 400 MHz. Sur les figures 4.3, page 70, 4.4, page 71, 4.5, page 72 et 4.6, page 73, montrant ces performances, nous pouvons faire les remarques suivantes :

(1) La première remarque, générale, est que, pour toutes les architectures testées, le nombre d'opérations obtenues dans le produit de matrices est bien supérieur à celui obtenu pour l'AXPY. Cela vient de l'utilisation de caches mémoire par les processeurs. En effet, comme les matrices sont petites, leurs éléments sont stockés dans les caches et les accès sont alors de coût très faible. Cela induit qu'un algorithme matriciel implémenté par une découpe en petits blocs (de taille 64×64, par exemple) permettra aux processeurs d'effectuer plus d'opérations par seconde que le même algorithme implémenté sans bloc.

(2) Sur l'IBM rs6000, le coût d'une multiplication et d'une addition d'entiers machine est sensiblement le même. Ainsi, il n'y a quasiment pas de différence entre *Zpz* et *GFq*. D'autre part, l'implémentation tabulée tire beaucoup mieux parti des performances de la machine, et est même quasiment aussi performante pour le produit de matrices sur petits corps que la version de référence (*ZpzLong*, n'effectuant qu'*une seule* opération machine).

(3) Au contraire, sur les machines SUN, les deux implémentations avec racines primitives peuvent même être meilleures que cette référence ! Cela vient des mauvaises performances de la multiplication machine d'entiers sur ces processeurs. On voit de plus que, pour le produit de matrices, l'implémentation par table devient moins bonne que l'implémentation *GFq* quand le corps devient trop grand, sans doute à cause de la taille des mémoires cache des machines qui ne permettent plus de stocker toutes les opérations.

(4) Avec les processeurs Intel, les performances n'atteignent pas la référence, mais l'implémentation tabulée reste compétitive, cette fois-ci, jusqu'à une taille de corps plus importante que pour les machines SUN, les caches étant sans doute plus grands. Néanmoins, comme ces caches sont plus grands, ces différences sont moins importantes.

(5) Le même comportement, enfin, est observé sur la machine Dec, avec ici une taille de mémoire cache sans doute proche de celle des SUN.

En conclusion, nous pouvons observer deux effets de cache différents : l'un sur les matrices, quand nous effectuons le produit de matrices, l'autre sur les tables des implémentations. Il est clair que pour de très petits corps l'implémentation tabulée est bien meilleure que toutes les autres. Cela est particulièrement flagrant pour le produit de matrices. Ensuite, dans le cas de l'AXPY, pour toutes les architectures, mis à part l'IBM, alors que le corps devient plus grand, l'autre implémentation avec racines primitives est préférable. Enfin, nous voyons l'intérêt des méthodes avec racines primitives, tant que l'espace mémoire nécessaire reste raisonnable ; en effet, ces méthodes permettent d'avoir des performances plus de 10 fois supérieures aux performances des méthodes avec division.

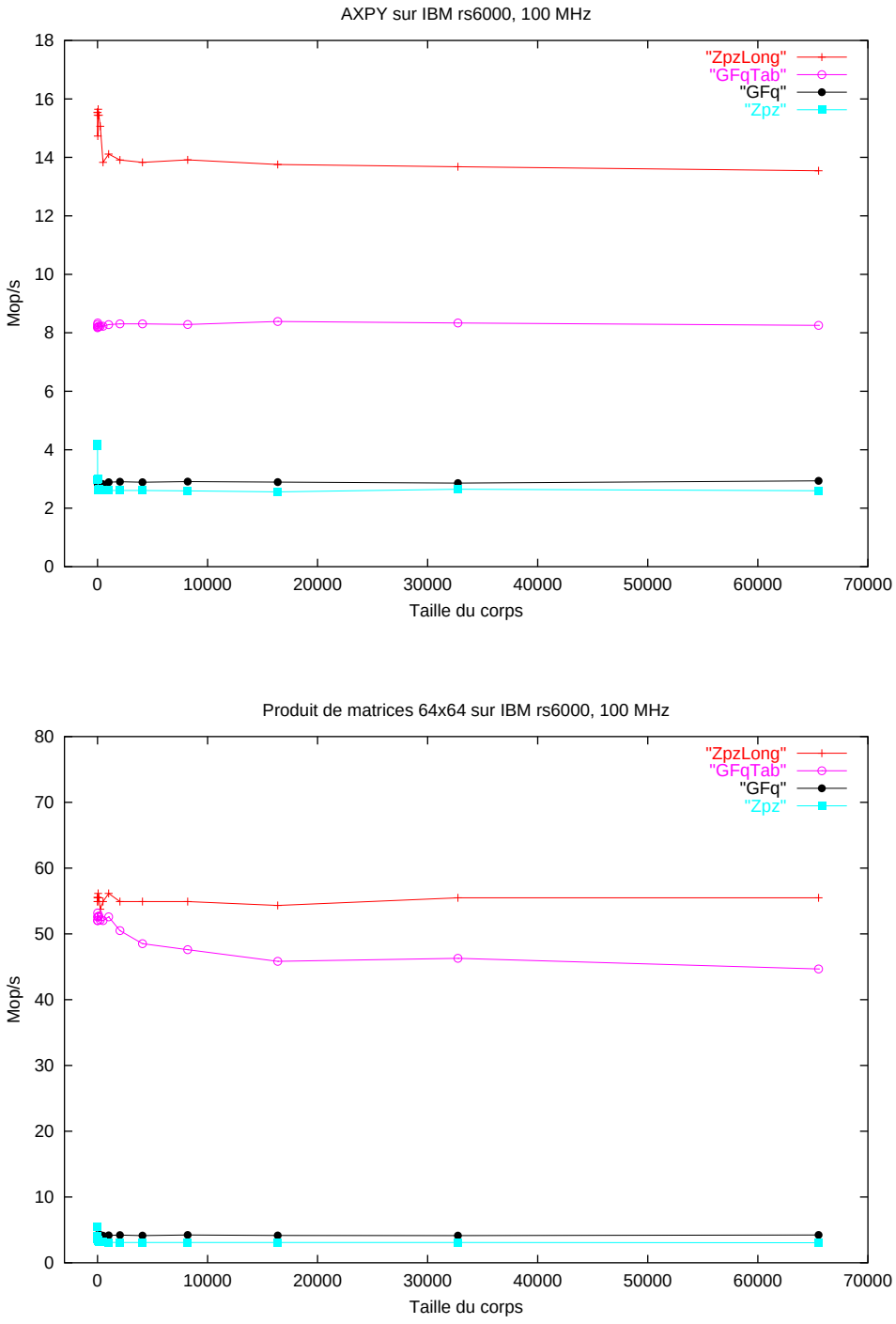

FIGURE 4.3 – AXPY et produit de matrices sur IBM rs6000, 100 MHz

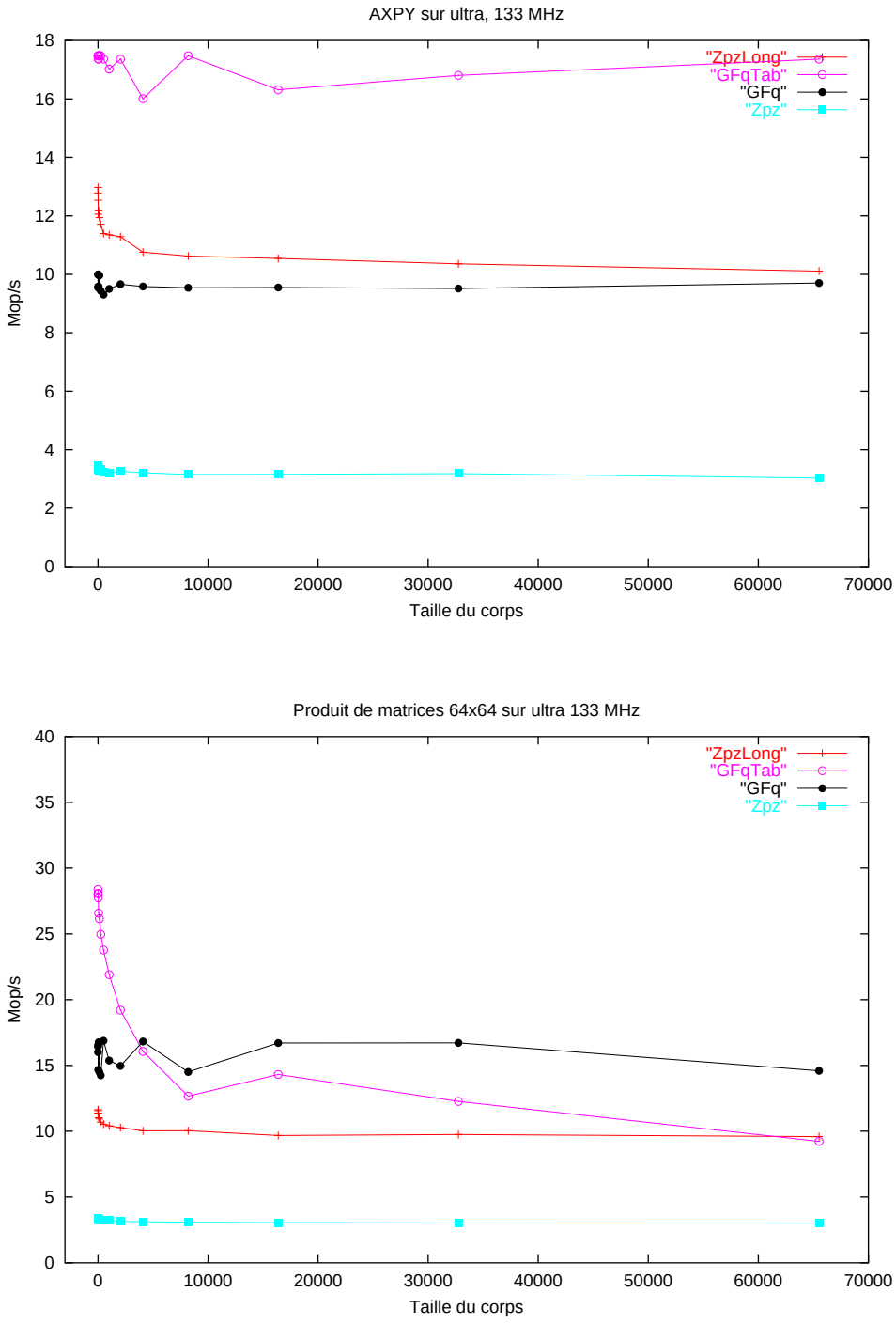

FIGURE 4.4 – AXPY et produit de matrices sur ultrasparc, 133 MHz

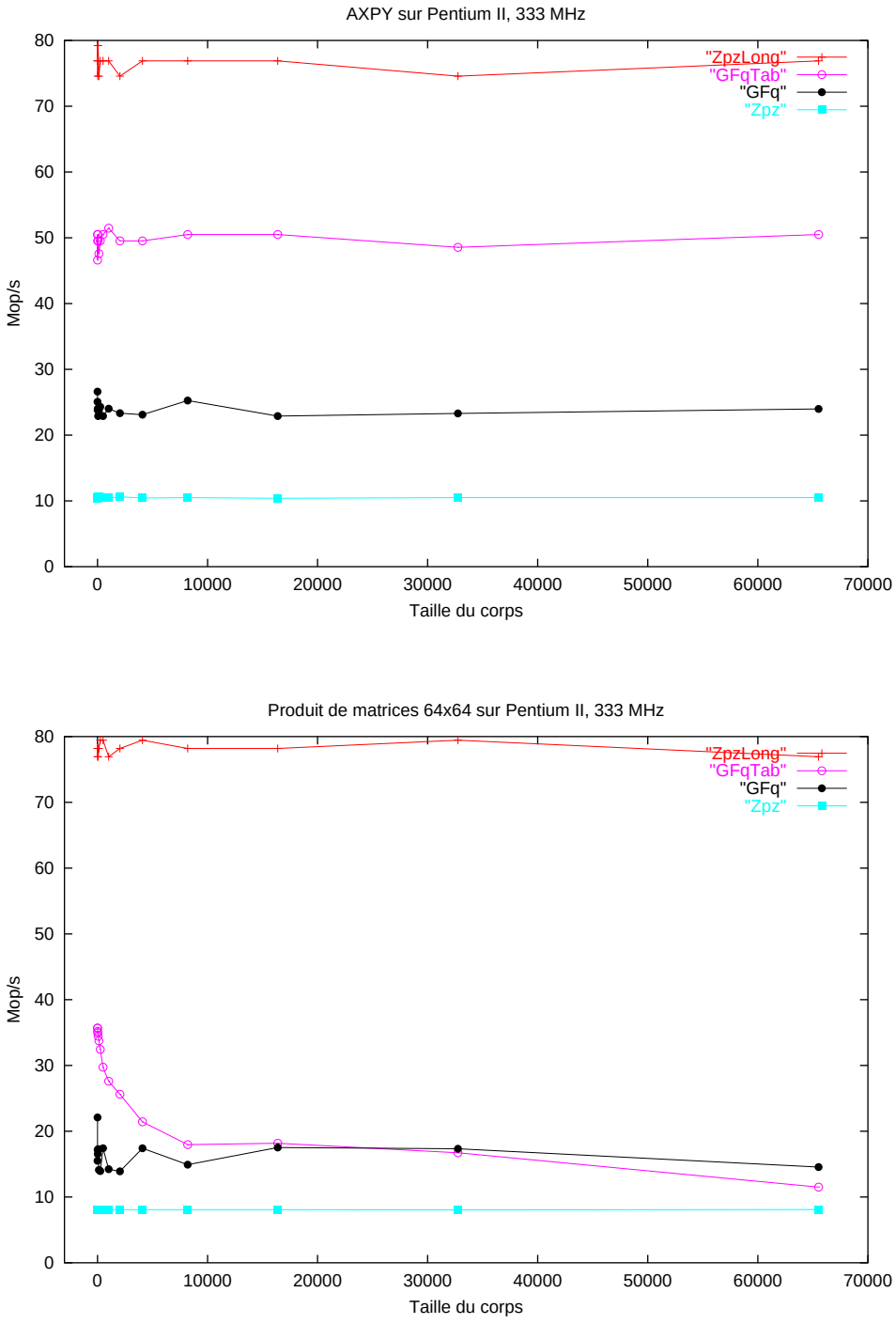

FIGURE 4.5 – AXPY et produit de matrices sur Pentium II, 333 MHz

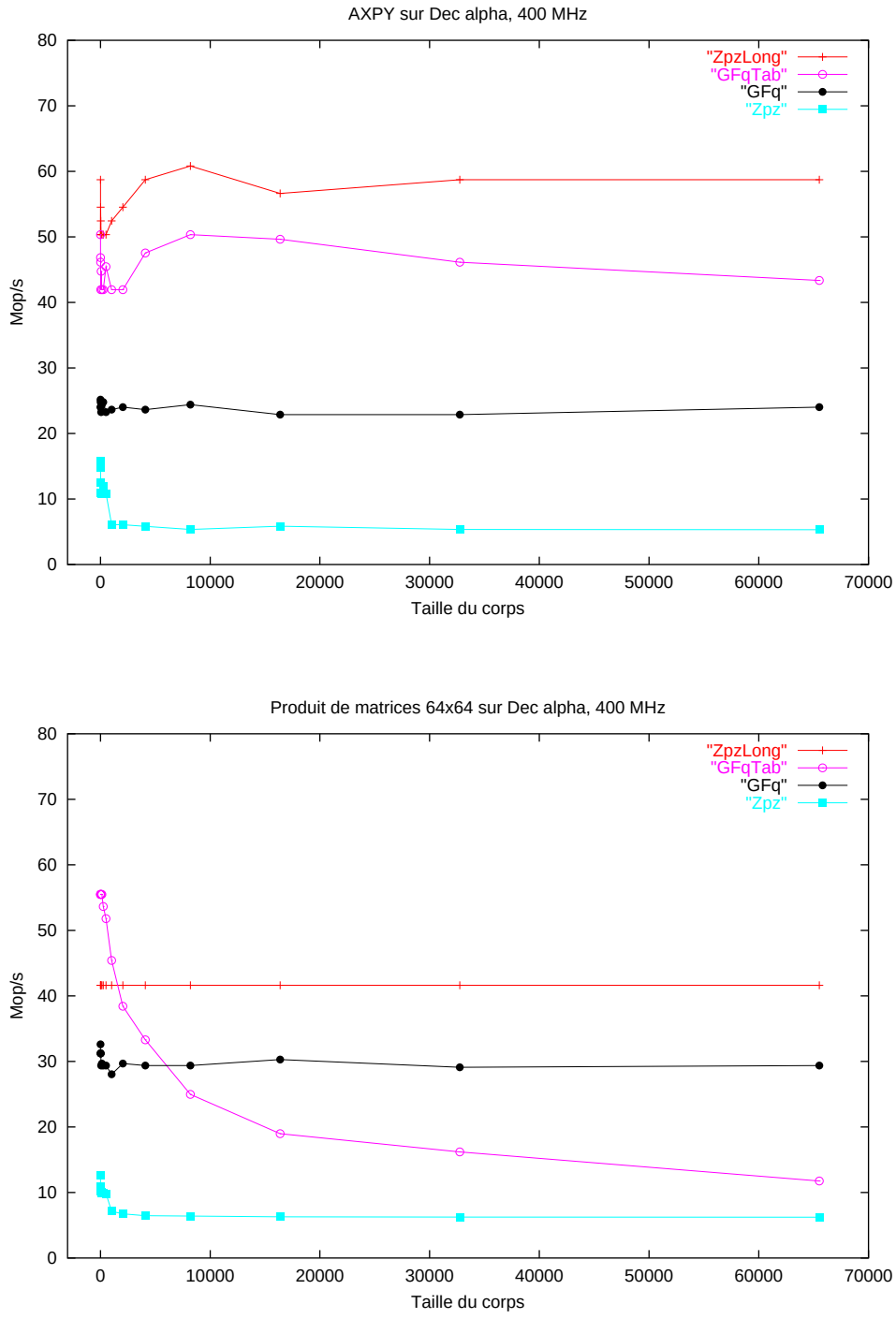

FIGURE 4.6 – AXPY et produit de matrices sur Dec alpha, 400 MHz

Deuxième partie
Algèbre linéaire creuse

Le but de cette partie est d'étudier le calcul du rang de matrices creuses. Une matrice creuse, par opposition à une matrice dense, est une matrice dont beaucoup d'éléments sont nuls. Le calcul du rang et, plus généralement, la résolution exacte de grands systèmes linéaires creux sont utilisés, par exemple, en Topologie algébrique, en combinatoire ou encore en analyse Diophantienne par le calcul de formes normales de matrices (voir chapitre 8). Ils sont aussi utiles en cryptographie pour factoriser de grands entiers ou calculer des logarithmes discrets [130 - Odlyzko (2000)].

Une première approche consiste à utiliser une méthode directe, l'élimination de Gauß. Cette méthode a une complexité arithmétique cubique dans le pire cas (inférieure au cube des dimensions de la matrice) et une complexité spatiale quadratique (inférieure au produit des dimensions). Toutefois, dans de nombreux cas, cette méthode est relativement efficace en pratique si la structure creuse est prise en compte. Le chapitre 5 est consacré à l'étude d'heuristiques séquentielles et parallèles adaptées au cas creux.

La deuxième approche que nous considérons consiste à utiliser des méthodes itératives. Celles-ci ont en général une complexité arithmétique théorique fonction du nombre d'itérations et du nombre d'éléments non nuls de la matrice. Le nombre d'éléments non nuls étant, au pire, quadratique, si le nombre d'itérations est au plus linéaire, la complexité théorique peut être meilleure que celle des méthodes directes. En outre, la complexité spatiale de ces méthodes est toujours fonction du nombre d'éléments non nuls. Cependant, pour ces méthodes, le terme de premier ordre de la complexité arithmétique est multiplié par des facteurs constants ou logarithmiques, ce qui induit, en pratique, des temps de calculs souvent importants. Le chapitre 6 étudie différentes méthodes itératives pour déterminer les plus efficaces en séquentiel et en parallèle.

Le chapitre 7, enfin, réalise la comparaison de ces deux méthodes, sur différentes classes de matrices creuses, pour déterminer leurs domaines d'applications préférentiels.

5

MÉTHODES DE GAUß

« Malheureusement, il est peu reconnu que les écrits scientifiques les plus valables sont ceux dans lesquels l'auteur indique clairement ce qu'il ne sait pas. »

Évariste Galois

Sommaire

5.1 STRATÉGIES DE PIVOT ET RENUMÉROTATION

Dans un corps, le rang d'une matrice est la dimension de l'image de l'application linéaire associée à A, ou encore la dimension de l'espace engendré par les lignes (ou les colonnes) de A.

5.1.1 DU REMPLISSAGE DANS L'ALGORITHME DE GAUSS

Nous considérons ici l'élimination de Gauß sur matrice creuse dans un corps $\mathbb{F}$ pour calculer le rang. Dans la suite, pour une matrice A, $A[i]$ désigne la ligne d'index i de A et a_{ij} désigne l'élément de ligne i et de colonne j de A. En supposant que la matrice est stockée par lignes, par exemple, l'algorithme est classique :

Algorithme 5.1.1 *Gauß-Lignes*

Entrées : – une matrice $A \in \mathbb{F}^{m\times n}$.

Sorties : – le rang de A sur $\mathbb{F}$.

1 : $r = 0$

2 : $\Lambda = \{k, \exists j, a_{kj} \neq 0\}$

3 : **Tant que** $\Lambda \neq \emptyset$ **Faire**

4 : $\quad ++r$

5 : $\quad$ Choisir et retirer $k \in \Lambda$.

6 : $\quad$ Choisir j tel que $a_{kj} \neq 0$.

7 : $\quad$ **Pour tout** $i \in \Lambda, i \neq k$ **tel que** $a_{ij} \neq 0$ **Faire**

8 : $\quad\quad A[i] = A[i] - \frac{a_{ij}}{a_{kj}} A[k]$.

9 : $\quad$ Retirer de Λ les indices dont les lignes sont devenues nulles.

10 : Renvoyer r.

Théorème 5.1.2

Soit A une matrice creuse de $\mathbb{F}^{m\times n}$ de rang r avec au plus ϖ éléments non nuls par ligne. Dans le pire cas, l'algorithme Gauß-Lignes *nécessite $2\sum_{k=1}^{r}(m-k)\min\{\varpi 2^k, n-k\} \leq 2rmn$ additions ou multiplications sur le corps de base et au plus mn unités mémoire.*

Preuve de 5.1.2 : Il faut considérer que le nombre d'éléments par ligne est au plus doublé à chaque étape ; ce qui donne au plus $\min\{\varpi 2^k, n-k\}$ éléments par ligne à l'étape k. La borne s'en déduit par sommation. □

Cela revient à dire qu'au cours de l'algorithme d'élimination, une matrice creuse se remplit, jusqu'à devenir quasiment dense. Pour quantifier ce remplissage, on peut considérer deux phases de l'algorithme. Une première phase tant que la matrice restante $(m-k) \times (n-k)$ est creuse, une deuxième phase quand cette matrice est devenue dense. En fait, dans le pire cas, le nombre d'étapes creuses est inférieur à $\log_2(\frac{n}{\varpi})$ puisque $\varpi 2^{\log_2(\frac{n}{\varpi})} = n > n-k$. La première phase est donc rapide, puisque les étapes sont linéaires (en $\mathcal{O}(\varpi(m-k))$), mais elle ne peut durer que $\log_2(\frac{n}{\varpi})$ étapes. Au contraire, la deuxième phase est lente, puisque les étapes sont devenues quadratiques (en $\mathcal{O}((n-k)(m-k))$) et qu'il peut y en avoir de l'ordre de n. La section suivante étudie différentes méthodes pour réduire le remplissage, c'est-à-dire augmenter le nombre d'étapes de la phase creuse.

5.1.2 HEURISTIQUES DE RENUMÉROTATION

Typiquement, les méthodes de réduction du remplissage ont été développées pour le calcul numérique. Il s'agit de jouer sur le choix de pivot ou, de manière équivalente, de renuméroter la matrice pour que les bons pivots soient choisis. Ces méthodes sont principalement issues de la théorie des graphes où une matrice est considérée comme la matrice d'adjacence d'un graphe G : une variable du système d'équations est considérée comme étant un sommet du graphe et chaque valeur non nulle est associée à une arête. Ainsi, l'élimination d'une variable du système d'équations (i.e. une colonne de la matrice) correspond à l'élimination d'un sommet du graphe. Pour éliminer un sommet v du graphe, il faut tout d'abord ajouter toutes les arêtes qui relient deux voisins de v, puis retirer v et toutes ses arêtes incidentes. Berman et Schnitger [13 - Berman et Schnitger (1990)], par exemple, donnent plus de détails sur la correspondance entre graphe et matrice. Simplement, le nombre de valeurs non nulles introduites par une étape d'élimination correspond, sur le graphe, au nombre d'arêtes ajoutées par une élimination de sommet. Malheureusement, Yannakakis a montré que le calcul du remplissage minimal est NP-complet pour un graphe quelconque [176 - Yannakakis (1981)]. De nombreuses heuristiques ont toutefois été développées à partir de cette idée, et elles réduisent le remplissage tout en n'étant pas optimales. En particulier, deux méthodes sont principalement utilisées, la méthode de Markowitz et différentes variantes d'« ordonnancement de degré minimal ». Cette dernière élimine à chaque étape le sommet de degré minimal. On fait alors correspondre un sommet et son degré avec une ligne et son nombre d'éléments. La méthode

de Markowitz utilise une fonction de coût : si, à l'étape k, $r_i(k)$ est le degré de la ligne i (le nombre d'éléments non nuls dans cette ligne) et $c_j(k)$ celui de la colonne j, la méthode de Markowitz choisit d'éliminer le sommet minimisant $(r_i(k)-1)(c_j(k)-1)$. Les travaux de Zlatev [178 - Zlatev (1992)], Duff et Reid [51 - Duff et al. (1986)] ou encore Amestoy et Davis [4 - Amestoy et al. (1996)] sont des références sur le sujet. D'autre part, [3 - Amestoy et al. (1999), 78 - Hendrickson et Rothberg (1999), 94 - Karypis et Kumar (1999)] et [49 - Doreille (1999), Section 8.3.2], par exemple, proposent une autre variante de réduction du remplissage, les dissections emboîtées ; si cette méthode réduit le remplissage de 20 à 30% dans certains cas numériques symétriques par rapport aux méthodes de degré minimal, elle nécessite une factorisation symbolique préalable trop coûteuse induisant souvent une perte de performances en temps et est en outre peu adaptable au cas non inversible, non symétrique. Nous ne l'utiliserons donc pas ici. Toujours est-il que toutes ces méthodes proviennent de l'analyse numérique et ne tiennent pas compte des valeurs des éléments de la matrice. C'est-à-dire qu'il est considéré qu'une opération de type $a_{ij} = a_{ij} + \delta_i * a_{kj}$ ne peut pas produire de zéro. Dans un très petit corps fini, par exemple, cela arrive pourtant souvent. En calcul formel, LaMacchia et Odlyzko [102 - LaMacchia et Odlyzko (1991)] ont considéré le problème de réduction du remplissage modulo 2. Cependant, ils proposent une élimination de Gauß structurée, dédiée à leur application : la factorisation. S. Cavallar a ensuite généralisé leur approche, toujours dans le cadre de la factorisation [34 - Cavallar (2000)]. Nous proposons ici une heuristique, adaptée de la méthode de Markowitz et utilisant quelques idées de LaMacchia et Odlyzko. Cette heuristique est en deux points.

(1) D'une part, avant le début de l'élimination, nous utilisons une des étapes de l'algorithme de LaMacchia et Odlyzko, celle qui consiste à réduire la matrice en une matrice plus petite : si il existe une ligne (respectivement une colonne) contenant une seule valeur non nulle, on supprime la ligne et la colonne associée à cette valeur et le rang est incrémenté. L'étape suivante de l'algorithme de LaMacchia et Odlyzko nécessite le maintien d'ensembles de lignes légères (avec peu d'éléments non nuls) et de lignes lourdes (avec plus d'éléments), ces ensembles pouvant évoluer au fil de l'algorithme et permettant de choisir les pivots. Pour ne pas introduire trop de surcoût de structures, nous avons préféré utiliser une variante de la méthode de Markowitz.

(2) Étant donné que les mises à zéro ne peuvent être prévues avant le déroulement de l'algorithme, notre heuristique minimise le remplissage seulement à chaque étape de calcul. Il s'agit de jouer sur les choix de pivots de l'algorithme classique : à l'étape k, nous choisissons une ligne i ayant un nombre minimal d'éléments non nuls (ainsi elle introduira un nombre minimal d'éléments pour

chaque élimination) puis nous choisissons une colonne ayant un nombre minimal d'éléments non nuls (ainsi le nombre d'éliminations sera minimisé). De cette manière, le choix de pivot coûte $m + r_i(k) \leq m + n$ tests tout au long de l'algorithme, alors que le calcul de la fonction de Markowitz coûte $\Omega = m\varpi$ tests et multiplications dans les premières étapes et devient quadratique au fur et à mesure que la matrice se remplit. L'idée utilisée ici rejoint celle des minima locaux de [146 - Rothberg et Eisenstat (1998)] utilisée pour la factorisation de Cholesky numérique. Nous montrons les bonnes performances obtenues par la conjugaison de ces deux heuristiques sur nos matrices creuses pour le calcul du rang dans les sections suivantes.

Nous présentons maintenant notre algorithme. Si nous supposons toujours que la matrice creuse est stockée par lignes, alors le nombre d'éléments non nuls par ligne est facile à obtenir, mais, pour les colonnes, il faut le calculer au fur et à mesure. L'algorithme est modifié comme suit (les degrés des colonnes sont stockés dans D_j et calculés lignes 6, 15 et 20).

Algorithme 5.1.3 *Gauß-Lignes-Renumérotation*

Entrées : – une matrice $A \in \mathbb{F}^{m \times n}$.

Sorties : – le rang de A sur $\mathbb{F}$.

1 : $r = 0$

2 : **Tant que** il existe une ligne avec une seule entrée $a_{kj} \neq 0$ **Faire**

3 : $++r$

4 : Retirer la ligne k et la colonne j de A.

5 : **Pour** $j = 1$ **jusqu'à** n **Faire**

6 : Calculer D_j, le nombre d'éléments non nuls dans la colonne j.

7 : **Tant que** il existe une colonne avec une seule entrée $a_{kj} \neq 0$ **Faire**

8 : $++r$

9 : Retirer la ligne k et la colonne j de A.

10 : $\Lambda = \{k, \exists j, a_{kj} \neq 0\}$

11 : **Tant que** $\Lambda \neq \emptyset$ **Faire**

12 : $++r$

13 : Choisir et retirer $k \in \Lambda$ tel que $|A[k]|$ soit minimal.

14 : **Pour tout** j **tel que** $a_{kj} \neq 0$ **Faire**

15 : Décrémenter D_j

16 : Choisir j tel que $a_{kj} \neq 0$ et D_j soit minimal.
17 : **Pour tout** $i \in \Lambda$ **tel que** $a_{ij} \neq 0$ **Faire**
18 : $A[i] = A[i] - \frac{a_{ij}}{a_{kj}} A[k]$.
19 : **Pour** $h == 1$ **jusqu'à** n **Faire**
20 : Décrémenter ou incrémenter D_h si a_{ih} a changé de statut.
{Devenu nul, ou devenu non-nul}
21 : Retirer de Λ les indices dont les lignes sont devenues nulles.
22 : Renvoyer r.

Pour cet algorithme, les premières lignes correspondent donc à l'étape de La-Macchia et Odlyzko. Ensuite, c'est l'algorithme de Gauß standard auquel nous avons ajouté les calculs de D_j. La différence vient du choix de pivots. Dans l'algorithme 5.1.1, page 81, ce choix était indifférent ; ici nous prenons le pivot minimisant le nombre d'éléments de sa ligne puis le nombre d'éléments de sa colonne.

5.2 RÉSULTATS EXPÉRIMENTAUX

Dans cette section, nous présentons les gains, obtenus par notre heuristique, en nombre d'opérations arithmétiques et en temps sur différentes matrices. Plus de détails sur ces matrices sont donnés section 7.1, page 146. Pour pouvoir comparer les matrices entre elles, nous montrons le *gain* de la renumérotation ; si tps_{no} (respectivement op_{no}) est le temps d'exécution (resp. le nombre d'opérations) sans renumérotation (le pivot choisi est le premier non nul en parcourant la matrice en largeur d'abord) et tps_{re} (resp. op_{re}) est le temps (resp. le nombre d'opérations) avec renumérotation, les gains sont définis comme suit :

$$gain(\text{temps}) = 100 * \frac{tps_{no} - tps_{re}}{tps_{no}}$$
$$gain(\text{opérations}) = 100 * \frac{op_{no} - op_{re}}{op_{no}}$$

Ainsi, nous avons une normalisation du rapport. Un gain proche de 0 est tout de même un gain, un gain négatif est une perte de performances. Le gain théorique maximal est 100.

Les expériences ont été menées dans $\mathbb{Z}/_{\mathrm{p}}\mathbb{Z}$, avec l'implémentation *GFq* de la section 4.1.2, page 61, sur des machines identiques (ultrasparc II à 133 MHz)

et pour deux nombres premiers p, un très petit, 3 et un de l'ordre du demi-mot machine, 65521.

5.2.1 MATRICES ALÉATOIRES

Tout d'abord, nous avons testé notre algorithme sur un ensemble de matrices creuses aléatoires de dimensions 1000×1000 jusqu'à 5000×5000 avec de 1 à quelques dizaines d'éléments non nuls par ligne. La figure 5.1, page 87 présente les résultats pour *un* tirage aléatoire de matrices. Chaque point sur les courbes correspond à une seule matrice. Nous avons toutefois réalisé ces expériences sur plusieurs jeux de matrices et les résultats étaient comparables. Enfin, la figure supérieure correspond à un nombre d'éléments par ligne entre 0 et 6, la figure inférieure entre 0 et 35. La figure inférieure reprend donc la partie supérieure, pour donner une visualisation d'ensemble.

(1) Pour les matrices extrêmement creuses (jusqu'à 1 ou 2 éléments non nuls par ligne), nous observons que notre heuristique est quasiment optimale. Clairement, cela est dû à l'astuce de LaMacchia et Odlyzko, qui perd de son efficacité dès que le nombre d'éléments par ligne dépasse 2.

(2) Ensuite, entre 2 et 4 éléments par ligne, on a une légère baisse du gain (toujours plus de 50% de gain tout de même !), puisque sur des matrices aléatoires avec beaucoup d'éléments, l'astuce de LaMacchia et Odlyzko n'est plus applicable. D'autre part, cette diminution du gain vient du fait que pour les matrices très creuses et d'assez petite dimension, même sans renumérotation, le remplissage n'est pas trop important.

(3) Au contraire, dès que le nombre d'éléments par ligne dépasse 3, le remplissage est réduit de façon spectaculaire, puisque le gain en opérations remonte à plus de 90%. Enfin, évidemment, le gain diminue à mesure que les matrices deviennent plus denses.

(4) En outre, on remarque de légères différences entre la renumérotation modulo 3 et la renumérotation modulo 65521. Comme conjecturé, le gain en opérations est meilleur pour les petits modulo. Néanmoins, les gains en temps de calcul sont parfois meilleurs pour les grands modulo, à nombre moyen d'éléments non nuls par ligne égaux. Cela vient du fait qu'une même matrice n'a pas le même nombre d'éléments non nuls modulo 3 et modulo 65521. Ainsi, à nombre moyen d'éléments non nuls par ligne égaux, une abscisse de la figure 5.1, page 87 représente une matrice de dimensions plus grandes modulo 3 que modulo 65521, ce

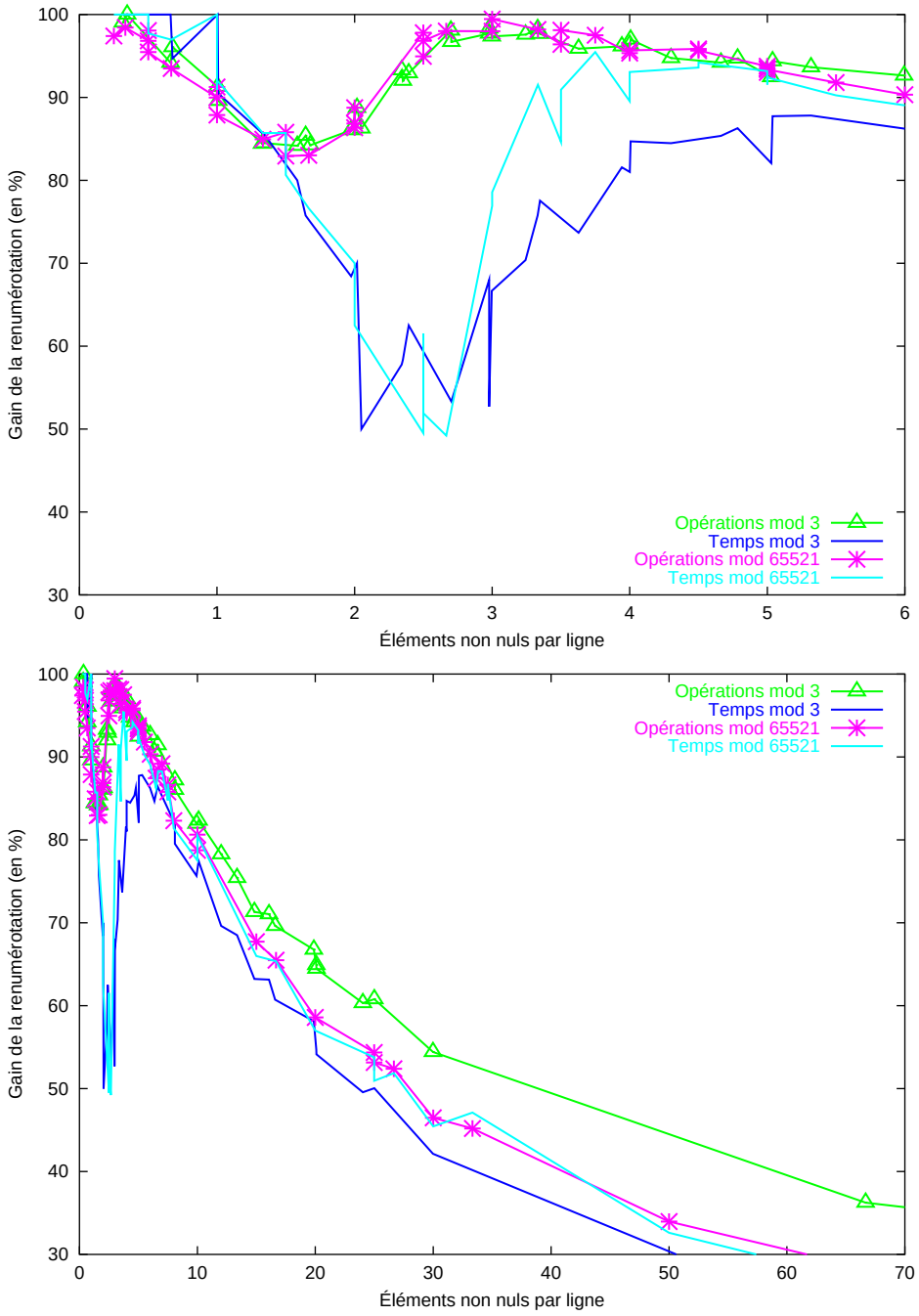

FIGURE 5.1 – Gain de la renumérotation sur matrices aléatoires ; la figure du bas continue la figure du haut, sur une plus grande échelle.

qui induit un surcoût de calcul. Ainsi, pour faire se correspondre les matrices et non plus le nombre moyen d'éléments non nuls par ligne, il faudrait décaler vers la droite, de manière progressive, les courbes modulo 3. On trouve alors une différence de gains pour le temps en faveur des petits modulo, la différence de gains en nombre d'opérations étant encore plus marquée.

5.2.2 MATRICES ISSUES DE BASES DE GRÖBNER

On observe ensuite que même sur des matrices beaucoup plus denses, comme les matrices du tableau 5.1, intervenant dans le calcul de bases de Gröbner [61 - Faugère (1994)], nous obtenons une baisse importante du nombre d'opérations. Toutefois, pour f855_mat9, ce gain n'est pas assez important et le surcoût de la renumérotation induit un léger ralentissement du calcul. Dans tous les autres cas, nous avons noté une accélération de plus de 20%.

Matrice	Éléments Non nuls	Dimensions	Rang	Gain modulo 65521 (%) Opérations	Temps
robot24c1_mat5	12.39%	404 x 302	262	51.28	30.59
rkat7_mat5	7.44%	694 x 738	611	69.50	41.78
f855_mat9	2.77%	2456 x 2511	2331	19.88	-10.22
cyclic8_mat11	9.37%	4562 x 5761	3903	37.57	20.92

TABLEAU 5.1 – Gain de la renumérotation pour matrices de bases de Gröbner

5.2.3 MATRICES D'HOMOLOGIE

Ensuite, les matrices de la figure 5.2, page 89 proviennent du calcul de groupes d'homologie de complexes simpliciaux de différents graphes [6 - Babson et al. (1999), 142 - Reiner et Roberts (2000)]. Ces matrices sont déjà quasiment diagonales et la renumérotation induit souvent un léger surcoût. Nous avons mesuré les performances de la renumérotation sur 37 de ces matrices modulo 65521. La figure présente le nombre de matrices par intervalle de gain. Ainsi, par exemple il y a 5 matrices pour lesquelles le gain en opérations est supérieur à 90%. Au contraire, pour quelques exceptions, la renumérotation conduit même à plus d'opérations. En outre, d'autres expériences ont montré que ces matrices étaient extrêmement sensibles aux moindres variations de stratégie de pivotage. En effet, ces matrices très creuses ont une phase d'élimination linéaire assez longue, puis en quelques étapes la matrice se remplit de façon spectaculaire. Par exemple, une

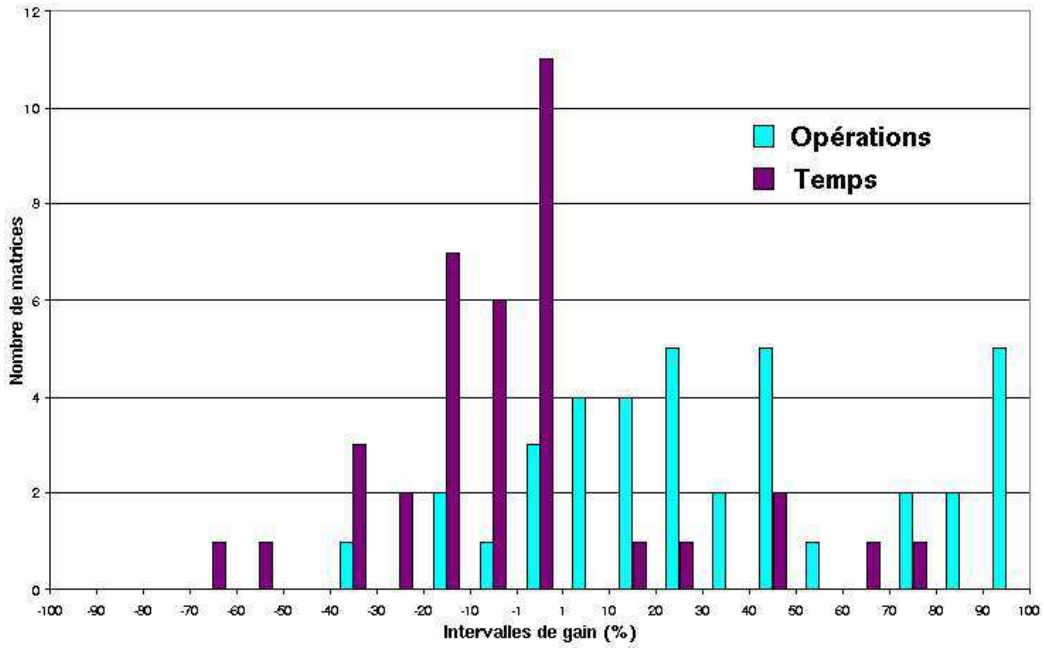

FIGURE 5.2 – Gain de la renumérotation sur matrices d'Homologie

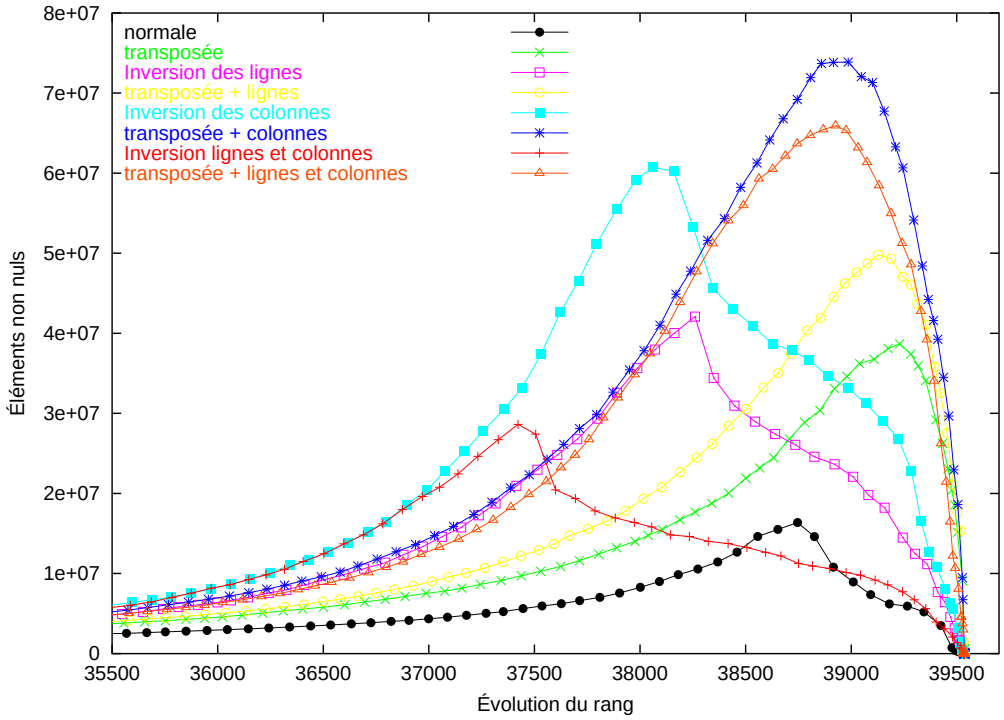

FIGURE 5.3 – Remplissage, avec renumérotation, dans la matrice mk12.b4, 62370x51975 de rang 39535

simple transposition de la matrice peut permettre de terminer le calcul avant d'arriver à la phase de remplissage et, par là même, inverser le gain. Sur la figure 5.3, page 89, nous testons la renumérotation sur la matrice mk12.b4 en échangeant les positions initiales des lignes et des colonnes (une inversion signifie que la ligne 1 se retrouve à la dernière ligne, la ligne 2 à la pénultième, etc.). Nous constatons que le remplissage est très rapide, a lieu dans les toutes dernières itérations et que de simples échanges de lignes ou de colonnes peuvent induire, sur ce type de matrices, un remplissage jusqu'à 4.5 fois plus important, et donc une mémoire minimale variant de 130 à 590 Mo. En outre, et contrairement à beaucoup d'autres matrices, le déficit de dimension en colonnes n'induit pas un remplissage moins important pour la transposée. Il semble très difficile de trouver une méthode générale de renumérotation efficace pour toutes ces matrices. Les méthodes itératives étant plus stables, de ce point de vue, nous les comparons au chapitre 7.

5.2.4 MATRICES BIBD

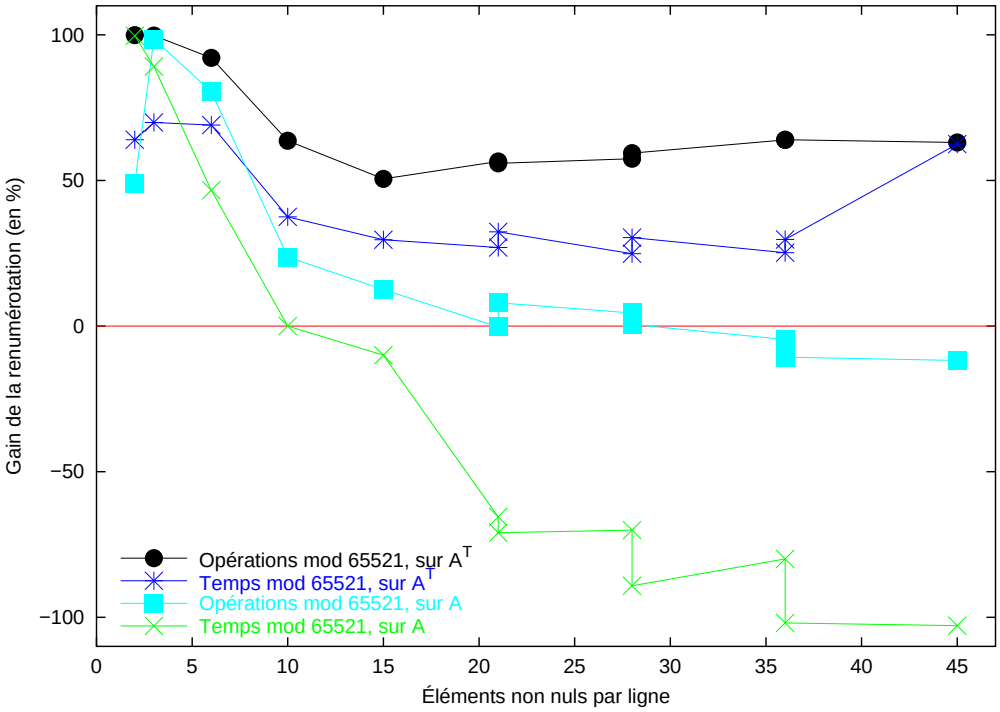

FIGURE 5.4 – Gain de la renumérotation sur matrices BIBD

Enfin, nous présentons les résultats de la renumérotation sur des matrices pro-

venant de problèmes combinatoires : les matrices de *Balanced Incomplete Block Design*. Ces matrices sont rectangulaires et très déséquilibrées ($n \times m$, avec n très inférieur à m). La figure 5.4, page 90 présente les mesures effectuées sur ces matrices ainsi que sur leur transposée. Le résultat est éloquent : si le nombre de lignes est très faible, la matrice ne se remplit pas et la renumérotation n'est pas utile.

5.2.5 MÉTHODE DE MARKOWITZ

Matrice	Gain modulo 65521 (%)	
	Opérations	Temps
robot24_m5	3.99	26.09
rkat7_m5	-19.06	44.12
f855_m9	-56.24	71.96
cyclic8_m11	-76.72	42.61
bibd.14.7	-3.80	55.77
bibd.15.7	6.01	51.61
bibd.16.8	-1.55	47.01
bibd.17.4	10.30	72.73
bibd.17.8	-5.69	45.86
bibd.18.9	-0.30	53.81
bibd.19.9	-12.38	53.95
bibd.20.10	5.49	60.20
bibd.22.8	5.26	60.82
bibd.81.3	31.60	91.31
ch6-6.b3	48.46	76.32
ch6-6.b4	47.72	78.16
mk9.b2	45.92	57.14
mk9.b3	8.95	42.86
mk10.b2	46.58	44.44
mk10.b3	69.75	77.94
mk11.b2	34.46	61.67
mk11.b3	66.19	62.81

Matrice	Gain modulo 65521 (%)	
	Opérations	Temps
n2c6.b4	27.93	68.66
n2c6.b5	74.21	85.77
n2c6.b6	45.30	77.52
n2c6.b7	23.63	80.69
n3c6.b5	41.66	78.78
n3c6.b6	36.67	77.18
n3c6.b7	31.73	78.07
n3c6.b8	84.02	88.65
n3c6.b9	1.41	78.76
n4c5.b3	44.14	62.50
n4c5.b4	64.18	78.35
n4c5.b5	61.70	83.58
n4c5.b6	48.34	82.52
n4c5.b7	41.30	86.77
n4c5.b8	21.35	82.24
n4c5.b9	21.45	71.43
n4c6.b3	19.78	65.46
n4c6.b4	67.58	63.72
n2c6.b8	16.82	62.07
n4c6.b13	19.96	85.41
n4c6.b14	3.85	70.59

TABLEAU 5.2 – Gain de notre heuristique par rapport à la méthode de Markowitz

Pour évaluer notre variante de renumérotation, nous la comparons avec la méthode de Markowitz. Nous avons implémenté celle-ci de la même manière que notre heuristique, c'est-à-dire, toujours avec la première phase de LaMacchia et

Odlyzko, seule la deuxième phase est modifiée (le choix de pivot diffère). En effet, nous sélectionnons le pivot de manière à avoir moins de $m+n$ tests à chaque étape (tout d'abord une ligne de taille minimale, puis un élément non nul de cette ligne avec une colonne de poids minimal). Au contraire, la fonction de coût de Markowitz est totale (sélectionne un élément non nul tel que le produit du poids de ligne *par* le poids de colonne soit minimal). Elle est donc optimale à chaque étape, mais avec un coût de calcul beaucoup plus important, commençant à $\Omega = m\varpi$ tests et multiplications par étape. En pratique, nous constatons sur le tableau 5.2, page 91 que la méthode de Markowitz n'est *jamais* plus rapide que notre heuristique. Toutefois, pour de nombreuses matrices, le remplissage est moins important avec cette fonction. Le coût de calcul de la fonction de Markowitz est donc clairement en cause. La différence est particulièrement marquante pour les matrices de Gröbner ($\Omega \approx 50m$) où, malgré un remplissage beaucoup plus important par notre heuristique, celle-ci reste plus rapide. Cela indique clairement que notre heuristique est loin d'être optimale pour ce type de matrices. Au contraire, pour les matrices d'homologie ($\Omega \approx 4m$), notre heuristique est meilleure en renumérotation tout en restant plus rapide. Enfin, il reste à trouver une méthode *générale* pour encore améliorer le remplissage tout en accélérant l'exécution.

5.2.6 CONCLUSION

Notre heuristique est validée, puisqu'elle apparaît extrêmement efficace en général, tant sur des matrices très creuses, que sur des matrices plus denses.

Bien sûr, si la matrice est quasiment diagonale ou rectangulaire et très déséquilibrée, la renumérotation n'est pas souvent efficace, sauf sur les plus grandes d'entre elles. En effet, la phase de remplissage n'est la plupart du temps pas atteinte pour les petites matrices. Il reste à trouver une heuristique efficace pour ces cas particuliers.

Enfin, le surcoût de notre méthode peut être estimé à partir de la figure 5.4, page 90, puisque elle présente des matrices pour lesquelles le gain en opérations est nul. En effet, pour cinq de ces matrices, non transposées, le gain en opérations est compris entre -5% et 5% et la perte de temps est de l'ordre de 70%. Le surcoût est donc relativement important puisque dans ce cas l'algorithme avec renumérotation induit donc un facteur d'environ 1.7. Il reste toutefois raisonnable par rapport à celui de la méthode de Markowitz par exemple.

5.3 ÉLIMINATION MODULO p^e

5.3.1 FORME DE SMITH ET NOTION DE RANG

Les précédents algorithmes sont prévus pour des matrices à coefficients dans un corps. Sur les entiers, les variantes de Bareiss [8 - Bareiss (1968)] et de H. Lee et D. Saunders [105 - Lee et Saunders (1995)], quand la matrice est creuse par bandes, permettent de calculer une forme triangulaire à coefficients entiers en utilisant seulement des divisions exactes. Dans la partie III, page 157, nous avons besoin de calculer le rang modulo une puissance d'un nombre premier. La différence avec le calcul sur les entiers est qu'un élément peut être non nul sans être inversible, c'est-à-dire être un diviseur de zéro dans $\mathbb{Z}/_{p^e}\mathbb{Z}$. En outre, la notion même de rang est moins claire dans un tel espace. Nous avons besoin d'une définition plus générale, mais nous n'utilisons pas la notion classique de McCoy [66 - Gantmacher (1959)] ; nous proposons plutôt une notion relative aux invariants de Smith.

Définitions 5.3.1

Soient un anneau euclidien $\mathcal{A}$ et une matrice $A \in \mathcal{A}^{m\times n}$.
- *Le déterminant d'une sous-matrice carrée de dimension $i \times i$ de A est un* **mineur** *de A d'ordre i.*
- *Le déterminant de la sous-matrice carrée $1..i \times 1..i$ de A est le* **mineur principal** *de A d'ordre i.*
- *Le* **rang** *de A est l'ordre du plus grand mineur non nul de A.*

Définitions 5.3.2

- *Le i-ème* **diviseur déterminantiel** *de A, $d_i(A)$, est le plus grand commun diviseur des mineurs de dimension $i \times i$ de A, avec i de 1 à n. Posons $d_0(A) = 1$.*
- *Le i-ème* **invariant de Smith** *de A est $s_i(A) = \frac{d_i(A)}{d_{i-1}(A)}$.*
- *La* **forme normale de Smith** *de A est la matrice diagonale $Smith(A) = S(A) = diag(s_1, \ldots, s_{\min(m,n)}) \in \mathcal{A}^{m\times n}$.*

Définition 5.3.3

Soit $q \in \mathcal{A}\backslash\{0\}$. Le **rang de McCoy de** A **modulo** q *est l'ordre du plus grand diviseur déterminantiel de A non divisible par q.*

Définition 5.3.4

Soit $q \in \mathcal{A}\backslash\{0\}$. Le **rang de Smith de** A **modulo** q, $rang_q(A)$, *est l'ordre du plus grand invariant de Smith de A non divisible par q.*

Dans un corps, ces deux définitions de rang coïncident avec la notion de dimension de l'image [66 - Gantmacher (1959)]. Sur $\mathbb{Z}$, par exemple, ces deux définitions peuvent être différentes comme le montre cet exemple : soit la matrice $A = \left[\begin{smallmatrix} 3 & 0 \\ 3 & 3 \end{smallmatrix}\right]$. Les différents mineurs d'ordre 1 sont 0 et 3, le mineur d'ordre 2 est 9. Nous avons donc $d_1(A) = 3$, $d_2(A) = 9$, mais $s_1(A) = 3$ et $s_2(A) = \frac{9}{3} = 3$. Ainsi, le rang de A sur $\mathbb{Z}$ est 2, alors que modulo 9, le rang de McCoy est 1 et le rang de Smith est 2. Dans la suite, nous utiliserons la dénomination "rang" pour le rang de Smith.

5.3.2 CALCUL DU RANG DANS $\mathbb{Z}/p^e\mathbb{Z}$

Nous voulons donc calculer le rang de Smith modulo p^e. Dans $\mathbb{Z}/p^e\mathbb{Z}$, il faut donc sélectionner des pivots inversibles, tant que cela est possible, puis factoriser par p. L'algorithme n'est donc pas très différent de l'algorithme sur un corps 5.1.1, page 81 ; par souci de simplification, nous ne donnons que la version sans renumérotation. Bien évidemment, les mêmes heuristiques sont applicables.

Algorithme 5.3.5 *Gauß-mod-p^e*

Entrées : – une matrice $A \in \mathbb{Z}^{m\times n}$.
– un nombre premier p et un entier e.

Sorties : – les rangs de A modulo p^f pour f de 1 à e.

1 : $r = 0$, $f = 0$.
2 : $\Lambda = \{k, \exists j, a_{kj} \neq 0 \mod p^{e-f}\}$
3 : **Tant que** $\Lambda \neq \emptyset$ **Faire** {Dans la boucle, les calculs sont modulo p^{e-f}}
4 : $\Lambda^* = \{k \in \Lambda, \exists j, a_{kj}$ inversible $\mod p^{e-f}\}$ { $\Lambda^* \subset \Lambda$ }
5 : **Tant que** $\Lambda^* \neq \emptyset$ **Faire**

6 : $++r$
7 : Choisir et retirer $k \in \Lambda^*$.
8 : Choisir j tel que a_{kj} est inversible.
9 : **Pour tout** $i \in \Lambda^*, i \neq k$ **tel que** $a_{ij} \neq 0$ **Faire**
10 : $A[i] \equiv A[i] - a_{kj}^{-1} a_{ij} A[k] \mod p^{e-f}$.
11 : Mettre à jour Λ^*.
12 : $++f$; $rang_{p^f} = r$.
13 : **Pour tout** i, j **Faire**
14 : $a_{ij} = \frac{a_{ij}}{p}$. {Les diviseurs de zéro sont des multiples de p.}
15 : Retirer de Λ les indices dont les lignes sont devenues nulles.
16 : $rang_{p^e} = r$.

Théorème 5.3.6

Soit un entier $e > 0$. L'algorithme Gauß-mod-p^e *est correct.*

Preuve de 5.3.6 : Il est équivalent de considérer le cas où tous les pivots sont choisis en position (k, k). Dans ce cas, nous avons une phase d'éliminations pour déterminer r_{p^f}, puis une phase de divisions. L'élimination peut être vue comme une multiplication par une matrice unitaire, triangulaire inférieure, d'inverse L_f. La division est simplement une multiplication par une matrice diagonale D_f^{-1}, où $D_f = diag(1, \ldots, 1, p, \ldots, p)$, avec le chiffre 1 intervenant r_{p^f} fois.

On se place dans $\mathbb{Z}\!\!\mathbb{Z}_{p^e}$. A a donc été factorisée : $A = P \prod_{f=1}^{e} L_f D_f \tilde{A} Q$, où P et Q sont des matrices de permutations et $\tilde{A}$ est la forme triangulaire supérieure obtenue après élimination sur A. Comme tous les éléments diagonaux non nuls de $\tilde{A}$ sont inversibles modulo une puissance de p, ils sont tous premiers avec p. Ainsi, tous les mineurs principaux non nuls de $\tilde{A}$ sont premiers avec p.

Notation 5.3.7

[86 - Kaltofen et al. (1990)] Si E_i^n est l'ensemble de tous les sous-ensembles de taille i de $\{1, \ldots, n\}$, alors pour I et J de E_i^n, A_{IJ} est le mineur d'ordre i pour les lignes I et les colonnes J de A.

On utilise alors la formule de Cauchy-Binet [66 - Gantmacher (1959), Proposition I.§2.14] :

$$\text{Si } C = AB, \text{ alors } \forall I, J \in E_i^n, C_{IJ} = \sum_{K \in E_i^n} A_{IK} B_{KJ} \tag{5.1}$$

Les mineurs d'un produit se déduisent donc des sommes et produits des mineurs. D'où, si $B = \prod_{f=1}^{e} L_f D_f$, il suffit de considérer les mineurs de $B\tilde{A}$. Les L_f sont triangulaires inférieures unitaires et les D_f sont diagonales, en utilisant la formule Cauchy-Binet, on voit alors que la forme de Smith de $L_f D_f$ est celle de D_f, donc que la forme de Smith de B ne peut être formée que des puissances de p et que $Smith(B) = \prod_{f=1}^{e} D_f$. Pour conclure sur les invariants de A, on utilise encore une fois la formule, car $\tilde{A}$ est triangulaire supérieure et ses mineurs principaux sont soit premiers avec p (pour les r_{p^e} premiers d'entre eux), soit nuls (pour les derniers). □

En pratique on travaille, à l'étape f, avec une matrice à coefficients dans $\mathbb{Z}/_{p^{e-f}}\mathbb{Z}$. Les opérations sont exactement les mêmes que sur un corps premier avec l'implémentation *Zpz* ; seulement, pour qu'une division soit possible, il faut que le pgcd du modulo et du numérateur divise le dénominateur.

Cet algorithme est très utile pour le calcul de la forme normale de Smith, puisqu'il permet de connaître les puissances des nombres premiers divisant les invariants en calculant modulo des petits nombres entiers. En effet, si les invariants d'une matrice ne contiennent que de petites puissances de nombres premiers, les calculs pourront s'effectuer avec l'arithmétique rapide des sous-groupes de $\mathbb{Z}$ (section 4.1.1, page 60). Nous utilisons cette technique au chapitre 10.

5.4 ALGORITHMES PARALLÈLES

Dans cette section, nous nous intéressons à la parallélisation de l'algorithme de factorisation LU exacte sur une matrice rectangulaire quelconque, à coefficients dans un corps. Notre objectif étant le calcul du rang de matrices inversibles ou non inversibles, nous relâchons les contraintes sur L, pour obtenir une factorisation TU, où U est triangulaire supérieure et T peut être quelconque. En pratique, T peut être calculée ou non et est creuse par blocs.

5.4.1 UN GRAIN TROP FIN

Une première approche consiste à employer une méthode directe parallèle [98 - Kumar et al. (1994)] sur matrice creuse stockée par lignes (respectivement colonnes), c'est-à-dire faire en parallèle les éliminations sur les $n - k - 1$ lignes restantes à l'étape k de l'algorithme classique. Cependant, les calculs modulaires

sont de faible coût et, de plus, pour des matrices très creuses, les tâches de calcul ainsi définies sont donc de grain très fin. Cet algorithme est difficilement exploitable efficacement en pratique. Il faut regrouper les opérations pour obtenir un grain moins fin. Une approche consiste à imiter le calcul numérique et à utiliser des sous-matrices. Le problème est qu'en calcul formel, ces blocs ne sont en général pas inversibles. Il y a alors deux options. La première est de rendre dynamique le découpage de la matrice et de l'ajuster pour que les blocs soient réduits et deviennent inversibles. Cela induit des synchronisations et des communications importantes à chaque étape pour calculer la redistribution des blocs. Nous proposons une deuxième méthode qui utilise des blocs non inversibles pour éviter ces synchronisations et redistributions.

5.4.2 ALGORITHME RÉCURSIF PARALLÈLE PAR BLOCS

Nous voulons donc construire un algorithme de calcul du rang tel que l'opération élémentaire soit une opération de bloc et non une opération scalaire. En outre, le découpage de la matrice en blocs doit être réalisé avant l'exécution de l'algorithme et ne pas être modifié. Notre stratégie, *TURBO*, conserve donc une structure par blocs, tels qu'ils ont été définis, dans le but de limiter les volumes de communications. Nous avons choisi de décrire l'algorithme de manière récursive pour en simplifier l'étude théorique. Le seuil d'arrêt de découpe est alors déterminé par la structure initiale de la matrice : on suppose que l'on a une matrice A de dimensions $2m \times 2n$ dans un corps $\mathbb{F}$. La méthode retenue consiste à diviser la matrice en quatre blocs :

$$A = \begin{bmatrix} NO & NE \\ SO & SE \end{bmatrix}$$

On effectue ensuite des factorisations TU locales sur chaque bloc. Cette méthode est alors appliquée récursivement jusqu'à la taille des blocs. Par souci de simplification, nous montrons ici l'algorithme pour une seule itération. L'algorithme s'effectue *en place*, c'est-à-dire que la matrice A en entrée n'est pas copiée : l'algorithme modifie les éléments au fur et à mesure. Nous présentons l'algorithme pour le calcul de la forme triangulaire supérieure U, celui-ci peut être aisément modifié pour conserver la matrice T telle que $A = TU$.

Algorithme 5.4.1 *TURBO*

Entrées : – une matrice $A \in \mathbb{F}^{2m \times 2n}$, de rang r.

Sorties : – la matrice A transformée en place pour obtenir une forme triangulaire supérieure de mineur principal $r \times r$ inversible.

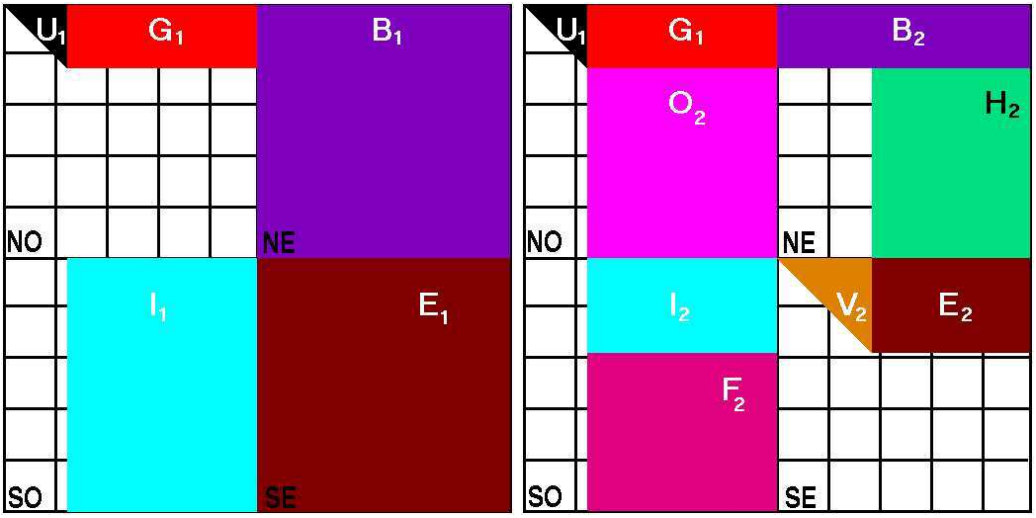

FIGURE 5.5 – Rang récursif par blocs, étapes 1 et 2

Triangularisation TU du bloc NO

1 : On calcule $L_1 \in \mathbb{F}^{m\times m}$ triangulaire inférieure, $U_1 \in \mathbb{F}^{q\times q}$ triangulaire supérieure et G_1 telles que

$$L_1 \times NO = \begin{bmatrix} U_1 & G_1 \\ 0 & 0 \end{bmatrix}$$

On peut alors lancer la mise à jour du bloc NE par $B_1 = L_1 \times NE$ et, en même temps, les calculs de mise à zéro sous U_1 dans le bloc SO ; c'est-à-dire que, si $SO_{(,1..q)}$ désigne les q premières colonnes de SO, il faut calculer $N_1 = -SO_{(,1..q)} \times U_1^{-1} \in \mathbb{F}^{m\times q}$. On effectue ensuite la mise à jour du reste du bloc SO : $I_1 = SO_{(,q+1..n)} + N_1 \times G_1$ et la multiplication par N_1 des premières lignes du bloc NE pour mettre à jour le bloc SE : $E_1 = SE + N_1 \times B_{1(1..q,)}$.

Triangularisation TU du bloc SE

2 : On calcule $L_2 \in \mathbb{F}^{m\times m}$ triangulaire inférieure, $V_2 \in \mathbb{F}^{p\times p}$ triangulaire supérieure et E_2 telles que

$$L_2 \times E_1 = \begin{bmatrix} V_2 & E_2 \\ 0 & 0 \end{bmatrix}$$

Puis de la même façon que précédemment, on peut lancer la mise à jour du bloc SO par $\left[\begin{smallmatrix} I_2 \\ F_2 \end{smallmatrix}\right] = L_2 \times I_1$ et faire les calculs de mise à zéro au-dessus de V_2 pour les dernières lignes des blocs NE et NO ; c'est-à-dire $N_2 = -B_{1(q+1..m,1..p)} \times V_2^{-1}$ puis $H_2 = B_{1(q+1..m,p+1..n)} + N_2 \times E_2$ et $O_2 = NO_{(q+1..m,q+1..n)} = N_2 \times I_2$.

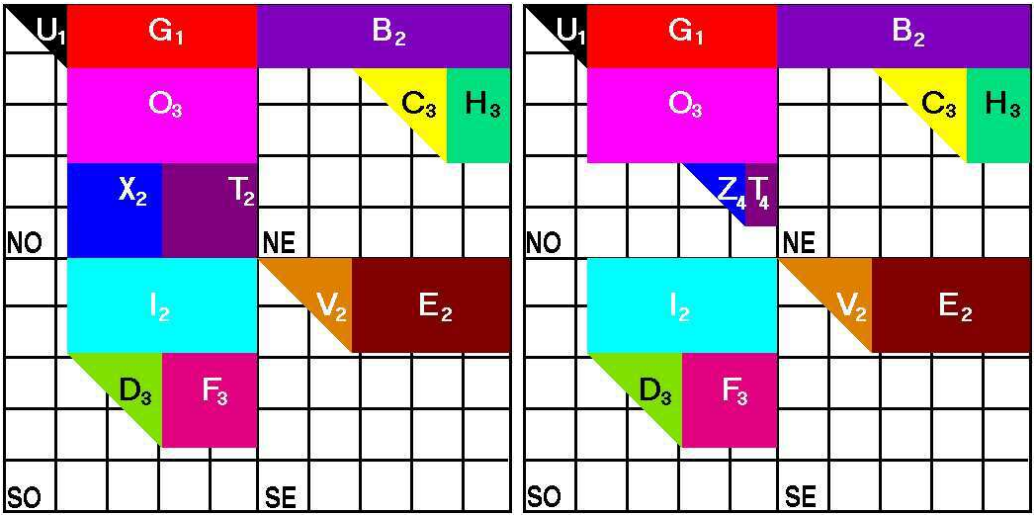

FIGURE 5.6 – Rang récursif par blocs, étapes 3 et 4

Triangularisations en parallèle dans les blocs SO et NE

3 : On calcule L_3, $D_3 \in \mathbb{F}^{d\times d}$, F_3 et M_3, $C_3 \in \mathbb{F}^{c\times c}$, H_3 telles que

$$L_3 \times F_2 = \begin{bmatrix} D_3 & F_3 \\ 0 & 0 \end{bmatrix} \text{ et } M_3 \times H_2 = \begin{bmatrix} C_3 & H_3 \\ 0 & 0 \end{bmatrix}$$

Puis mise à jour du bloc NO

$$\left[\frac{O_3}{X_2\ T_2}\right] = M_3 \times O_2$$

et ensuite mise à zéro au-dessus de D_3 par $N_3 = -X_2 \times D_3^{-1}$ et calcul de $T_3 = T_2 + N_3F_3$. Suivant la taille de D_3 et de C_3, il est possible de mettre à zéro à gauche de C_3 plutôt qu'au-dessus de D_3 dans le bloc NO. Il faut choisir celui des deux qui est le plus petit, pour minimiser calculs et communications. D'un autre côté, une implémentation parallèle peut choisir plutôt celui des deux qui est prêt avant l'autre.

Triangularisation dans le bloc NO

4 : On calcule L_4, $Z_4 \in \mathbb{F}^{z\times z}$ et T_4, telles que

$$L_4 \times T_3 = \begin{bmatrix} Z_4 & T_4 \\ 0 & 0 \end{bmatrix}$$

Permutations de lignes

5 : I_2–V_2–E_2 sont insérés entre U_1–G_1–B_2 et O_3–C_3–H_3.

6 : Z_4–T_4 sont descendus au-dessous de F_3.

Permutations de colonnes

7 : B_1–C_3–H_3–E_2 sont insérés entre U_1 et G_1–I_2–O_3–D_3–F_3–Z_4–T_4.

Dans les figures 5.5, page 98 et 5.6, page 99 les numéros correspondent à la dernière étape de modification. L'algorithme global est une modification de celui-ci, conservant aussi les matrices intermédiaires L_i et les permutations des deux dernières étapes pour leur utilisation récursive.

Enfin, le graphe 5.7, page 100 montre les dépendances de données entre les différents calculs. Ce graphe peut être récursif pour les calculs des L_i et les premières tâches de multiplication peuvent aussi être subdivisées pour obtenir plus de parallélisme. L'intérêt de cet algorithme est qu'il réduit les communications

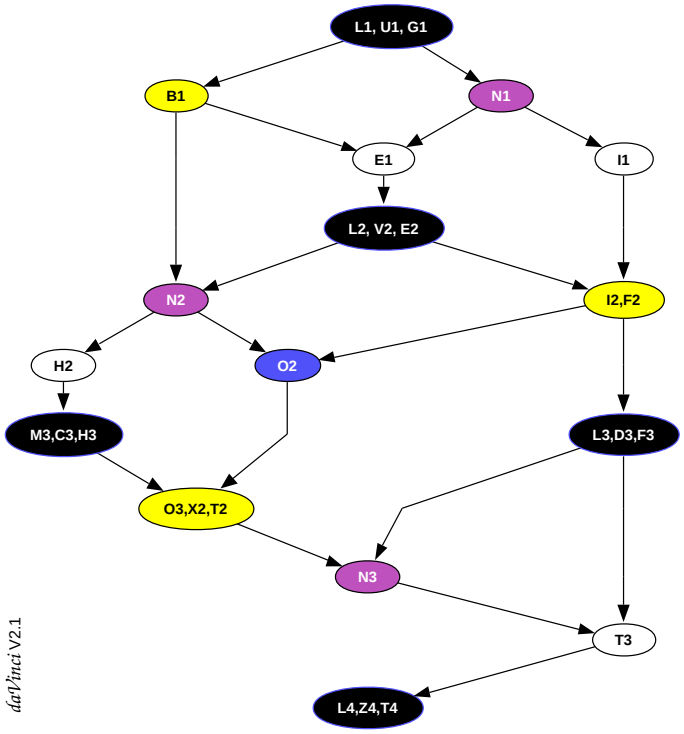

FIGURE 5.7 – Rang récursif par blocs : graphe de dépendance des tâches

puisque la plupart des calculs se font localement et que seules les petites matrices de mise à jour sont communiquées. En outre, le fait que les calculs soient locaux permet de tirer parti des effets de cache comme pour les bibliothèques de type

BLAS [46 - Dongarra et al. (1990)]. Dans les sections suivantes, nous étudions les complexités temporelles et les volumes de communications de notre technique. En particulier, le tableau 5.3, page 106, montre que le dernier bloc est souvent nul ($z = 0$), mais que la répartition des quatre autres rangs est relativement homogène.

5.4.3 COMPLEXITÉ ARITHMÉTIQUE

Nous montrons dans cette section que notre algorithme a une complexité arithmétique séquentielle identique à celle des meilleurs algorithmes actuels et une complexité parallèle identique à celle des meilleurs algorithmes d'élimination.

Nous notons ω l'exposant de la multiplication de matrices (c'est-à-dire 3 pour la multiplication classique ou $\log_2(7) \approx 2.807355$ pour la multiplication de Strassen [159 - Strassen (1969)], le record actuel étant inférieur à 2.375477 [42 - Coppersmith et Winograd (1990)] et le minimum étant 2). Le coût de la multiplication pour une matrice de dimensions $2m \times 2n$ s'écrit alors $M(h) = \mathcal{O}(h^\omega)$ pour $h = max\{2m; 2n\}$.

Théorème 5.4.2

Soit $T_{urbo}(h)$ la complexité arithmétique séquentielle de l'algorithme TURBO *pour une matrice rectangulaire quelconque de plus grande dimension h. Alors,*

$$T_{urbo}(h) \leq \frac{29}{4} M(h) + 2h^2 = \mathcal{O}(h^\omega).$$

Preuve de 5.4.2 : En suivant le graphe 5.7, page 100 et la description précédente, l'algorithme *TURBO* nécessite 4 multiplications, 1 inversion de matrice triangulaire et 2 additions pour l'étape 1, 4 multiplications, 1 inversion de matrice triangulaire et 1 addition pour l'étape 2 et 2 multiplications, 1 inversion de matrice triangulaire et 1 addition pour l'étape 3 ; soit un total de 10 multiplications, 3 inversions et 4 additions de matrices de taille au plus $\frac{h}{2}$. Ainsi, si le coût de la multiplication est $M(h) = Kh^\omega$, le coût de l'inversion de matrice triangulaire est alors majoré par $\frac{3}{2}M(h)$ [2 - Aho et al. (1974), Théorème 6.2] et on obtient la majoration suivante sur le coût de notre algorithme :

$$T_{urbo}(h) \leq T_{urbo}(p) + T_{urbo}(q) + T_{urbo}(c) + T_{urbo}(d) + T_{urbo}(z) + \frac{29}{2} K \left(\frac{h}{2}\right)^\omega + h^2$$

Supposons maintenant que $T_{urbo}(x) \leq K'x^\omega + 2x^2, \forall x < h$, avec $K' = \frac{29}{4}K$. Alors, par récurrence

$$T_{urbo}(h) \leq K'(q^\omega + c^\omega + z^\omega + p^\omega + d^\omega + 2\frac{h^\omega}{2^\omega}) \\ +2(q^2 + c^2 + z^2 + p^2 + d^2) + h^2.$$

Mais, comme $\omega \geq 2$, et que tous les rangs sont positifs, on majore $q^\omega + c^\omega + z^\omega$ par $(q + c + z)^\omega$ et $p^\omega + d^\omega$ par $(p + d)^\omega$. Enfin, grâce à la description de la section précédente, q, p, c, d, z, les cinq rangs intermédiaires des matrices U_1, V_2, C_3, D_3, Z_4, vérifient :

$$q + c + z \leq \frac{h}{2} \qquad \text{et} \qquad p + d \leq \frac{h}{2} \tag{5.2}$$

$$q + d + z \leq \frac{h}{2} \qquad \text{et} \qquad p + c \leq \frac{h}{2} \tag{5.3}$$

Par les relations 5.2, on obtient alors

$$T_{urbo}(h) \leq (\frac{4K'}{2^\omega})h^\omega + 2h^2 \leq \frac{29}{4}M(h) + 2h^2$$

et, comme $T_{urbo}(2) = 3$, la récurrence est prouvée. □

La complexité obtenue ici est donc identique à la meilleure complexité séquentielle connue pour ce problème [80 - Ibarra et al. (1982), Théorème 2.1]. L'algorithme d'Ibarra et al. est un regroupement de lignes en deux sous-ensembles. Ensuite, selon le rang obtenu sur un des sous-ensembles, après le premier appel récursif, la structure de la matrice est modifiée. Par notre découpage en blocs, nous garantissons que les accès sont locaux, ce qui permet de mieux tirer parti des effets de cache.

D'autre part, avec notre algorithme, nous obtenons une complexité parallèle linéaire, en considérant que les calculs de multiplications et d'inversions de matrices triangulaires en parallèle s'effectuent en temps logarithmique $K_\infty \log_2{}^2(h)$:

Théorème 5.4.3

Soit $T_{urbo//}(h)$ la complexité arithmétique parallèle de l'algorithme TURBO *pour une matrice rectangulaire quelconque de plus grande dimension h. Alors,*

$$T_{urbo//}(h) \leq 3K_\infty h = \mathcal{O}(h).$$

Preuve de 5.4.3 : $T_{urbo//}(c)$ et $T_{urbo//}(d)$ s'effectuent en parallèle. Nous pouvons supposer que $c \geq d$ sans perte de généralité. Alors, en posant $h_1 = q$, $h_2 = p$, $h_3 = c$, $h_4 = z$ on obtient :

$$\begin{aligned} T_{//}(h) &= K_\infty \log_2{}^2(h) + T_{//}(q) + T_{//}(p) + T_{//}(c) + T_{//}(z) \\ &= K_\infty \log_2{}^2(h) + \sum_{i=0}^{4} T_{//}(h_i) \end{aligned}$$

Donc, en développant sur une étape et en supposant les rangs non nuls, on obtient

$$T_{//}(h) = K_\infty(\log_2{}^2(h) + \log_2{}^2(q) + \log_2{}^2(c) + \log_2{}^2(p) + \log_2{}^2(z)) + \sum_{i=0}^{4}\sum_{j=0}^{4} T_{//}(h_{i;j})$$

où $T_{//}(h_{2;3})$, par exemple, représente le coût de calcul associé au rang p, puis au rang c, c'est-à-dire au rang du bloc C à la deuxième itération, de la découpe récursive du bloc V de la première itération. De même que dans l'analyse du cas séquentiel, les coûts sont regroupés pour que la somme des rangs dans un groupe ne dépasse pas $\frac{h}{2}$. Ainsi, on a $q + c \leq \frac{h}{2}$. Dans un premier cas, si $q \geq 1$ et $c \geq 1$, alors $\log_2{}^2(q) + \log_2{}^2(c)$ est maximal quand $q = c = \frac{h}{4}$ pour $h \geq 16$. Sinon, si l'un des deux est nul, le coût est en fait borné par $\log_2{}^2(\frac{h}{2})$. Or $\log_2{}^2(\frac{h}{2}) < 2\log_2{}^2(\frac{h}{4})$, pour $h > 2^{3+\sqrt{2}} \approx 21.3212$. Ainsi, asymptotiquement, $(\log_2{}^2(q) + \log_2{}^2(c)) \leq 2\log_2{}^2(\frac{h}{4})$ et, de même, $(\log_2{}^2(p) + \log_2{}^2(z)) \leq 2\log_2{}^2(\frac{h}{4})$. Ainsi, la formule devient

$$T_{//}(h) \leq K_\infty\big(\log_2{}^2(h) + 4\log_2{}^2(\frac{h}{4})\big) + \sum_{i,j=0}^{4} T_{//}(h_{i,j}).$$

Ensuite, en regroupant alors de la même manière, deux par deux dans chacun des quatre groupes, on termine la preuve :

$$T_{//}(h) \leq K_\infty \sum_{i=0}^{\log_4(h)} 4^i \log_2{}^2(\frac{h}{4^i}) \leq K_\infty \frac{80}{27} h \leq 3K_\infty h.$$

□

Notre algorithme donne donc une moins bonne complexité théorique que l'algorithme de Mulmuley [118 - Mulmuley (1987)]. En effet, celui-ci montre que le calcul du rang est dans NC^2 (avec un nombre polynomial de processeurs ce calcul peut s'effectuer en temps parallèle $\mathcal{O}(\log_2{}^2(n))$). Néanmoins, notre algorithme à

un coût parallèle d'ordre identique à celui que l'on obtiendrait en parallélisant l'algorithme d'Ibarra et al. Enfin, et au contraire de l'algorithme précédent, les blocs ne sont pas redimensionnés pour donner des blocs inversibles. En pratique, notre technique est donc intéressante puisqu'elle conserve une plus grande localité des données. Nous montrons en outre, dans la section suivante, que cette technique permet de réduire le volume de communications, par rapport aux algorithmes classiques.

5.4.4 GAIN DE COMMUNICATIONS

Dans cette section, nous calculons donc le gain de communications obtenu avec notre algorithme parallèle par blocs par rapport aux algorithmes classiques. Considérons une matrice dense. Dans un premier temps, nous étudions la différence de volume de communications, sur une matrice inversible $2n \times 2n$, entre une découpe par lignes et une découpe par blocs, sur P processeurs. Prenons, tout d'abord une découpe par lignes, en supposant que ces lignes sont réparties cycliquement sur les P processeurs. À chaque étape la ligne de pivot doit être communiquée à tous les autres processeurs d'où le volume total de communications :

$$\sum_{k=1}^{2n}(P-1)(2n-k) = n(2n-1)(P-1) = L(2n,2n,P)$$

Dans le cas d'une découpe en Q blocs B_{ij} carrés de taille $\frac{2n}{\sqrt{Q}} \times \frac{2n}{\sqrt{Q}}$ cycliquement répartis, nous supposons en outre qu'il existe à chaque étape au moins un bloc inversible. À l'étape k, la ligne de pivots par blocs doit être communiquée, c'est-à-dire que chacun des $\sqrt{Q} - k + 1$ blocs de cette ligne (les B_{kj} pour j de k à $\sqrt{Q}$) doit être communiqué aux $\sqrt{Q} - k$ blocs restants de sa colonne. Ensuite, chaque ligne restante effectue la multiplication par l'inverse du bloc B_{kk} puis communique aux $\sqrt{Q} - k$ autres blocs de sa ligne le produit $-B_{ik}.B_{kk}^{-1}$. Pour un bloc sur P, la communication n'a pas lieu puisque ce bloc est local. Le total de communications est donc :

$$\Big(1-\frac{1}{P}\Big)\sum_{k=1}^{\sqrt{Q}}\Big((\sqrt{Q}-k+1)(\sqrt{Q}-k)\frac{2n}{\sqrt{Q}}\frac{2n}{\sqrt{Q}} + (\sqrt{Q}-k)^2\frac{2n}{\sqrt{Q}}\frac{2n}{\sqrt{Q}}\Big)$$
$$= \frac{2}{3}(2n)^2\sqrt{Q}\Big(1-\frac{1}{P}\Big) - \mathcal{O}(n^2) = B(2n,2n,P,Q)$$

En général, pour un parallélisme optimal, on choisira Q de sorte que $P = \sqrt{Q}$ [49 - Doreille (1999)]. Dans ce cas là, le volume total de communications est supérieur à celui nécessaire par lignes. En prenant $P = Q$, il y a un gain de communications de l'ordre de $\sqrt{P}$: le gain est la différence de volume de communications entre la méthode par lignes et la méthode par blocs, divisée par le volume de la méthode par lignes et vaut dans ce cas :

$$\frac{L(2n, 2n, P) - B(2n, 2n, P, P)}{L(2n, 2n, P)} \leq 1 - \frac{4}{3\sqrt{P}}.$$

Ce type de rapport est donc notre objectif dans le cas non inversible.

Nous voulons maintenant estimer le gain obtenu avec notre algorithme, pour une matrice rectangulaire $2m \times 2n$, non inversible et de rang $r \leq min\{2m, 2n\}$. Nous calculons alors le volume de communications, au pire, pour une seule phase (pas de récursion, juste les quatre étapes) avec 4 processeurs (un pour chaque bloc), en fonction des cinq rangs intermédiaires (q, p, c, d, z, des matrices U_1, V_2, C_3, D_3, Z_4), et en supposant que le calcul de la matrice X_i a lieu sur le processeur sur lequel X_i doit être. En suivant le graphe 5.7, page 100, on voit qu'il faut alors communiquer une seule fois les blocs suivants : L_1, U_1, $B_{1(1..q,)}$, N_1, V_2, L_2, E_2, N_2, I_2, M_3, D_3, et F_3. La fonction obtenue avec 4 processeurs est déjà assez complexe :

$$mn + 2qm + 2pn + q^2 + pm + dn - pq - dq = B_1(2m, 2n, r, 4) \qquad (5.4)$$

Si l'on a choisi C_3 plutôt que D_3 à l'étape 3, il faut remplacer d par c dans la formule précédente. Pour donner une idée du gain de communications que cette méthode peut générer, nous comparons cette formule avec le volume de communications nécessaires pour faire le calcul par lignes, sur P processeurs, maintenant dans le cas non inversible :

$$\sum_{k=1}^{r}(P-1)(2n-k) = r(2n - \frac{r+1}{2})(P-1) = L(2n, r, P)$$

où $r = q + p + c + d + z$ est le rang de la matrice. En pratique, dans le cas des matrices creuses, le tableau 5.3, page 106 indique les gains obtenus pour différentes matrices issues du calcul de groupes d'homologie et de bases de Gröbner. De même que précédemment, le gain est la différence de volume de communications entre la méthode par lignes et notre méthode récursive, divisée par le volume de la méthode par lignes. Notre algorithme n'est implémenté que pour des matrices denses alors que les matrices testées sont très creuses. Néanmoins, nous pouvons

Matrice	$2m \times 2n$	r	q	p	c	d	z	Gain
mk9.b2	1260 x 378	343	189	102	52	0	0	-67.36%
mk9.b3	945 x 1260	875	286	334	136	119	0	11.80%
ch4-4.b2	96 x 72	57	24	23	7	3	0	10.40%
ch5-5.b2	600 x 200	176	100	55	21	0	0	-57.97%
ch5-5.b3	600 x 600	424	156	158	70	40	0	32.80%
ch6-6.b2	2400 x 450	415	225	105	85	0	0	-123.66%
rob24_m5	404 x 302	262	54	141	10	57	0	9.08%
rkat7_m5	694 x 738	611	95	239	130	108	39	34.02%
f855_m9	2456 x 2511	2331	123	1064	189	164	791	34.68%
c8_m11	4562 x 5761	3903	392	1927	521	354	709	21.02%

TABLEAU 5.3 – Gain de communications pour le rang récursif par blocs

constater, en général, un fort gain de communications. Dans la table, seules trois matrices nécessitent moins de communications avec la méthode par lignes : en effet, ces matrices sont très rectangulaires avec donc de très petites lignes à communiquer. En outre elle sont particulières pour notre algorithme ($d = z = 0$). Dans tous les autres cas, nous avons de bonnes performances : pour les matrices les plus carrées, nous arrivons même aux performances de l'algorithme pour matrices avec mineurs principaux inversibles (le rapport des méthodes inversibles prévoyait un gain de $1 - \frac{4}{3\sqrt{4}} = \frac{1}{3} \approx 33\%$).

Toutefois, la mise en œuvre récursive est assez délicate car pour limiter le surcoût de contrôle et de communications, le seuil d'arrêt de découpe doit être assez élevé. Le parallélisme induit n'est donc pas très extensible et l'algorithme est intéressant sur un nombre de processeurs relativement restreint (4, 8, 16, ...). Néanmoins, il y a la possibilité d'utiliser cette découpe seulement pour les premières étapes, les plus gourmandes en communications puis de changer d'algorithme pour un algorithme classique dans les étapes restantes. D'autre part, cet algorithme peut s'adapter facilement aux matrices creuses. Avec la seule restriction que dans ce cas, il n'est pas possible d'appliquer complètement les heuristiques de renumérotation de la section 5.1, page 81 sur la matrice globale sans ajouter des communications. Il est toujours faisable de les appliquer localement, dans chaque bloc, mais avec des effets limités.

5.4.5 CONCLUSIONS

En conclusion, nous avons développé un nouvel algorithme d'élimination TU par blocs. Ses complexités arithmétiques théoriques séquentielle et parallèle sont

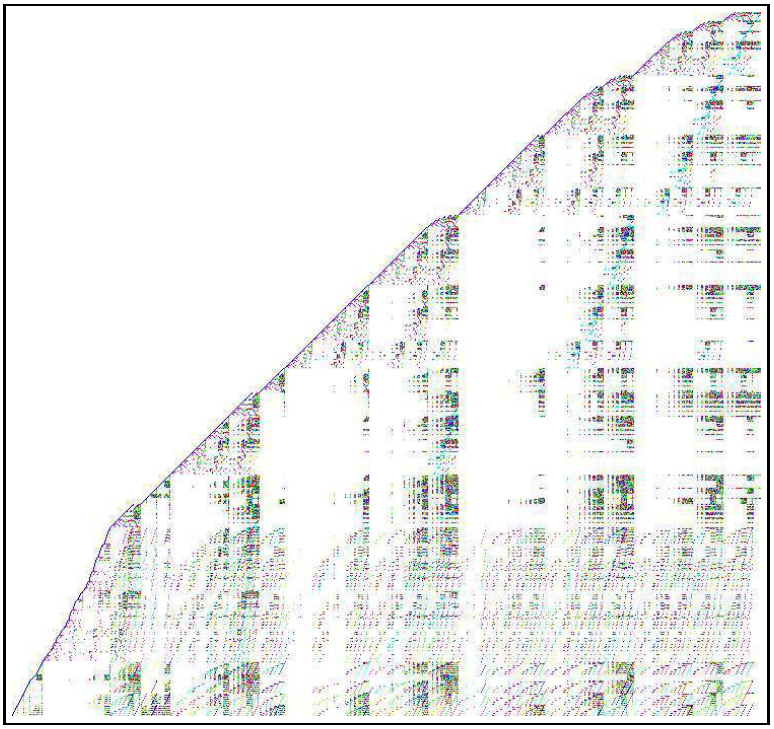

FIGURE 5.8 – Matrice rkat7_mat5, 694×738 de rang 611

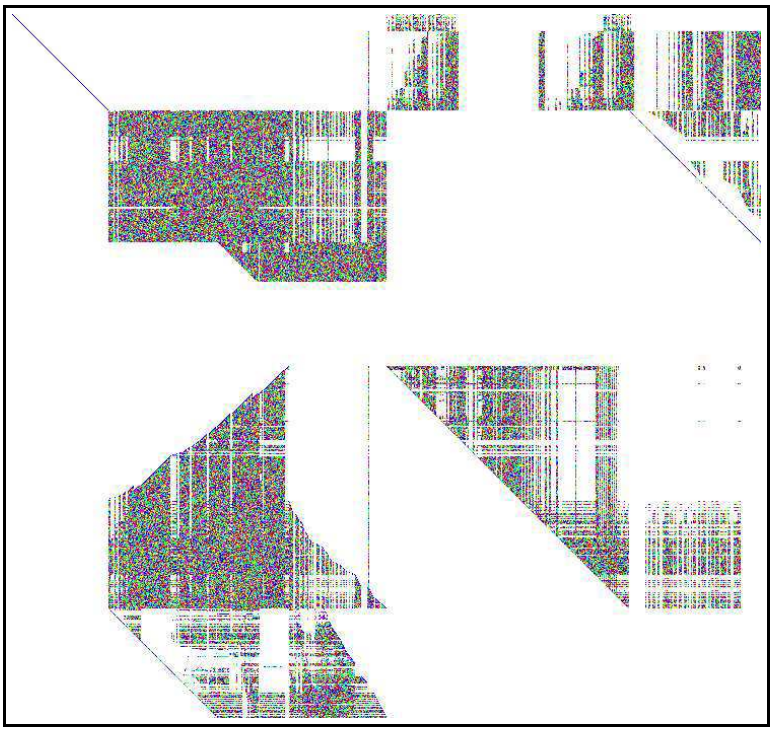

FIGURE 5.9 – Matrice rkat7_mat5 après le calcul du rang par 4 blocs

identiques à celles des meilleurs algorithmes d'élimination actuels pour ce problème. En outre, il est particulièrement adapté aux matrices non inversibles et permet de calculer le rang de manière exacte. Il permet, d'une part, une gestion plus souple du grain de calcul et réduit de moitié les communications par rapport aux précédents algorithmes pour 4 processeurs et pour un seul niveau de récursivité.

D'autre part, si l'augmentation de la localité réduit le nombre de communications, elle permet aussi d'augmenter la vitesse de calcul par un plus grand profit des effets de cache comme nous l'avons vu au chapitre 4. Enfin, il reste à étudier une méthode efficace pour renuméroter en parallèle dans le cas des matrices creuses. La figure 5.9, page 107 montre le résultat d'une itération de notre algorithme sur une matrice creuse issue du calcul de bases de Gröbner ; nous voyons qu'un remplissage assez important a eu lieu dans les dernières phases de cette itération. À titre de comparaison, la forme TU finale de cette matrice, calculée avec l'algorithme séquentiel sans renumérotation 5.1.1, page 81, comporte 64622 éléments non nuls. Celle calculée avec l'algorithme par blocs récursif avec une seule itération, sans renumérotation 5.4.1, page 97, comporte 105516 éléments non nuls (figure 5.9, page 107). Enfin, la forme calculée par l'algorithme séquentiel, avec renumérotation 5.1.3, page 84, ne comporte que 39477 éléments non nuls.

6

MÉTHODES DE KRYLOV

« It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. [...] In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. »

Alan M. Turing

Sommaire

Définition 6.0.1

Soit A une matrice de $\mathbb{F}^{n\times n}$. Le **sous-espace de Krylov** *associé à la matrice A et au vecteur u est le sous-espace vectoriel engendré par les vecteurs produits des puissances de A par u : $Krylov(A,u) = K(A,u) = Vect\{u, Au, A^2u, \dots\} = Vect\{u, Au, A^2u, \dots, A^{n-1}u\}$.*
Par extension, le sous-espace de Krylov d'une matrice non carrée est le sous-espace de Krylov associé à la matrice augmentée de colonnes ou de lignes nulles.

Le premier but de ce chapitre est d'utiliser les méthodes de Krylov pour calculer le rang. En effet, celles-ci sont intéressantes dans le cas de l'algèbre linéaire creuse puisqu'elles considèrent les matrices comme des **boîtes noires** [84 - Kaltofen (2000), Problème 3], c'est-à-dire que l'on n'accède pas aux éléments, seul le produit matrice-vecteur est utilisé. L'intérêt est double. D'une part, la matrice reste creuse tout au long de l'algorithme, elle ne se remplit pas. D'autre part, ce modèle permet de manipuler des matrices abstraites sans représentation interne en machine. Par exemple, une matrice diagonale est simplement une fonction qui prend un vecteur en entrée et renvoie un vecteur dont les éléments sont ceux du vecteur initial multipliés par les éléments diagonaux.

Le second objectif de ce chapitre est d'isoler la meilleure des variantes de Krylov. Habituellement, les complexités arithmétiques sont données en $\mathcal{O}$, c'est-à-dire à un facteur constant près, ou même en $\tilde{\mathcal{O}}$, c'est-à-dire à un facteur logarithmique près. Au contraire, nous donnons ici les complexités arithmétiques les plus précises possibles. Nous nous intéressons ici aux complexités arithmétiques exactes. En effet, la complexité en $\mathcal{O}$ des méthodes de Krylov est meilleure pour les matrices creuses que celle des méthodes par élimination, par exemple. Mais, en pratique, de grandes constantes rendent souvent celles-ci moins performantes. Nous nous attachons donc, dans ce chapitre, à calculer ces constantes pour les différentes variantes existantes afin de déterminer la meilleure implémentation possible. Le chapitre suivant réalise ensuite la comparaison expérimentale entre les méthodes de Krylov et les méthodes d'élimination puisque pour ces dernières, seules des bornes supérieures sont connues. Au contraire de Turing [164 - Turing (1948)], et d'après l'analyse du chapitre 4, une addition, une soustraction, une multiplication ou une division seront comptées comme une seule opération arithmétique.

Nous montrons tout d'abord la différence entre le calcul numérique et le calcul formel, puis nous décrivons les algorithmes de Lanczos pour la résolution de

système et la factorisation de matrices. Nous regardons ensuite une variante, assez différente au premier abord, l'algorithme de Wiedemann, calculant le polynôme minimal.

6.1 DU NUMÉRIQUE AU FORMEL

6.1.1 KRYLOV NUMÉRIQUE

Il existe de nombreuses méthodes numériques utilisant les sous-espaces de Krylov pour la résolution de systèmes ou le calcul des valeurs propres. Il est en effet possible de calculer des approximations itératives des valeurs propres d'une matrice par l'orthonormalisation de sous-espaces de Krylov [74 - Golub et Van Loan (1996), §9.1.1]. La méthode d'orthogonalisation de Gram-Schmidt peut être employée pour cela : à partir d'un vecteur initial w_1, on construit une base $(w_1, \ldots, w_k)$ de $K(A, w_1)$; voir par exemple [74 - Golub et Van Loan (1996), §9.1.1]. pour plus de détails. De manière générique, on peut montrer qu'à chaque étape, le calcul du nouveau vecteur orthogonal w_k ne dépend que de deux de ses prédécesseurs et non de tous [59 - Eberly et Kaltofen (1997), Lemme 2.1]. Cela induit une récurrence à trois termes du type $\beta_{j+1} w_{j+1} = A w_j - \alpha_j w_j - \beta_j w_{j-1}$ qui permet de calculer les w_k avec une complexité arithmétique en $\mathcal{O}(\Upsilon\Omega)$ où Ω est le coût du produit matrice-vecteur par A et $(2\Upsilon) \leq (2n)$ est le nombre d'itérations. Dans le cas des matrices creuses, cette méthode peut donc être intéressante, en pratique, par rapport à l'élimination de Gauß, puisque ce coût est proportionnel au nombre d'éléments non nuls. En outre, en calcul numérique, se rajoute le fait qu'il suffit souvent de peu d'itérations pour obtenir des solutions approchées de bonne qualité [155 - Simon (1984), 76 - Grimes et al. (1994), 149 - Saad (1996a)]. D'autre part, pour une matrice symétrique A, on montre que la tridiagonalisation orthonormale $Q^T A Q = T$ correspond à la factorisation *QR* de $K(A, w_1)$ où $w_1 = Q e_1$ [104 - Lascaux et Théodor (1994), §11.2]. On retrouve ainsi la récurrence à trois termes précédente, pour calculer une tridiagonalisation de A. À partir de cette tridiagonalisation, on peut résoudre facilement un système linéaire avec toujours une complexité globale en $\mathcal{O}(n\Omega)$: c'est la méthode de Lanczos.

De nombreuses variantes existent : Lanczos, Gram-Schmidt, gradient conjugué, etc. Elles ont des comportements différents tant du point de vue de la convergence que de la stabilité [149 - Saad (1996a)].

Au contraire, en calcul exact, ces différences disparaissent et les méthodes deviennent probabilistes. Lambert a montré l'équivalence de ces trois dernières méthodes [103 - Lambert (1996)]. D'autre part, Dornstetter [50 - Dornstetter (1987)] puis Teitelbaum [162 - Teitelbaum (1998)] ont unifié, dans le cas formel, l'algorithme de Berlekamp/Massey [110 - Massey (1969)] avec l'algorithme d'Euclide de pgcd de Polynômes, pour le premier, et l'algorithme d'Euclide et la méthode de Lanczos, pour le second. Or, l'algorithme de Wiedemann n'est autre que l'algorithme de Berlekamp/Massey, adapté au cas matriciel. Ainsi, en calcul formel, toutes les méthodes sont équivalentes. Les seules différences sont les probabilités de réussite et les constantes de complexité arithmétique. C'est par l'étude de ces différences que nous cherchons la meilleure variante.

6.1.2 ADAPTATION AU CALCUL DU RANG

Avec les nombres entiers, si la convergence et la stabilité ne sont plus de mise, de nouveaux problèmes apparaissent néanmoins :

(1) $\mathbb{Z}$ n'est pas un corps. Ainsi, les divisions, par exemple, ne sont plus possibles. Une solution est de travailler dans un corps premier $\mathbb{Z}/p\mathbb{Z}$.

(2) Les entiers sont discrets. Le calcul des racines carrées, par exemple, est alors beaucoup plus ardu dans un corps fini [130 - Odlyzko (2000)]. Il est souvent plus aisé de changer l'algorithme pour éviter un calcul de racine carrée.

(3) En caractéristique non nulle, il existe des vecteurs isotropes. C'est-à-dire des vecteurs non nuls dont la norme est nulle. Par exemple, $[1, 3, 4]$ est isotrope dans $\mathbb{Z}/13\mathbb{Z}$ puisque $1 + 9 + 16 = 2 * 13$. Au vu du lemme suivant, les chances que des vecteurs choisis aléatoirement soient isotropes se réduisent avec la taille du corps.

Lemme 6.1.1

La probabilité qu'un vecteur au hasard dans $\mathtt{GF}(q)^n$ *soit isotrope est plus petite que* $\begin{cases} \frac{2}{q} & \text{si } n = 2 \\ \frac{1}{q-1} & \text{si } n \neq 2 \end{cases}$

Preuve de 6.1.1 : Ce résultat découle de deux théorèmes de Dickson [45 - Dickson (1901), Théorèmes 65 et 66] (repris par Laksov etc Thorup [101 - Laksov et Thorup (1994)]) : si s vaut 1 ou -1 suivant que $(-1)^{n/2}$ est un carré ou pas dans $\mathtt{GF}(q)$, alors le nombre de vecteurs isotropes (y compris le nul) est : q^{n-1} si n est impair et $q^{n-1} + s(q-1)q^{n/2-1}$ si n est pair.

Si n est impair, $q^{n-1}/q^n = 1/q < 1/(q-1)$. Sinon, le nombre de vecteurs isotropes est inférieur à $q^{n-1} + (q-1)q^{n/2-1}$ lui même inférieur à $q^n/(q-1)$ si $n \geq 4$, car dans ce cas $q^{n/2-1}(q^{n/2} - (q-1)^2)/(q-1) > 0$. Enfin, pour $n = 2$, il y a au plus $2q - 1$ vecteurs isotropes ce qui est inférieur à $2q^2/q$. □

Pour réduire le risque d'isotropie, une première solution peut donc être de plonger un corps fini dans une de ses extensions. Cependant, cette solution ne fait pas disparaître le risque, elle ne fait que le diminuer. Une autre solution est de modifier l'algorithme pour qu'il prenne en compte cette isotropie. Enfin, il est souvent possible d'effectuer, après coup, une vérification de la validité du résultat. Dans les sections suivantes, nous donnons de telles vérifications pour chacun des algorithmes.

(4) Le calcul de solutions exactes et non approchées implique un nombre d'itérations plus important en calcul formel, de l'ordre de $2n$. Il est cependant possible de réduire ce nombre par des méthodes probabilistes si le rang de la matrice, ou le degré du polynôme minimal est faible. Dans la suite, pour pouvoir comparer les différentes variantes, nous notons Υ le nombre d'itérations nécessaires. En effet, nous verrons que ce nombre peut varier suivant les besoins (demande de probabilité de réussite élevée ou, au contraire, heuristique de calcul non prouvée, par exemple), mais est *identique pour toutes les variantes*, une fois ce besoin défini.

(5) Le problème suivant concerne les polynômes minimaux et le côté probabiliste des méthodes de Krylov.

Définition 6.1.2

Soit A une matrice de $\mathbb{F}^{n\times n}$ et u un vecteur de $\mathbb{F}^n$.
Le **polynôme minimal** *d'un vecteur u par rapport à A, $\pi_{u,A}(X)$, est le polynôme unitaire de degré minimal annulant u, i.e. tel que $\pi_{u,A}(A)u = 0$.*
Par extension, le polynôme minimal d'un sous-espace E par rapport à A, $\pi_{E,A}(X)$, est le polynôme unitaire de degré minimal annulant tous les vecteurs de E.

Dans un corps à q éléments, soient $\pi_{A,q}$ le polynôme minimal de A, et d_i le degré du $i^{ème}$ facteur irréductible de $\pi_{A,q}$; avec une probabilité d'au moins $\prod_i (1 - q^{-d_i})$, on a $\pi_{A,q} = \pi_{u,A,q}$ pour un vecteur u choisi au hasard [174 - Wiedemann (1986), Proposition 4]. Le polynôme minimal d'une matrice peut donc être obtenu de manière probabiliste grâce à celui d'un vecteur. La solution est alors d'utiliser l'algorithme plusieurs fois, pour augmenter la probabilité de réussite autant que nécessaire. En effet, Wiedemann assure que celle-ci augmente si l'on considère le plus petit commun multiple des polynômes générés par plusieurs exécutions

(de $\prod_i(1-q^{-d_i})$ à $\prod_i(1-q^{-kd_i})$ pour k exécutions) [174 - Wiedemann (1986), Proposition 4].

(6) Le dernier problème est d'adapter ces algorithmes pour calculer le rang. La structure générale des matrices va nous permettre cela. En effet, pour les algorithmes de Lanczos, par exemple, si la matrice est de profil général, la tridiagonalisation peut être menée jusqu'à ce qu'il ne reste plus que des zéros. Le rang est alors déduit directement. D'autre part, l'algorithme de Wiedemann calcule le polynôme minimal de la matrice. Alors, si le polynôme caractéristique n'est que le polynôme minimal de la matrice multiplié par une puissance de X, le rang se déduit du degré du polynôme minimal. Nous montrons, section 6.6, page 137, que ces deux conditions sont liées, qu'elles sont vérifiées en général et qu'il est en outre possible de les assurer en préconditionnant la matrice.

Dans les prochaines sections, notre propos est d'effectuer le décompte des opérations ; nous ne tenons donc pas toujours compte de tous ces problèmes. En particulier, la section 6.6, page 137 justifie les algorithmes pour le rang a posteriori.

6.2 ALGORITHMES DE LANCZOS

D'une manière générale, l'algorithme de Lanczos orthogonalise l'espace de Krylov associé à une matrice et un vecteur aléatoire. La thèse de Rob Lambert [103 - Lambert (1996)] comporte une étude extensive de l'algorithme de Lanczos formel. Il prouve en outre que la méthode du gradient conjugué lui est strictement identique une fois les optimisations effectuées de part et d'autre. Nous reprenons certains des algorithmes qu'il a proposés, non pas pour les redémontrer, mais pour analyser précisément leur complexités arithmétiques.

6.2.1 CAS SYMÉTRIQUE

Le premier algorithme que nous proposons est une tridiagonalisation symétrique. Il est directement inspiré de celui de Lambert [103 - Lambert (1996), Algorithme 2.2.4]. Nous en déduisons un algorithme Las Vegas pour calculer la dimension du sous espace de Krylov associé à un vecteur w, par l'ajout d'une vérification finale.

Algorithme 6.2.1 *Lanczos-Symétrique* ou tridiagonalisation

Entrées : – une matrice A symétrique de taille $n \times n$ à coeffi cients dans un corps $\mathbb{F}$.
– un vecteur $w \in \mathbb{F}^n$.

Sorties : – la dimension de $K(A, w)$ ou "échec".

Initialisation

1 : $\delta = w^T w$

2 : $v = Aw$

3 : $r = 0$

4 : $\lambda = 1$

Orthogonalisation

5 : **Tant que** $\delta \neq 0$ **Faire**

6 : $r = r + 1$

7 : $\alpha = w^T v$ 2n

8 : $w = v - \frac{\alpha}{\delta} w$ 2n

9 : $\beta = w^T w$ 2n

10 : $v = Aw - \frac{\beta}{\delta} w$ $\Omega + 2n$

11 : $\delta = \beta$

Vérifi cation

12 : **Si** $w == 0$ **Alors**

13 : dim $= r$

14 : **Sinon**

15 : "Échec"

Cet algorithme est en fait une tridiagonalisation de A car si l'on considère W la matrice contenant les vecteurs w successifs comme vecteurs colonnes, alors $W^T AW$ est tridiagonale symétrique et vaut

$$\begin{bmatrix} \alpha_1\delta_1 & \delta_2 & \dots & \\ \delta_2 & \alpha_2\delta_2 & \delta_3 & \\ & \delta_3 & \alpha_3\delta_3 & \ddots \\ & & \ddots & \ddots \end{bmatrix}$$

Le problème ici est que l'isotropie des vecteurs peut faire en sorte que δ soit nul sans que l'orthonormalisation de l'espace de Krylov soit complète (i.e. sans que $w_k = 0$) et donc cette dernière égalité n'est pas vérifiée. Nous avons donc ajouté une vérification finale testant ce fait.

Théorème 6.2.2

Soit A une matrice symétrique dans $\mathbb{F}^{n\times n}$, de polynôme minimal de degré d. L'algorithme Lanczos-Symétrique *est correct et nécessite au plus d produits matrice-vecteur par A et au plus $8dn$ opérations supplémentaires sur le corps $\mathbb{F}$. L'espace mémoire requis est celui nécessaire pour le stockage de la matrice auquel il faut ajouter $2n$.*

Preuve de 6.2.2 : En caractéristique zéro, les vecteurs w_k forment une base orthogonale de $K(A, w_1)$, d'après [74 - Golub et Van Loan (1996), Théorème 9.1.1], et donc d est une borne supérieure de la dimension de $K(A, w_1)$. Dans les corps finis, une différence est que l'on ne peut pas normer les vecteurs car on n'effectue pas de racine carrée ; les w_k ne sont plus orthonormés, ils sont seulement orthogonaux ($W^TW = diag(\delta_1, \delta_2, \dots)$). Le lecteur se reportera à [103 - Lambert (1996), Algorithme 2.2.4] pour une preuve plus formelle. Dans la boucle, les calculs sont : n multiplications et additions pour le premier produit scalaire (α) ; n multiplications et soustractions pour calculer le nouveau w ; n multiplications et additions pour le deuxième produit scalaire (β) ; un produit matrice-vecteur et n multiplications et soustractions pour calculer le nouveau v. □

Il est possible de circonvenir l'échec (*breakdown*) par la méthode de prédiction (*lookahead*) [134 - Parlett et al. (1985), 22 - Brezinski et al. (1991)] (ou [103 - Lambert (1996), Section 3.2] pour le cas formel) ; celle-ci dérive de l'algorithme de Wiedemann que nous étudions section 6.3, page 119. Notre propos étant le décompte des opérations arithmétiques, nous utilisons ici une vérification, pour ne pas ajouter trop de code. En outre, il suffit souvent de plonger le corps fini dans une de ses extensions et de choisir alors un vecteur initial dans celle-ci, pour ne pas rencontrer de problème d'isotropie.

6.2.2 CAS NON SYMÉTRIQUE

Nous donnons maintenant la version générale [103 - Lambert (1996), Algorithme 2.4.2], non symétrique, qui nécessite l'orthogonalisation simultanée de deux vecteurs.

Algorithme 6.2.3 *Lanczos* ou bi-diagonalisation

Entrées : – une matrice $A \in \mathbb{F}^{m\times n}$.
– un vecteur $v_0 \in \mathbb{F}^m$ non isotrope.

Sorties : – la dimension de $K(A^T, v_0)$ ou "échec".

Initialisation

1 : Soit $u \in \mathbb{F}^n$; $v \in \mathbb{F}^m$;
2 : $u = 0$;
3 : $\lambda = v_0^T v_0$
4 : $\rho = 1$; $\delta = -\frac{1}{\lambda}$
5 : $v = \delta v_0$
6 : $d = -1$

Bi-orthogonalisation

7 : **Tant que** $\delta \neq 0$ **Faire**
8 : $u = \frac{\delta}{\rho}u + A^T v$ $\Omega + 2n$
9 : $\theta = \frac{(u^T u)}{\delta}$ $2n$
10 : **Si** $\theta == 0$ **Alors** Arrêter.
11 : $\rho = \frac{\delta}{\theta}$
12 : $u = \frac{1}{\theta}u$ n
13 : $d = d + 1$
14 : **Si** $d > min(m, n)$ **Alors** Arrêter.
15 : $v = v + Au$ $\Omega + m$
16 : $\delta = -v^T v$ $2m$

Vérifi cations

17 : **Si** $\theta == 0$ **et** $u \neq 0$ **Alors** Renvoyer "Échec"
18 : **Si** $\delta == 0$ **et** $v \neq 0$ **Alors** Renvoyer "Échec"
19 : Renvoyer dim $= d$

Pour voir la bi-diagonalisation, il faut considérer les matrices $U, \overline{U}$ et $V, \overline{V}$. $\overline{U}$ est la matrice des vecteurs u après la ligne 8, U celle des vecteurs $\frac{1}{u^T u}u$. De même, V est la matrice des vecteurs v après la ligne 15 et $\overline{V}$ celle des $\frac{1}{v^T v}v$. On obtient alors $U\overline{U}^T = I$ et $\overline{V}^T V = I$. Et l'algorithme assure que $\overline{V}^T AU = L$ est une matrice bi-diagonale et aussi que $U^T A^T \overline{V} = L^T$.

Corollaire 6.2.4

Soit A une matrice dans $\mathbb{F}^{m\times n}$, de polynôme minimal de degré d. L'algorithme Lanczos *est correct et nécessite d produits matrice-vecteur par A, d produits matrice-vecteur par A^T et au plus $d(5n+3m)$ opérations supplémentaires sur le corps $\mathbb{F}$. L'espace mémoire requis est celui nécessaire pour le stockage de la matrice auquel il faut ajouter $n+m$.*

6.3 ALGORITHME DE WIEDEMANN

Nous étudions dans cette section l'autre variante scalaire, l'algorithme de Wiedemann. Celui-ci calcule le polynôme minimal, associé à la matrice, d'un vecteur choisi au hasard. Le polynôme minimal d'un vecteur donnant la plus petite relation annulant les produits des puissances de la matrice avec ce vecteur (définition 6.1.2, page 114), le degré de ce polynôme correspond à la dimension du sous-espace de Krylov associé. Wiedemann généralise cette notion aux séquences linéairement générées.

Définitions 6.3.1

Soit $S=(s_i)$ une séquence de scalaires.
- *Un* **générateur linéaire** *de S est un polynôme $P=\sum_{i=0}^{d} p_i X^i$ générant S, c'est-à-dire tel que $\forall k, \sum_{i=k}^{k+d} p_i s_{k+d-i}=0$.*
- *Un* **polynôme minimal** *d'une séquence de scalaires est un générateur linéaire unitaire de degré minimal.*
  Par exemple, pour une matrice A et deux vecteurs u et v, on notera $\pi_{u,A,v}$ le polynôme minimal de la séquence de scalaires $(v^T A u)$.

L'algorithme de Wiedemann projette une famille génératrice d'un sous-espace de Krylov $K(A,u)$ sur la droite vectorielle engendrée par un vecteur v. Cette famille étant linéairement générée (par le polynôme caractéristique de A, par exemple) la suite des projections est donc une suite récurrente linéaire. On peut alors utiliser l'algorithme de Berlekamp/Massey [110 - Massey (1969)] pour générer le polynôme minimal de cette suite. Avec une probabilité $\prod_i (1-q^{-kd_i})$, où q est le nombre d'éléments du corps, et d_i est le degré du $i^{ème}$ facteur irréductible de $\pi_{A,q}$, le plus petit commun multiple des polynômes minimaux de $K(v,A,u)$ pour k différents vecteurs v *est* aussi le polynôme minimal de $K(A,u)$

[174 - Wiedemann (1986), Proposition 3]. Enfin, avec une probabilité $\prod_i(1 - q^{-hd_i})$, le plus petit commun multiple des polynômes minimaux des $K(A, u)$ pour h différents vecteurs u *est* celui de la matrice [174 - Wiedemann (1986), Proposition 4]. Dans tous les cas le polynôme obtenu, après $2n$ itérations, *est un facteur* du polynôme minimal de la matrice. Nous proposons ici l'algorithme calculant le polynôme minimal de A si elle est symétrique *et*, en même temps, celui calculant le polynôme minimal de A^TA. La seule différence est le gain d'un produit matrice-vecteur dans le cas de A^TA.

Algorithme 6.3.2 *Wiedemann*

Entrées : – une matrice $A \in E^{m\times n}$, respectivement symétrique $A \in E^{n\times n}$. E est un sous-ensemble fi ni d'un corps $\mathbb{F}$.

Sorties : – Le polynôme minimal de A^tA, respectivement de A, sur $\mathbb{F}$ ou échec.

Initialisation de la séquence des projections

1 : Soit $u \in \mathbb{F}^n$ un vecteur quelconque.

2 : $S_0 = u^Tu$

Initialisation de Berlekamp/Massey

3 : $b = 1; e = 1; L = 0; \varphi = 1 \in \mathbb{F}[X]; \psi = 1 \in \mathbb{F}[X];$

4 : **Pour** $k = 0$ **jusqu'à** $2\ min(m, n)$ **Faire**

Récurrence linéaire par Berlekamp/Massey

5 : $\delta = S_k + \sum_{i=1}^{L} \varphi_i S_{k-i}$ $2\frac{k}{2}$

6 : **Si** $(\delta == 0)$ **Alors**

7 : $++e$

8 : **Sinon, si** $2L > k$ **Alors**

9 : $\varphi = \varphi - \frac{\delta}{b}X^e\psi$ $2\frac{k}{2}$

10 : $++e$

11 : **Sinon**

12 : $\varphi = \varphi - \frac{\delta}{b}X^e\psi$ // $\psi = \varphi$ $2\frac{k}{2}$

13 : $L = k + 1 - L; b = \delta; e = 1$

Terminaison anticipée

14 : **Si** $e > SeuilTerminaison$ **Alors** Arrêter.

[Calcul du prochain coeffi cient, cas non-symétrique]

15 a : **Si** k pair **Alors**

16 a : $v = Au$ Ω

17 a : $S_{k+1} = v^T v$ 2m

18 a : **Sinon** {ou}

19 a : $u = A^T v$ Ω

20 a : $S_{k+1} = u^T u$ 2n

[Calcul du prochain coeffi cient, cas symétrique]

15 b : **Si** k pair **Alors**

16 b : $v = Au$ Ω

17 b : $S_{k+1} = u^T v$ 2n

18 b : **Sinon** {ou}

19 b : $u = v$

20 b : $S_{k+1} = u^T u$ 2n

Vérifi cation

21 : Soit $z \in \mathbb{F}^n$ un vecteur quelconque.

[Appliquer φ à $A^t A$ et z, cas non-symétrique]

22 a : $w = \varphi(A^t A).v$

[Appliquer φ à A et z, cas symétrique]

22 b : $w = \varphi(A).v$

23 : **Si** $w == 0$ **Alors**

24 : Renvoyer φ.

25 : **Sinon**

26 : “Échec”

Si le nombre de ces itérations est d'au plus $2n$, il est possible de terminer l'algorithme par anticipation [87 - Kaltofen et al. (2000)]. En effet, si, par exemple, le polynôme calculé reste inchangé pendant plusieurs itérations (*SeuilTerminaison*), il y a de bonnes chances que ce soit *le* polynôme minimal. Dans ce cas, nous arrêtons le calcul et vérifions ce fait. Cette méthode est totalement heuristique. Néanmoins, elle est justifiée d'une part grâce à l'analogie avec l'algorithme de Lanczos et d'autre part grâce aux préconditionnements de la section 6.6, page 137. En effet, en caractéristique zéro, δ ne s'annule que lorsque l'algorithme est terminé. L'algorithme de Wiedemann est alors l'algorithme de Lanczos avec prise en compte du fait que δ peut s'annuler, dans un corps fini, sans que l'on soit au terme des itérations.

6.3.1 CAS NON SYMÉTRIQUE

Théorème 6.3.3

Soit A une matrice de $\mathbb{F}^{m\times n}$, de polynôme minimal de degré d. L'algorithme Wiedemann *non symétrique nécessite au plus $2d$ produits matrice-vecteur par A et au plus $4d^2+2d(n+m)$ opérations supplémentaires sur le corps $\mathbb{F}$. L'espace mémoire requis est celui nécessaire pour le stockage de la matrice auquel il faut ajouter $n+m+3\ min(m,n)$. Avec vérification, il faut un supplément de $2dn$ opérations sur le corps et $2d$ produits matrice-vecteur ou dn unités de mémoire.*

Preuve de 6.3.3 : Pour la complexité arithmétique : la boucle s'arrête avec terminaison anticipée quand le degré du polynôme φ reste inchangé. Son degré est inférieur à celui du polynôme minimal de A, puisque c'en est un facteur. C'est à dire quand $k = 2d$. Cette boucle a le coût suivant : $\frac{k}{2}$ multiplications et additions pour le calcul de la divergence δ ; $\frac{k}{2}$ multiplications et soustractions pour la mise à jour des polynômes, $\frac{k}{2}$ étant une borne sur les degrés de φ et ψ ; un produit matrice-vecteur ; n ou m multiplications et additions pour le calcul du produit scalaire.

Pour l'espace mémoire nécessaire : il faut stocker v et u, les deux polynômes φ et ψ ainsi que la séquence S. Un troisième polynôme n'est pas nécessaire pour effectuer les opérations sur φ et ψ. En effet, il est possible de calculer en place la ligne 13 de l'algorithme 6.3.2, page 120, $\varphi = \varphi - \delta b^{-1} X^x \psi \ \ // \ \ \psi = \varphi$, de la manière suivante. En réservant un espace de $2n$, dès le départ pour les deux polynômes φ et ψ, en tirant parti du fait que $x \geq 1$ et en commençant par les degrés les plus grands, il est possible de mettre à jour $\psi[i]$ pendant le calcul de $\varphi[i]$ puisque celui-ci ne nécessite que la valeur de $\psi[i-x]$.

Enfin, la vérification consiste à appliquer le polynôme sur la suite des vecteurs $A^i z$. Une première possibilité est de les conserver (dn unités mémoire) et le calcul se fait directement avec les coefficients du polynôme. Sinon, la solution est reconstruite en effectuant de nouveau des produits matrice-vecteur. On en déduit les $2dn$ opérations et les $2d$ produits matrice-vecteur supplémentaires. □

6.3.2 CAS SYMÉTRIQUE

Corollaire 6.3.4

Soit A une matrice symétrique dans $\mathbb{F}^{n\times n}$, de polynôme minimal de degré d. L'algorithme Wiedemann *symétrique nécessite au plus d produits matrice-vecteur par A et $4d^2 + 4dn$ opérations supplémentaires sur le corps $\mathbb{F}$. L'espace mémoire requis est celui nécessaire pour le stockage de la matrice auquel il faut ajouter $5n$. Avec vérification, il faut un supplément de $2dn$ opérations sur le corps et d produits matrice-vecteur ou dn unités de mémoire.*

Preuve de 6.3.4 : L'emploi du même vecteur de projection à gauche et à droite permet de diviser par deux le nombre de produits matrice-vecteur à effectuer. En effet, dans ce cas $u_0^T A^{2i} u_0 = (A^i u_0)^T (A^i u_0)$ et $u_0^T A^{2i+1} u_0 = (A^i u_0)^T (A^{i+1} u_0)$. □

6.4 ALGORITHMES DE LANCZOS PAR BLOCS

Dans cette section et la suivante, nous nous intéressons aux versions des algorithmes précédents utilisant plusieurs vecteurs initiaux en même temps. Cela permet de regrouper les calculs par blocs et de réduire les itérations. Leur utilité est double ; d'une part, cela améliore les probabilités de réussite dans les petits corps finis [170 - Villard (1997c)] et d'autre part, cela augmente le parallélisme de l'algorithme tout en conservant le même nombre de produits matrice-vecteur. Nous allons voir cependant que le nombre total d'opérations augmente.

6.4.1 LANCZOS NUMÉRIQUE PAR BLOCS

Les algorithmes par blocs ont été largement étudiés dans la littérature numérique [76 - Grimes et al. (1994), 131 - O'Leary (1980), 148 - Saad (1987)]. En calcul exact, les algorithmes ont été étudiés dans le corps fini à deux éléments, $\mathtt{GF}(2)$ [40 - Coppersmith (1993), 115 - Montgomery (1995)] mais sont utilisables dans d'autres corps. Nous proposons, pour mieux comprendre le fonctionnement de cet algorithme, une adaptation de la version numérique de tridiagonalisation par blocs [74 - Golub et Van Loan (1996), §9.2.6] au cas formel, pour tout corps.

Celui-ci considère la tridiagonalisation orthonormale de A (les B_k sont triangulaires supérieures) :

$$Q^T AQ = T = \begin{bmatrix} M_1 & B_2^T & 0 & \dots & 0 \\ B_2 & M_2 & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & B_{\overline{p}}^T \\ 0 & \dots & 0 & B_{\overline{p}} & M_{\overline{p}} \end{bmatrix} \text{ avec } Q^T Q = I \qquad (6.1)$$

Si l'on pose $Q = [X_1, \dots, X_{\overline{p}}]$, en regroupant les colonnes de A par blocs de taille p, l'équation 6.1, se transforme en $AQ = QT$ et donne la récurrence à trois termes : $AX_k = X_{k+1}B_{k+1} + X_k M_k - X_{k-1}B_k^T$. En outre, l'orthogonalité de Q nous donne $M_k = X_k^T AX_k$. Pour rendre plus claire cette explication, nous donnons l'algorithme numérique complet.

Algorithme 6.4.1 *Bloc-Lanczos-numérique*

Entrées : – une matrice A symétrique de taille $n \times n$ à coeffi cients dans $\mathbb{C}$.
– la taille p des blocs, $n = p\overline{p}$.

Sorties : – Tridiagonalisation par blocs de A sur $\mathbb{C}$.

Initialisations

1 : $X_0 = 0$; $B_1 = 0$;

2 : Soit $X_1 \in \mathbb{C}^{n \times p}$ un bloc de vecteurs tels que $X_1^T X_1 = I_p$.

Récurrence

3 : $M_1 = X_1^T AX_1$

4 : **Pour** $k = 1$ **jusqu'à** $\overline{p} - 1$ **Faire**

5 : $R_k = AX_k - X_k M_k - X_{k-1}B_k^T$

6 : $X_{k+1}B_{k+1} = R_k$ {Factorisation *QR* de R_k}

7 : $M_{k+1} = X_{k+1}^T AX_{k+1}$

6.4.2 LANCZOS FORMEL PAR BLOCS, CAS SYMÉTRIQUE

Par analogie avec le travail de R. Lambert sur l'algorithme classique, nous pouvons dériver une adaptation sans racine carrée (i.e. sans factorisation *QR*). Il

faut considérer cette fois la tridiagonalisation par blocs en relâchant la contrainte d'orthonormalisation en une contrainte d'orthogonalisation par blocs : $R^T R = B = diag(B_1, \ldots, B_{\overline{p}})$ avec $\forall k, B_k \in \mathbb{F}^{p\times p}$, *symétrique inversible*, et $RR^T = \tilde{B}$. Il y a là une différence, par rapport au cas numérique classique : au lieu d'avoir des matrices B_k triangulaires supérieures, nous avons des matrices symétriques. Cela se comprend en considérant qu'une matrice symétrique est en fait le produit par sa transposée d'une matrice triangulaire (factorisation de Cholesky, avec racines carrées). L'information est donc bien la même, simplement nous n'effectuons pas de racines carrées.

Dans la suite, nous aurons besoin de calculer avec $\tilde{B}$ mais, à l'inverse de B, $\tilde{B}$, si elle est symétrique, peut ne pas être diagonale par blocs. Prenons, par exemple :

$$A = \left[\begin{smallmatrix} 2 & -2 \\ 1 & 4 \end{smallmatrix}\right] \text{ et } A^T A = \left[\begin{smallmatrix} 5 & 0 \\ 0 & 20 \end{smallmatrix}\right] \text{ et } AA^T = \left[\begin{smallmatrix} 8 & -6 \\ -6 & 17 \end{smallmatrix}\right]$$

$A^T A$ est bien diagonale mais AA^T ne l'est pas ! Toutefois, il est possible d'expliciter les produits de $\tilde{B}$ avec les colonnes de R :

Lemme 6.4.2

Soit $R \in F^{m\times n}$, $n = p\overline{p}$, une matrice orthogonale par blocs à gauche, i.e. $R^T R = diag(B_1, \ldots, B_{\overline{p}})$ et $\forall k, B_k \in F^{p\times p}$ inversible.
Si $R = [W_1, \ldots, W_{\overline{p}}]$, groupée par blocs de colonnes, et $RR^T = \tilde{B}$ alors on a $\forall k \in [\![1, \overline{p}]\!]$:

1. $\tilde{B}W_k = W_k B_k$.
2. $W_k^T \tilde{B} = B_k W_k^T$.

Si de plus $m = n$ alors $\tilde{B}$ est inversible et on a $\forall k \in [\![1, \overline{p}]\!]$:

3. $\tilde{B}^{-1} W_k = W_k B_k^{-1}$.
4. $W_k^T \tilde{B}^{-1} = B_k^{-1} W_k^T$.

Preuve de 6.4.2 :

1. $R^T W_k = [0, \ldots, B_k, \ldots, 0]^T$ donc $R(R^T W_k) = W_k B_k$.
2. On transpose 1.

Si $m = n$, R est carrée et $R^T R$ est inversible donc R et R^T puis $\tilde{B}$ le sont aussi.

3. $\tilde{B}^{-1}(W_k B_k)B_k^{-1} = \tilde{B}^{-1}(\tilde{B}W_k)B_k^{-1}$ d'après 1.
4. On transpose 3.

□

Nous supposons maintenant qu'en relâchant la contrainte d'orthonormalité, nous obtenons la factorisation suivante :

$$R^T AR = \tilde{T} = \begin{bmatrix} M_1 & B_2 & 0 & \dots & 0 \\ B_2 & M_2 & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & B_{\overline{p}} \\ 0 & \dots & 0 & B_{\overline{p}} & M_{\overline{p}} \end{bmatrix} \tag{6.2}$$

En écrivant de la même façon que précédemment, on trouve $\tilde{B}AR = R\tilde{T}$ et une récurrence à trois termes :

$$\tilde{B}AW_k = W_{k+1}B_{k+1} + W_k M_k + W_{k-1}B_k \tag{6.3}$$

On utilise ensuite le lemme 6.4.2, page 125 :

- En multipliant cette récurrence à gauche par W_{k-1}^T, on obtient la relation : $B_{k-1}(W_{k-1}^T AW_k) = B_{k-1}B_k$. D'où, avec B_{k-1} inversible, l'on tire $B_k = W_{k-1}^T AW_k$.
- En multipliant à gauche la relation par W_k^T, cette fois-ci, on obtient la même définition pour M_k : $B_k(W_k^T AW_k) = B_k(M_k)$.
- On peut donc maintenant écrire la récurrence $W_{k+1} - AW_k = \tilde{B}^{-1}(-W_k M_k - W_{k-1}B_k)$ en remarquant que $W_{k+1}B_{k+1} = \tilde{B}W_{k+1}$. En utilisant le lemme, on voit alors que $W_{k+1} - AW_k = -W_k B_k^{-1} M_k - W_{k-1}B_{k-1}^{-1}B_k$.

En posant $V_k = AW_k - W_{k-1}B_{k-1}^{-1}B_k$ pour réduire l'espace mémoire nécessaire, on en déduit l'algorithme suivant :

1. $B_k = W_{k-1}^T AW_k = W_k^T AW_{k-1} = W_k^T W_k$
2. $V_k = AW_k - W_{k-1}B_{k-1}^{-1}B_k$
3. $M_k = W_k^T AW_k = W_k^T V_k$
4. $W_{k+1} = V_k - W_k B_k^{-1} M_k$

Cela donne une fois les optimisations d'espace effectuées :

Algorithme 6.4.3 *Tridiagonalisation-Formelle-par-blocs*

Entrées : – une matrice A symétrique de taille $n \times n$ à coeffi cients dans un corps $\mathbb{F}$.
– la taille p des blocs, $n = p\overline{p}$.

Sorties : – Une tridiagonalisation par blocs de A sur $\mathbb{F}$ ou "échec".

Initialisation

1 : Soit $W \in \mathbb{F}^{n\times p}$ un bloc de vecteurs quelconques.

2 : $B = W^T W$

3 : $V = AW$

4 : **Tant que** B est inversible **Faire**

5 :	$LU = B$	$\mathsf{LU(p)}$
6 :	$W_0 = W$	
7 :	$M = W^T V$	$2\mathsf{p}^2\mathsf{n}$
8 :	$W = V - W_0 U^{-1} L^{-1} M$	$\mathsf{pn} + 2\mathsf{pTR(p)} + 2\mathsf{p}^2\mathsf{n}$
9 :	$B = W^T W$	$2\mathsf{p}^2\mathsf{n}$
10 :	$V = AW - W_0 U^{-1} L^{-1} B$	$\mathsf{p}\Omega + \mathsf{pn} + 2\mathsf{pTR(p)} + 2\mathsf{p}^2\mathsf{n}$

Vérifi cation

11 : **Si** Aucune ligne de W n'est nulle **Alors** Renvoyer "Échec".

Théorème 6.4.4

Soit A une matrice symétrique dans $\mathbb{F}^{n\times n}$. L'algorithme Tridiagonalisation-Formelle-par-blocs *est correct et nécessite $\Upsilon \leq n$ produits matrice-vecteur par A et au plus*

$$8\Upsilon pn + \frac{7}{3}\Upsilon p^2 + 2\Upsilon(n+p)$$

opérations supplémentaires sur le corps $\mathbb{F}$. L'espace mémoire requis est celui nécessaire pour le stockage de la matrice auquel il faut ajouter $3pn + 2p^2$.

Preuve de 6.4.4 : Il faut prouver que notre supposition est fondée, c'est-à-dire que les W_k calculés par l'algorithme sont bien orthogonaux à gauche par blocs et que la matrice $R^T AR$ est bien la matrice tridiagonale par blocs de l'équation 6.2, page 126. On procède par une quadruple récurrence avec l'hypothèse suivante, à l'étape k :

1. $\forall 1 \leq i < k, W_i^T W_k = 0$, orthogonalité à gauche.
2. $\forall 1 \leq i < (k-1), W_i^T A W_k = 0$, tridiagonalité.
3. $M_k = W_k^T A W_k$, la diagonale est correcte.
4. $B_k = B_k^T = W_{k-1}^T A W_k$, les deux bandes autour de la diagonale sont correctes.

À l'étape suivante, on obtient alors :

1. – si $i = k, W_k^T(W_{k+1}) = W_k^T(AW_k - W_{k-1}B_{k-1}^{-1}B_k - W_kB_k^{-1}M_k)$. Or la récurrence 1 donne $W_k^TW_{k-1} = 0$ et l'algorithme pose $B_k = W_k^TW_k$ donc $W_k^TW_{k+1} = W_k^TAW_k - M_k$. Or l'algorithme pose $M_k = W_k^TV_k = W_k^TAW_k$ par la récurrence 3, et l'on obtient bien $W_k^TW_{k+1} = 0$.
   – De même, si $i = k-1, W_{k-1}^TW_{k+1} = W_{k-1}^TAW_k - B_k$ par la récurrence 1 or $B_k = W_{k-1}^TAW_k$ par la récurrence 4.
   – Enfin, si $1 \leq i < k-1, W_i^TW_{k+1} = W_i^TAW_k = 0$ par les récurrences 1 puis 2.
2. si $i < k, (W_i^TAW_{k+1})^T = W_{k+1}^TAW_i = W_{k+1}^T(W_{i+1} + W_iH + W_{i-1}G) = 0 + 0 + 0$ par l'orthogonalité.
3. $M_{k+1} = W_{k+1}^TV_{k+1} = W_{k+1}^T(AW_{k+1} - W_kF) = W_{k+1}^TAW_{k+1}$ par l'orthogonalité.
4. $B_{k+1} = W_{k+1}^TW_{k+1}$ donc B_{k+1} est bien symétrique. En outre $W_{k+1}^TW_{k+1} = W_{k+1}^T(AW_k - W_kH - W_{k-1}G) = W_{k+1}^TAW_k$ par l'orthogonalité.

Pour la complexité, nous notons $\lceil \frac{\Upsilon}{p} \rceil$ le nombre d'itérations ; $LU(p)$ est borné par $\frac{1}{3}p^3$ et $TR(p)$, une résolution de système triangulaire, est bornée par $\frac{1}{2}p^2$. □

De même que pour l'algorithme de Lanczos classique, les problèmes d'isotropie apparaissent ici lorsque B_k n'est pas inversible. Pour ce problème, il est possible d'adapter les méthodes de prédiction au cas par blocs [40 - Coppersmith (1993)] ; pour le cas non symétrique, l'adaptation est similaire au cas scalaire de la section 6.2, page 115. Nous ne l'étudions pas en détails, le nombre d'opérations supplémentaires sera de toute façon au moins celui de l'algorithme symétrique, et le nombre de produits matrice-vecteur sera multiplié par 2.

6.5 POLYNÔMES GÉNÉRATEURS MATRICIELS

L'algorithme de Wiedemann par blocs développé par Coppersmith [41 - Coppersmith (1994)] est une généralisation de l'algorithme de Wiedemann. La suite des projections scalaires de A, les $s_i = vA^iu$, se généralise à une suite de matrices (les blocs) VA^iU, où V et U sont des blocs de vecteurs. Le problème ici est de trouver un polynôme générateur de cette suite de matrices. Une première idée est de généraliser l'algorithme de Berlekamp/Massey pour des matrices [41 - Coppersmith (1994)] ; elle a été utilisée par A. Lobo [109 - Lobo (1995), 88 - Kaltofen et Lobo (1996)] pour la factorisation d'entiers. Cette méthode est très proche de l'algorithme de Lanczos par blocs 6.4.3, page 126 avec prédiction. Une autre idée est de considérer la recherche du polynôme générateur comme la

solution d'un système linéaire de Toeplitz par blocs [83 - Kaltofen (1995)] ; M. Doreille, par exemple, utilise cette méthode pour paralléliser efficacement la résolution de systèmes linéaires structurés [48 - Doreille (1995)]. Enfin, la méthode de G. Villard [170 - Villard (1997c), 169 - Villard (1997b)] consiste à utiliser une résolution par les approximants de Padé de B. Beckermann et G. Labahn [9 - Beckermann et Labahn (1994)] ; Penninga [135 - Penninga (1998)] a comparé cette méthode avec l'algorithme de A. Lobo, dans le cas de la résolution de systèmes lineaires.

Dans tous les cas, la méthode de Wiedemann par blocs est identique et se déduit directement de l'algorithme classique si l'on considère l'extension de la définition de générateur linéaire au cas matriciel. La notion de minimalité, toutefois, n'est plus valable. Il faudra assurer que le polynôme obtenu est sous **forme de Popov** [170 - Villard (1997c), Section 2] :

Définition 6.5.1

Soit $D(X)$ une matrice polynomiale. Pour tout j, soit d_j le degré de la colonne j. $D(X)$ est **sous forme de Popov** *si les conditions suivantes sont réunies :*

1. *Les colonnes sont ordonnées par degrés croissants*
2. *La matrice scalaire formée par les vecteurs des coefficients du terme de plus haut degré de chaque colonne est de même rang que $D(X)$.*
3. *Le dernier élément de degré d_j dans chaque colonne est unitaire. Cet élément est appelé le pivot de la colonne j avec un indice de ligne r_j.*
4. *Si $d_j = d_k$ et $j < k$ alors $r_j < r_k$.*
5. *Le pivot est l'élément de plus haut degré dans sa ligne.*

Le lecteur se reportera à [170 - Villard (1997c)] pour plus de détails sur la forme de Popov et le lien avec l'algorithme de Wiedemann par blocs.

Algorithme 6.5.2 *Wiedemann-par-blocs*

Entrées : – un sous-ensemble fini E d'un corps $\mathbb{F}$.
– une matrice $A \in E^{m \times n}$, respectivement symétrique $A \in E^{n \times n}$.
– un bloc de vecteurs $U \in \mathbb{F}^{n \times p}$, $n = pq$.

Sorties : – Un polynôme matriciel générateur de la séquence $\{U^T U, U^T A^T A U, \dots\}$, respectivement générateur de la séquence $\{U^T U, U^T A U, \dots\}$

Initialisation de la séquence des projections

1 : $S_0 = U^T U$

2 : **Pour** $k = 0$ **jusqu'à** $2\lceil \frac{n}{p} \rceil$ **Faire**

Récurrence linéaire par blocs

3 : Calcul d'un nouveau générateur prenant en compte la projection S_k.

Terminaison anticipée

4 : **Si** Le générateur reste inchangé **Alors** Arrêter.

[Calcul du prochain coeffi cient, cas non-symétrique]

5 a : **Si** k pair **Alors**

6 a : $V = AU$ $\mathsf{p\Omega}$

7 a : $S_{k+1} = V^T V$ $\mathsf{2p^2m}$

8 a : **Sinon** {ou}

9 a : $U = A^T V$ $\mathsf{p\Omega}$

10 a : $S_{k+1} = U^T U$ $\mathsf{2p^2n}$

[Calcul du prochain coeffi cient, cas symétrique]

5 b : **Si** k pair **Alors**

6 b : $V = AU$ $\mathsf{p\Omega}$

7 b : $S_{k+1} = U^T V$ $\mathsf{2p^2n}$

8 b : **Sinon** {ou}

9 b : $U = V$

10 b : $S_{k+1} = U^T U$ $\mathsf{2p^2n}$

La génération d'un bloc de la séquence de matrices dans l'algorithme nécessite $2p$ produits matrice-vecteur dans le cas symétrique ou p produits matrice-vecteur dans le cas non-symétrique. Elle nécessite en outre $p^2(n+m)$ opérations supplémentaires sur le corps. L'espace mémoire requis est celui nécessaire pour le stockage de la matrice auquel il faut ajouter $p(n+m)+p^2$.

Nous étudions maintenant la méthode de Coppersmith pour calculer ce polynôme générateur matriciel.

6.5.1 Algorithme de Coppersmith

Avant de passer à la version par blocs de l'algorithme de Berlekamp/Massey, nous montrons l'équivalence entre la version de Massey et la version par blocs de

Coppersmith dans le cas où les blocs sont de taille 1. En posant, dans l'algorithme 6.3.2, page 120, $d_\varphi = L$, $d_\psi = k - L$ (on a alors, à la fin de chaque itération, d°$\varphi \leq d_\varphi$ et d°$\psi \leq d_\psi$, d_φ et d_ψ sont appelés les degrés nominaux de φ et ψ), puis en remplaçant l'incrémentation de l'exposant e par une multiplication de ψ par X on obtient le code suivant (nous avons repris les numéros de ligne de l'algorithme 6.3.2, page 120 pour plus de clarté) :

Algorithme 6.5.3 *Berlekamp/Massey version Coppersmith*

Initialisation de Berlekamp/Massey

3 : $b = 1; d_\varphi = 0; d_\psi = 0 - 0; \varphi = 1 \in \mathbb{F}[X]; \psi = 1 \in \mathbb{F}[X]$

4 : **Pour** $k = 0$ **jusqu'à** $2\ min(m, n)$ **Faire**

Récurrence linéaire par Berlekamp/Massey

5 : $\delta = S_k + \sum_{i=1}^{d_\varphi} \varphi_i S_{k-i}$

6 : $\psi = X\psi$

7 : **Si** $(\delta \neq 0)$ **Alors**

8 : **Si** $d_\varphi > d_\psi$ **Alors**

9 : $\varphi = \varphi - \frac{\delta}{b}\psi$

10 : **Sinon**

11 : $\varphi = \varphi - \frac{\delta}{b}\psi$ // $\psi = \varphi$

12 : $b = \delta$

13 : $d_\varphi = d_\psi$ // $d_\psi = d_\varphi$

14 : $++d_\psi$

De cet algorithme, Coppersmith déduit alors l'algorithme matriciel à l'aide des degrés nominaux [41 - Coppersmith (1994)]. En effet, nous considérons maintenant $\Phi(X) = \big(\varphi_{ij}(X)\big) = \sum \Phi_l X^l$ et $\Psi = \big(\psi_{ij}(X)\big) = \sum \Psi_l X^l$ comme des polynômes matriciels et D_Φ et D_Ψ sont des vecteurs de degrés nominaux sur les lignes de ces matrices, i.e. tels que $\forall j$, d°$\varphi_{ij} \leq D_\Phi[i]$ et $\forall j$, d°$\psi_{ij} \leq D_\Psi[i]$.

Nous proposons maintenant une nouvelle version, optimisée, de l'algorithme de Coppersmith. Dans la suite, nous montrons que cet version est la méthode itérative par blocs qui requiert actuellement le moins d'opérations supplémentaires.

Algorithme 6.5.4 *Coppersmith*

Entrées : – Une suite S de matrices $p \times q$ linéairement générées par un polynôme de degré au plus N.

– h le nombre de lignes du générateur linéaire.

Sorties : – $\Phi \in \mathbb{F}^{h\times p}[X]$ un générateur linéaire pour S.

1 : $\Phi = 1 \in \mathbb{F}^{h\times p}[X]$; $\Psi = 1 \in \mathbb{F}^{q\times p}[X]$

2 : $D_\Phi = 0 \in \mathbb{F}^h$; $D_\Psi = 0 \in \mathbb{F}^q$; $B = I_q \in \mathbb{F}^{q\times q}$;

3 : **Pour** $k = 0$ **jusqu'à** $2N$ **Faire**

4 : $\quad \Delta = (S_k + \sum_{l=1}^{||D_\Phi||_\infty} \Phi_l S_{k-l}) \in \mathbb{F}^{h\times q}$ $\quad 2\,\mathsf{p}\,\mathsf{h}\,\mathsf{q}\,\frac{\mathsf{k}}{2}$

5 : $\quad \Psi = X\Psi$

6 : $\quad$ **Pour** $j = 1$ **jusqu'à** q **Faire**

7 : $\quad\quad \Lambda_\Phi = \{i, \Delta_{ij} \neq 0\}$

8 : $\quad\quad$ **Si** $\Lambda_\Phi \neq \emptyset$ **Alors**

9 : $\quad\quad\quad$ Sélectionner $i_\varphi \in \Lambda_\Phi$ tel que $D_\Phi[i_\varphi]$ soit minimal.

10 : $\quad\quad\quad$ **Si** $D_\Phi[i_\varphi] > D_\Psi[j]$ **Alors**

11 : $\quad\quad\quad\quad$ **Pour tout** $i \in \Lambda_\Phi$ **Faire** {Opérations sur les lignes de Φ}

12 : $\quad\quad\quad\quad\quad \Phi[i] = \Phi[i] - \frac{\Delta_{ij}}{B_{jj}}\Psi[j]$ $\quad 2\,\frac{\mathsf{k}}{2}\,\mathsf{p}$

13 : $\quad\quad\quad\quad\quad \Delta[i] = \Delta[i] - \frac{\Delta_{ij}}{B_{jj}}B[j]$ $\quad 2\,(\mathsf{q}-\mathsf{j})$

14 : $\quad\quad\quad$ **Sinon**

15 : $\quad\quad\quad\quad$ **Pour tout** $i \neq i_\varphi \in \Lambda_\Phi$ **Faire** {Opérations sur les lignes de Φ}

16 : $\quad\quad\quad\quad\quad \Phi[i] = \Phi[i] - \frac{\Delta_{ij}}{\Delta_{i_\varphi j}}\Phi[i_\varphi]$ $\quad 2\,\frac{\mathsf{k}}{2}\,\mathsf{p}$

17 : $\quad\quad\quad\quad\quad \Delta[i] = \Delta[i] - \frac{\Delta_{ij}}{\Delta_{i_\varphi j}}\Delta[i_\varphi]$ $\quad 2\,(\mathsf{q}-\mathsf{j})$

18 : $\quad\quad\quad\quad \Psi[j] = \Psi[j] - \frac{B_{jj}}{\Delta_{i_\varphi j}}\Phi[i_\varphi]$ $\quad 2\,\frac{\mathsf{k}}{2}\,\mathsf{p}$

19 : $\quad\quad\quad\quad B[j] = B[j] - \frac{B_{jj}}{\Delta_{i_\varphi j}}\Delta[i_\varphi]$ $\quad 2\,(\mathsf{q}-\mathsf{j})$

20 : $\quad\quad\quad\quad$ Échanger $\Phi[i_\varphi]$ et $\Psi[j]$

21 : $\quad\quad\quad\quad$ Échanger $\Delta[i_\varphi]$ et $B[j]$

22 : $\quad\quad\quad\quad$ Échanger $D_\Phi[i_\varphi]$ et $D_\Psi[j]$

23 : $\quad\quad$ $++D_\Psi$

Une interprétation matricielle de cet algorithme est due à A. Lobo [109 - Lobo (1995)]. Elle permet de donner une meilleure idée du déroulement de l'algorithme. En considérant, à chaque étape, la matrice construite avec B en partie supérieure et Δ en partie inférieure, il s'agit de trouver une transformation T_k telle que

$$T_k \begin{bmatrix} B \\ \Delta \end{bmatrix} = \begin{bmatrix} U \\ 0 \end{bmatrix}$$

avec U triangulaire et ordonnée de telle sorte que les degrés nominaux correspondants soient en ordre croissant. Cela n'est possible en général que si on a au moins q lignes dans B, d'où le choix de ce nombre dans notre algorithme. Cependant, à la différence de A. Lobo, non seulement nous conservons B triangulaire d'étape en étape et n'avons donc pas besoin d'effectuer les opérations de triangularisation sur les lignes de B, mais en plus, comme les lignes de B sont les anciennes lignes de Δ, nous n'avons pas besoin de recalculer B à chaque étape. Il reste ensuite, pour mettre à jour les polynômes, à effectuer les transformations sur Φ et Ψ :

$$\begin{bmatrix} \Psi \\ \Phi \end{bmatrix} = \begin{bmatrix} X\,I_q & 0 \\ 0 & I_h \end{bmatrix} T_k \begin{bmatrix} \Psi \\ \Phi \end{bmatrix}$$

À l'aide de cette formalisation, on voit en outre que tant que B est inversible, il est possible de retrouver les mêmes opérations que pour l'algorithme de Lanczos par blocs sans prédiction 6.4.3, page 126. Il faut considérer, après les permutations de lignes induites,

$$T_k = \begin{bmatrix} L^{-1} & 0 \\ -\Delta U^{-1} L^{-1} & I_h \end{bmatrix}$$

avec L et U triangulaires inférieure et supérieure telles que $B = LU$. En posant $\Delta = W_0$, on retrouve bien les mêmes opérations : la factorisation LU, et les deux multiplications par $W_0 U^{-1} L^{-1}$.

Théorème 6.5.5

L'algorithme Coppersmith *est correct et nécessite au plus*

$$4\ h\ p\ q\ N^2 + 2\ h\ (q-p)\ q\ N$$

opérations sur le corps. L'espace mémoire requis est celui nécessaire pour la séquence auquel il faut ajouter $2h(q+pN)$.

Preuve de 6.5.5 : Le calcul de Δ nécessite $||D_\Phi||_\infty \leq \frac{k}{2}$ produits de matrices $h \times p$ par $p \times q$, i.e. $(2p-1)(hq)(\frac{k}{2})$ opérations, et le même nombre d'additions

de matrices $h \times q$, d'où un total de moins de $(2p)hq\frac{k}{2}$ opérations par itération et donc $phqN(2N-1)$ pour les $2N$ itérations.

La mise à jour d'un élément de Φ ou Ψ, c'est-à-dire d'un polynôme de degré au plus $\frac{k}{2}$, nécessite $2\frac{k}{2}$ opérations. Donc, pour une ligne de longueur p (les matrices Φ et Ψ ont p colonnes), il faut $kp+2(q-j)$ opérations en comptant les opérations sur les matrices Δ et B. Il y a h lignes à mettre à jour, d'où $h(kp+2(q-j))$ opérations par colonne mise à zéro. Ensuite, il faut effectuer ceci pour les q colonnes, d'où $\sum_{j=1}^{q} h(kp+2(q-j)) = hq(kp+q-1)$ opérations par itération. Enfin, il y a $2N$ itérations au plus d'où un total de $\sum_{k=0}^{2N-1} hq(kp+q-1) = hqN(2Np+2q-p-2)$. Pour finir, le nombre d'opérations est la somme de ces deux totaux et $hqN(2Np+2q-p-2)+phqN(2N-1) \leq 4hpqN^2+2hqN(q-p)$. Pour l'espace mémoire, il faut $2hq$ pour Δ et B, ainsi que $2hpN$ pour Φ et Ψ. □

Nous revenons à l'algorithme de Wiedemann. Il faut alors choisir le paramètre h. Pour obtenir un générateur minimal, nous avons vu qu'il doit être sous forme de Popov. Notre algorithme garantit les conditions sur les degrés nominaux, qui rejoignent les conditions sur les degrés des pivots de la forme de Popov. Enfin, la forme de Popov doit être inversible et carrée et donc $h = p$, ce qui rejoint aussi les conditions de Coppersmith [41 - Coppersmith (1994), section 3] pour que le nombre d'itérations soit en $n/p + n/q + \mathcal{O}(1)$. Pour comparer avec l'algorithme de Lanczos, nous considérons alors que $p = q$ et que $2\frac{\Upsilon}{p}$ est le nombre d'itérations nécessaires.

Corollaire 6.5.6

L'algorithme Wiedemann-par-blocs *non symétrique nécessite* $2\,\Upsilon \leq n$ *produits matrice-vecteur par* A *et au plus*

$$2p\Upsilon(n+m) + 4\Upsilon^2 p$$

opérations supplémentaires sur le corps. L'espace mémoire supplémentaire requis est de $2p(n+m) + 2pd + 2p^2$*, où* d *est le degré maximal des polynômes.*

Corollaire 6.5.7

L'algorithme Wiedemann-par-blocs *symétrique nécessite* $\Upsilon \leq n$ *produits matrice-vecteur par* A *et au plus*

$$4\Upsilon pn + 4\Upsilon^2 p$$

opérations supplémentaires sur le corps. L'espace mémoire supplémentaire requis est de $4pn + 2pd + 2p^2$.

Preuve de 6.5.6 et 6.5.7 : Le nombre d'itérations est $2N = 2\frac{\Upsilon}{p}$, et nous avons choisi $h = q = p$ puisque nous considérons le même bloc de vecteurs de projection U à gauche et à droite. Le calcul du polynôme générateur nécessite donc au plus $4p\Upsilon^2$ opérations auxquelles il faut ajouter les $2\frac{\Upsilon}{p} \times p^2(n+m)$ opérations supplémentaires pour calculer les blocs de la séquence. Pour l'espace mémoire, il faut $2pn$ pour les blocs de vecteurs, $2\frac{\Upsilon}{p}p^2 = 2pr$ pour la séquence et $2p(p+d)$ pour les polynômes et les matrices intermédiaires. □

Ce dernier résultat est à comparer avec le coût de l'algorithme de Lanczos par blocs 6.4.4, page 127 : $8\Upsilon pn + \frac{1}{3}\Upsilon p^2 + 2\Upsilon(n+p)$ opérations supplémentaires et $3pn + 2p^2$ unités mémoire supplémentaires. Comme $d \leq n$, il y a un besoin d'au plus $3pn$ unités mémoires supplémentaires pour Wiedemann par blocs (la différence était de $3n$ dans le cas scalaire). Si le gain était de $4\Upsilon(n - \Upsilon)$ dans le cas scalaire, dans le cas par blocs de taille p, ce gain a été multiplié par p : $4\Upsilon p(n-\Upsilon)$, mais en outre, on a gagné des petits facteurs qui rendent l'algorithme de Wiedemann par blocs plus rapide même quand $n = \Upsilon$.

Nous avons donc obtenu que les algorithmes de Wiedemann nécessitent moins d'opérations arithmétiques que les algorithmes de Lanczos. Avec les méthodes suivantes, nous allons voir qu'il est en outre possible de diminuer la complexité théorique de l'algorithme de Wiedemann en utilisant des algorithmes rapides pour le polynôme générateur matriciel.

6.5.2 Approximants de Padé vectoriels

L'idée de G. Villard [170 - Villard (1997c)] consiste à transformer la séquence de matrices $p \times q$ en une séquence de vecteurs de dimension pq [135 - Penninga (1998)], puis à utiliser les algorithmes de B. Beckermann et G. Labahn pour trouver un polynôme générateur vectoriel. En fait, l'algorithme FPHPS de B. Beckermann et G. Labahn [9 - Beckermann et Labahn (1994), Théorème 3.4] peut s'appliquer à notre cas [169 - Villard (1997b)].

Nous avons alors exactement l'algorithme de Coppersmith 6.5.4, page 132 avec $q = 1$ (des vecteurs à la place des matrices).

L'important est que B. Beckermann et G. Labahn définissent ensuite un algorithme rapide (*superfast*) [9 - Beckermann et Labahn (1994), Théorème 6.2]. Ainsi, grâce à la multiplication rapide de polynômes par *Fast Fourier Transform*, le polynôme générateur sera calculé de manière *déterministe* avec un coût théorique en

$$\mathcal{O}\big((p+q)^2 p\frac{n}{q}\ln^2(p\frac{n}{q})\ln\ln(p\frac{n}{q})\big).$$

6.5.3 RÉSOLUTION TOEPLITZ

Une autre idée est de se ramener à la résolution d'un système linéaire de Toeplitz par blocs. En effet, il s'agit de trouver un polynôme matriciel $\Phi(X) = \sum_{l=0}^{d} \Phi_l X^l$ de degré borné vérifiant, pour une séquence S de matrices $p \times q$, $\forall i, S_i\Phi_0 + S_{i+1}\Phi_1 + \ldots + S_{i+d}\Phi_d = 0$. Ce problème s'exprime alors comme un système linéaire :

$$\begin{bmatrix} S_d & S_{d-1} & \ldots & S_1 & S_0 \\ S_{d+1} & S_d & \ddots & \vdots & S_1 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ S_{2d-1} & S_{2d-2} & \ldots & S_d & S_{d-1} \\ S_{2d} & S_{2d-1} & \ldots & S_{d+1} & S_d \end{bmatrix} \begin{bmatrix} \Phi_d \\ \Phi_{d-1} \\ \vdots \\ \Phi_1 \\ \Phi_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{6.4}$$

Ce type de système linéaire peut être résolu de manière rapide par l'algorithme de Bitmead et Anderson [15 - Bitmead et Anderson (1980)] ou encore l'algorithme de V. Pan et E. Kaltofen [89 - Kaltofen et Pan (1992), 90 - Kaltofen et Pan (1994)]. Une étude exhaustive du comportement de ces algorithmes se trouve dans [48 - Doreille (1995)], par exemple. L'application de ces méthodes à l'algorithme de Wiedemann est étudiée dans [83 - Kaltofen (1995), 170 - Villard (1997c)]. Le polynôme générateur sera donc calculé de manière *probabiliste* avec un coût similaire à celui de l'algorithme déterministe précédent,

$$\mathcal{O}\big((p+q)^2 n\ln^2(n)\ln\ln(n)\big).$$

6.6 PRÉCONDITIONNEMENTS

Nous voulons calculer le rang d'une matrice A sur un corps fini ou sur $\mathbb{Q}$. Les coefficients du polynôme caractéristique d'une matrice A, $\Pi_A(X)$, sont des combinaisons linéaires des mineurs de A, le rang de A est donc le degré de $\Pi_A(X)$, moins la plus grande puissance de X divisant $\Pi_A(X)$. Il est possible de calculer ce rang via l'algorithme de Wiedemann en préconditionnant A [91 - Kaltofen et Saunders (1991)]. En effet, si le polynôme caractéristique est égal au polynôme minimal multiplié par une puissance de X, le degré du polynôme minimal moins sa valuation (le degré du plus petit monôme non nul) révèle le rang. En pratique, on utilise le préconditionnement proposé par W. Eberly & E. Kaltofen [59 - Eberly et Kaltofen (1997), Théorème 6.2] qui permet de ne pas augmenter le coût du produit matrice-vecteur :

Théorème 6.6.1

Soit E un sous-ensemble fini, ne contenant pas zéro, d'un corps $\mathbb{F}$. Soit une matrice $A \in \mathbb{F}^{m\times n}$ de rang r. Soient deux matrices diagonales quelconques $\Delta \in E^{m\times m}$ et $D \in E^{n\times n}$. Alors degré$(\pi_{D\times A^T\times\Delta\times A\times D})-$ valuation$(\pi_{D\times A^T\times\Delta\times A\times D}) = r$, avec une probabilité au moins $1 - \frac{11.n^2+n}{2|E|}$.

En utilisant ce préconditionnement, on construit alors l'algorithme de Wiedemann générique suivant pour le calcul du rang :

Algorithme 6.6.2 *Rang-Wiedemann*

Entrées : – une matrice A de taille $m \times n$ à coeffi cients dans un corps $\mathbb{F}$.
– un sous-ensemble E de $\mathbb{F}$ fi ni et ne contenant pas zéro.

Sorties : – le rang r de A sur $\mathbb{F}$ avec une probabilité au moins $(1-\frac{11.n^2-n}{2|E|})$(Réussite de Wiedemann).

1 : Choisir $\Delta \in E^{m\times m}$ et $D \in E^{n\times n}$ diagonales quelconques.

[Initialisation de la Boîte noire avec multiplication explicite par les diagonales]

2 a : Calculer $B = \Delta * A * D$, la matrice telle que $B_{ij} = \Delta_{ii}A_{ij}D_{jj}$.

3 a : Appliquer *Wiedemann* sur B. φ est un facteur du polynôme minimal.

[Formation d'une Boîte noire, multiplication paresseuse]

2 b : $C = DA^T \Delta AD$

3 b : Appliquer *Wiedemann* symétrique sur C. φ est un facteur du polynôme minimal.

4 : $rang$ =degré(φ)−valuation(φ).

En théorie, si l'on veut calculer le rang dans $\mathbb{Z}/p\mathbb{Z}$, il suffit de plonger A dans une extension $\mathtt{GF}(p^k)$ suffisamment grande. En pratique, les calculs sont plus rapides si p^k ne dépasse ni la capacité mémoire, ni la taille du mot machine. Si l'on veut calculer le rang sur des grands corps premiers, une implémentation sans table nous permet donc de calculer efficacement sur des corps premiers de taille 2^{32} des matrices de dimensions 28000×28000 et de plus de 1.8 milliard de lignes si le corps premier est de taille proche de 2^{64}. Au contraire, si l'on veut calculer sur un petit corps premier, une extension est nécessaire et la seule arithmétique efficace nécessite alors des tables précalculées En l'occurrence une taille de 2^{24} nécessite déjà 100 Mo (voir section 4.2.2, page 67). Cela permet de calculer le rang de matrices de dimensions maximales 1750×1750. L'algorithme de Wiedemann peut toutefois être utilisé comme une heuristique pour le rang, la valeur retournée étant alors une borne inférieure du véritable rang. Il faudrait trouver une méthode pouvant certifier si une matrice est de rang donné. Une idée de G. Villard, par exemple, est d'orthogonaliser un vecteur z au hasard à partir d'une première orthogonalisation d'un sous-espace de Krylov $K(A, u)$. Si z est dans le noyau de $K(A, u)$, alors le rang est correct avec une probabilité d'au moins

$$1 - \frac{1}{q}.$$

D'autre part, de manière empirique, nous avons constaté que les problèmes de mauvais conditionnement intervenaient surtout quand la matrice contenait des lignes ou des colonnes avec un seul élément. En combinant l'algorithme de Wiedemann avec la méthode de Odlyzko (voir 5.1, page 81) qui élimine ces lignes et colonnes, nous avons obtenu un algorithme ne renvoyant de rang incorrect sur une extension de taille 2^{32} qu'extrêmement rarement.

6.7 QUELLE MÉTHODE DE KRYLOV ?

Nous commençons par un tableau récapitulatif des seconds ordres des différentes complexités arithmétiques des méthodes de Krylov. Nous considérons une matrice symétrique $n \times n$ et des blocs de taille $p \times q$. Nous ne mentionnons pas

le nombre de produits matrice-vecteur puisqu'il est identique pour tous ces algorithmes, et considérons de même que le nombre d'itérations $\Upsilon \leq rang \leq n$ est identique.

Algorithme	Opérations	Mémoire
Lanczos	$\Upsilon(8n)$	$2n$
Wiedemann	$\Upsilon(4\Upsilon + 4n)$	$2n + 3\Upsilon$
Blocs Lanczos	$\Upsilon(8pn + \frac{7}{3}p^2 + 2(n+p))$	$3pn + 3p^2$
Coppersmith	$\Upsilon(4p\Upsilon + 4pn)$	$4pn + 2p^2 + 2\Upsilon p$

TABLEAU 6.1 – Seconds ordres des complexités des méthodes de Krylov

Pour la résolution formelle de systèmes linéaires, l'algorithme de Wiedemann s'intéresse seulement aux coefficients de la récurrence, c'est-à-dire au polynôme minimal, mais doit ensuite reconstruire un vecteur solution. L'algorithme de Lanczos, d'autre part, ne conserve pas les coefficients mais peut construire un vecteur solution pendant l'itération. Il s'en suit que, pour la résolution de système, l'algorithme de Lanczos est plus performant que l'algorithme de Wiedemann [103 - Lambert (1996), 59 - Eberly et Kaltofen (1997)].

Au contraire, pour le calcul du rang d'une matrice, où un vecteur solution est inutile, nous avons vu que l'algorithme de Wiedemann est alors plus performant même si l'algorithme de Lanczos nécessite un peu moins de mémoire (la différence est de $3n$ unités de mémoire). Cependant, la différence ici n'est pas significative et, en pratique, on préfèrera l'algorithme de Wiedemann lorsque la vérification n'est pas nécessaire, par exemple si l'on calcule dans un corps contenant beaucoup d'éléments. En effet, l'algorithme de Wiedemann nécessite moins d'opérations arithmétiques. Si l'on considère que le nombre d'itérations Υ de l'algorithme de Lanczos est la dimension du sous-espace de Krylov $K(A, u_0)$ alors ce nombre est aussi le degré du polynôme minimal $\pi_{u_0,A}$ et la différence d'opérations est donc de $4\Upsilon(n - \Upsilon)$ opérations si l'on utilise une terminaison anticipée. D'autre part, dans un petit corps fini, il est possible d'augmenter le seuil de terminaison. En effet, en pratique, pour toutes les matrices que nous avons traitées, et pour tout corps, un seuil de terminaison proche de 20 est suffisant. Cela reste totalement empirique pour l'instant même si une série de plus de 20 polynômes isotropes consécutifs est assez improbable (dans $\mathbb{Z}/2\mathbb{Z}$, cela risque d'arriver environ 1 fois sur un million). Un dernier avantage de l'algorithme de Wiedemann est qu'il est possible de calculer plusieurs de ces polynômes, sans vérification, puis de combiner les résultats obtenus et de ne faire qu'une seule vérification finale. En outre, il est possible de conserver les vecteurs intermédiaires en mémoire jusqu'à un certain seuil. Quand le degré du polynôme est faible, cela permet d'éviter

de les recalculer pour la vérification. Celle-ci n'est alors plus très coûteuse. Cette méthode est utilisée pour le calcul sur les entiers, par exemple, section 9.5, page 174.

Enfin, les algorithmes par blocs nécessitent plus d'opérations que les algorithmes scalaires. Toutefois, utilisés sur un corps de caractéristique 2, et en prenant p et q égaux à 32 ou 64, suivant la taille du mot machine, une arithmétique spécialisée peut être mise en œuvre. Le coût de l'algorithme par blocs est alors divisé par 32 ou 64. Le nombre d'opérations additionnelles devient équivalent à celui des algorithmes scalaires et d'un autre côté, le coût du produit matrice-vecteur est lui aussi divisé par 32 ou 64, rendant l'algorithme par blocs plus intéressant. Le deuxième intérêt des algorithmes par blocs est une plus grande indépendance des produits matrice-vecteur, induisant une parallélisation plus aisée. Nous étudions dans la prochaine section le comportement des parallélisations de l'algorithme de Wiedemann.

6.8 PARALLÉLISATIONS

Dans cette section, nous nous intéressons à la parallélisation des méthodes itératives. Une première approche consiste à paralléliser la meilleure variante, l'algorithme de Wiedemann. Nous présentons ensuite la parallélisation de l'algorithme par blocs qui permet de réduire les communications et d'augmenter ainsi le parallélisme malgré un nombre d'opérations plus important.

6.8.1 PARALLÉLISATION DE L'ALGORITHME DE WIEDEMANN

L'algorithme de Wiedemann est une boucle comportant trois phases :
- produit matrice-vecteur $v_{i+1} = Av_i$, ($v_0 = v$),
- produit scalaire $s_{i+1} = u^t v_{i+1}$,
- mise à jour du polynôme générateur avec les s_1, …, s_{i+1}.

Par passage dans la boucle, à l'étape i, le calcul du produit scalaire nécessite $2n$ opérations arithmétiques ; la mise à jour du polynôme nécessite $2i$ opérations arithmétiques ; si la matrice A est creuse avec Ω éléments non nuls, le produit matrice-vecteur nécessite Ω opérations*.

*En prenant $u = v$, le nombre total de produits est réduit de moitié si A est symétrique.

Nous avons choisi de découper A en blocs bi-dimensionnels pour paralléliser le produit matrice-vecteur et les produits scalaires [31 - Carton et Giquel (1999), Théorème 5.5]. Le stockage utilisé pour les blocs creux est le ligne-compressé standard [98 - Kumar et al. (1994), 150 - Saad (1996b)] (*i.e.* un tableau de valeurs, un tableau avec les indices de colonne correspondants et un troisième tableau de pointeurs sur le premier élément de chaque ligne, voir 11.2, page 202). Ce format est particulièrement adapté au produit matrice-vecteur. En outre, comme nous avons appliqué cet algorithme sur des matrices assez grandes (jusqu'à 500000 lignes) et très creuses, nous avons aussi utilisé des vecteurs creux intermédiaires. À l'initialisation, les blocs sont chargés de manière cyclique bi-dimensionnelle sur les processeurs. Les produits sont ensuite gérés par un algorithme de liste qui peut prendre la décision de migrer les données.

Nous avons découplé le calcul des éléments de la séquence de leur utilisation pour la mise à jour du polynôme générateur. En effet, c'est cette mise à jour qui conditionne le nombre total de produits matrice-vecteur à effectuer. Celui-ci n'est pas connu au départ et Athapascan-1 ne permet pas une terminaison anticipée de ces tâches. Il faut donc lancer de manière asynchrone un groupement de produits, Athapascan-1 se chargeant des dépendances de données. Si la mise à jour nécessite d'autres produits scalaires, un nouveau groupe de produits est relancé. Cela induit quelques synchronisations non désirées mais permet de réduire le nombre total de produits effectués [52 - Dumas (2000)].

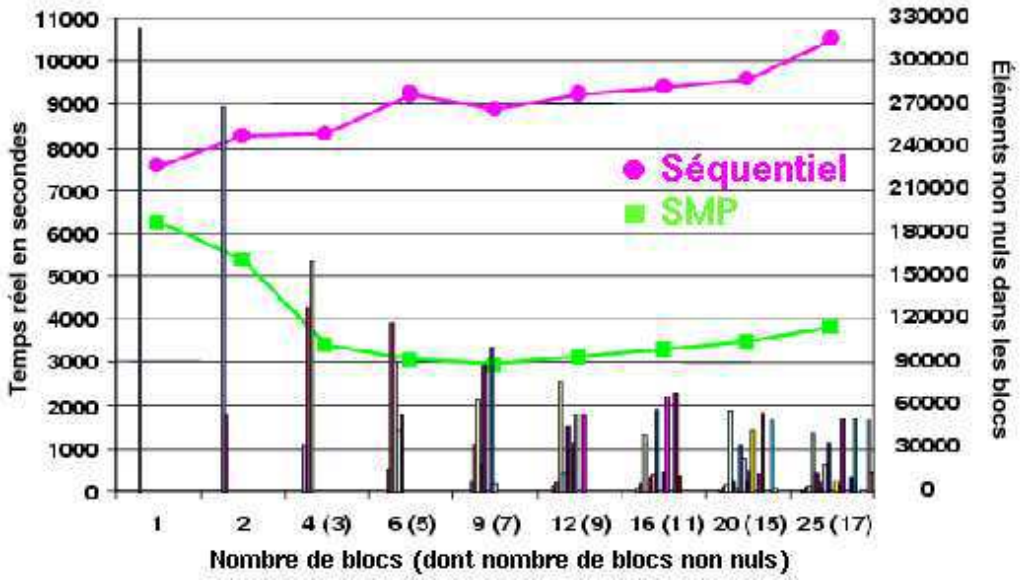

FIGURE 6.1 – Calcul du polynôme minimal de degré 39535 d'une matrice 62370×51975

Dans la figure 6.1, page 141, nous étudions les effets du découpage, suivant le nombre de blocs et la taille du groupement de produits, sur une matrice représentant les relations entre la dimension 3 et la dimension 4 du complexe simpliciel associé aux couplages du graphe complet à 12 sommets (cf. [16 - Björner et al. (1994)]). Cette matrice est de taille 62370×51975 avec 311850 éléments non nuls et intervient dans le calcul du groupe d'homologie du complexe simpliciel. Nous calculons le polynôme minimal de degré 39535 d'une matrice carrée symétrique préconditionnée associée $D_1 A^t D_2 A D_1$; il y a donc au moins 79070 produits par A à effectuer. Les calculs se font modulo 32749.

Cette figure présente les temps d'exécution, en secondes, sur un quadri-processeurs Sun Ultra-II 4×250 MHz. Pour cette matrice, on observe une accélération absolue - meilleur temps séquentiel sur meilleur temps parallèle - de 2,55 pour 4 processeurs. Le découpage se fait en blocs carrés, mais l'irrégularité de la matrice donne des blocs déséquilibrés. En effet, les meilleurs résultats sont obtenus pour un découpage en 9 blocs. Dans ce cas, nous constatons, sur la figure, que seuls 7 blocs sont non nuls et que le nombre d'éléments par bloc varie de 5772 à 98718.

Il est à noter que le meilleur temps séquentiel est obtenu sans découpage de la matrice. L'accélération reflète donc le surcoût dû au parallélisme ainsi que le surcoût de structure (découpage en blocs, vecteurs creux ...) ; en fait, l'accélération comparée - meilleur temps sur 1 processeur avec la version parallèle sur meilleur temps parallèle - serait légèrement supérieure à 3. En outre, l'accélération est déjà intéressante même sans découpage, ce qui illustre l'utilité du découplage des trois phases de la boucle. Le tableau 6.2 montre les accélérations obtenues pour quelques autres matrices d'homologie (ϖ est le nombre d'éléments non nuls par ligne).

Matrice	ϖ, $n \times m$	Séquentiel	4 processeurs	Accélération
bibd_22_8	38760, 231x319770	995.41	343.80	2.90
ch7-6.b4	5, 15120x12600	412.42	240.24	1.72
ch7-7.b5	6, 35280x52920	4141.32	1865.12	2.22
mk12.b4	5, 62370x51975	7539.21	2959.42	2.55
ch8-8.b4	5, 376320x117600	33 heures	10 heures	3.37
ch7-9.b5	6, 423360x317520	105 heures	34 heures	3.10
ch8-8.b5	6, 564480x376320	-	55 heures	-

TABLEAU 6.2 – Accélérations sur quatre processeurs, modulo 32749

Enfin, nous montrons ici des accélérations sur un quadri-processeurs ; malheureusement, sur une architecture distribuée, les performances sont moins bonnes. En effet, sur p processeurs, et pour chaque produit matrice-vecteur par blocs, il faut communiquer un total de $2n\sqrt{p}$ valeurs. Pour les matrices très creuses, ce

volume de communications est donc du même ordre que le volume de calcul, induisant un recouvrement insuffisant. Il est alors nécessaire de paralléliser notre application différemment. La prochaine section étudie la parallélisation des algorithmes utilisant des polynômes générateurs matriciels.

6.8.2 PARALLÉLISATION DES ALGORITHMES PAR BLOCS

De même que l'algorithme de Wiedemann, l'algorithme de Coppersmith est une boucle comportant trois phases :

- produits matrice-vecteur $V_{i+1} = AV_i$, ($V_0 = V$),
- produits scalaires $S_{i+1} = U^t V_{i+1}$,
- mise à jour du polynôme générateur matriciel avec les S_1, …, S_{i+1}.

Cependant, dans ce cas, les produits matrice-vecteur et les produits scalaires sont effectués sur des blocs de vecteurs. Il devient alors possible de paralléliser ces deux opérations directement, par exemple en distribuant les vecteurs V_i sur différents processeurs et en dupliquant le vecteur U pour effectuer les produits scalaires en place. Il n'y a alors plus que la communication des petites matrices S_i à effectuer.

Considérons une matrice creuse A avec Ω éléments non nuls. Prenons la symétrique de taille $n \times n$ pour simplifier. Dans la suite, nous supposons que Ω est plus grand que n ; en effet, dans le cas contraire, il vaut mieux utiliser l'algorithme de renumérotation. Les corollaires 6.3.4, page 123 et 6.5.6, page 134 nous donnent le nombre d'opérations nécessaires pour calculer son polynôme minimal de degré d : $4d^2 + 4dn + 2d\Omega$ avec l'algorithme de Wiedemann et $4qd^2 + 4qdn + 2d\Omega$ pour l'algorithme par blocs avec q vecteurs initiaux. Le nombre d'opérations nécessaires croît donc avec le nombre de vecteurs initiaux, alors que le nombre de produits matrice-vecteur reste constant. L'intérêt de cette parallélisation est donc de faire en parallèle ces produits puisqu'ils osnt indépendants. Le temps de calcul des produits peut donc être divisé par le nombre de processeurs, le calcul du polynôme générateur restant séquentiel. Nous avons vu dans la section précédente que le facteur prépondérant pour des matrices creuses avec $\Omega = \mathcal{O}(n)$ est le produit matrice-vecteur. Pour ces matrices, avec l'algorithme parallèle par blocs, le facteur prépondérant devient le calcul du polynôme générateur matriciel. Il reste à mener des expériences pour déterminer précisément les conditions intéressantes d'utilisation de ce dernier algorithme. Néanmoins, nous pouvons d'ores et déjà affirmer que l'algorithme parallèle par blocs ne sera pas très efficace si Ω est un petit multiple de n, mais qu'il sera sans doute avantageux si Ω est entre $\mathcal{O}(n \ln(n))$ et n^2.

7

DAVID ET GOLIATH : CALCUL DU RANG DE MATRICES CREUSES

« Une solution qui vous démolit vaut mieux que n'importe quelle incertitude »

Boris Vian

Sommaire

Ce chapitre est consacré à la comparaisons des algorithmes pour le calcul du rang de matrices creuses. Dans la première section, nous décrivons plus en détails les différentes matrices que nous avons utilisées pour réaliser cette étude et nous nous proposons, dans la section suivante, de déterminer les domaines privilégiés de résolution des méthodes directes et itératives. Nous avons utilisé comme arithmétique commune l'arithmétique sur un corps fini (*GFq* décrite au chapitre 4).

7.1 BESTIAIRE

7.1.1 ALÉATOIRES

Cette collection est un ensemble de matrices carrées creuses ayant un petit nombre d'éléments ($\mathcal{O}(n)$) non nuls modulo 2^{16}, placés aléatoirement. Nous les avons déjà utilisées section 5.1, page 81 pour étudier les effets de la renumérotation.

7.1.2 HOMOLOGIE

Ces matrices sont issues de la topologie algébrique. Elles nous ont été données par Volkmar Welker du Fachbereich Mathematik und Informatik à la Philipps-Universität de Marburg en Allemagne. Un simplexe est la généralisation du tétraèdre régulier en dimension n (un segment en dimension 1, un triangle en dimension 2, un tétraèdre en dimension 3, ...). Un complexe simpliciel de dimension n est une triangulation d'un espace, c'est-à-dire une collection de simplexes de dimensions n pour lesquels deux simplexes sont d'intersection vide ou n'ont en commun qu'une face de chacun d'eux (un simplexe de dimension $n-1$). Une matrice des limites (*boundary map*) d'un complexe simpliciel est une matrice contenant les relations entre faces et arêtes du complexe pour une dimension donnée (par exemple, en dimension 3, les faces sont les tétraèdres et les arêtes sont les triangles). Le calcul du rang et des invariants de Smith de ces matrices intervient dans le calcul des groupes d'homologie de ces complexes [119 - Munkres (1994)]. Nous utilisons trois classes de ces matrices issues de complexes simpliciaux formés à partir de graphes. Ces matrices sont identifiées de la manière suivante :

- **mk**i**.b**j est la matrice des limites en dimension j du complexe formé à partir des couplages du graphe complet à i sommets.

- **ch**i-k**.b**j est la matrice des limites en dimension j du complexe formé par les différentes positions de tours sur un échiquier de taille $i \times k$. Ces tours doivent être dans l'impossibilité de se prendre.
- **n**i**c**k**.b**j est la matrice des limites en dimension j du complexe formé à partir des graphes non-i-connectés à k sommets (un graphe i-connecté est un graphe dans lequel, pour toute paire de sommets non adjacents, il existe au moins i chemins de l'un à l'autre dont les ensembles de sommets sont deux à deux disjoints).

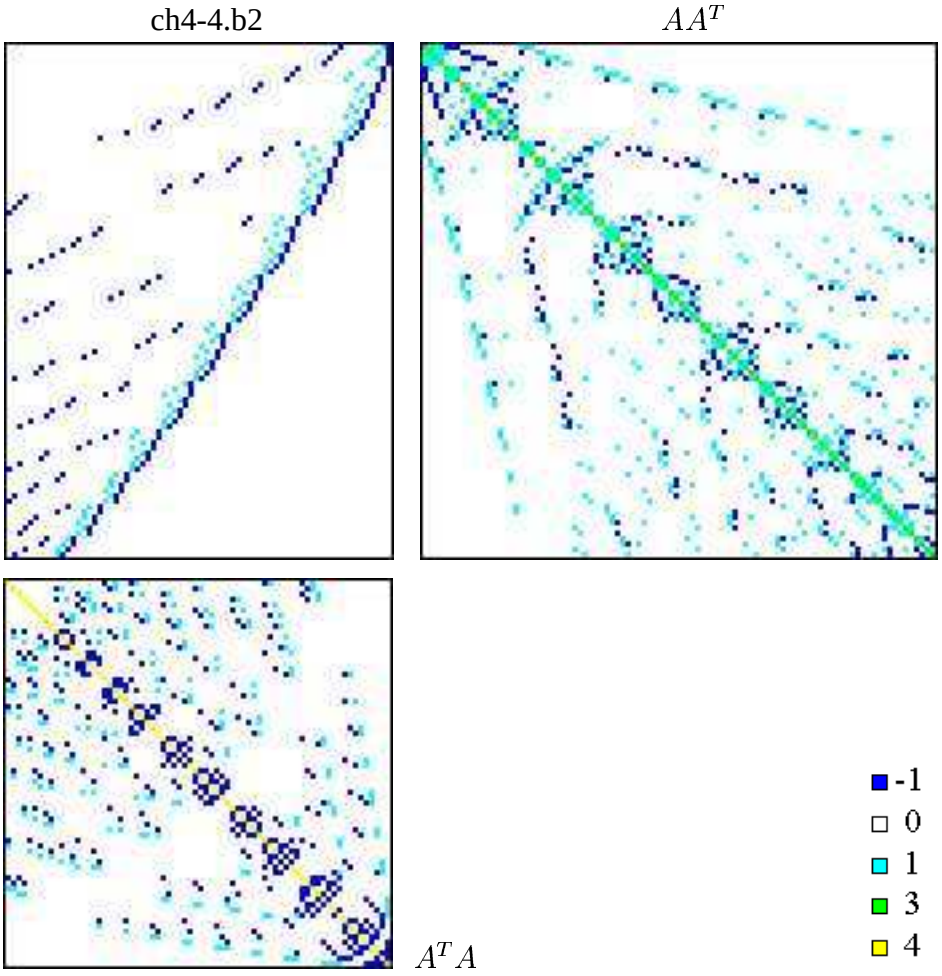

FIGURE 7.1 – Échiquier 4-4, matrices des limites en dimension 2

Pour tous ces complexes, les matrices des limites sont creuses avec un nombre fixé d'éléments non nuls par ligne (k) et par colonne (l). Si A_i est la matrice des limites en dimension i d'un complexe, alors $k = i + 1$. Une ligne de la matrice correspond à une face (simplexe de dimension i), une colonne correspond

à une arête (simplexe de dimension $i-1$). Les éléments signifient simplement que telle face contient telle arête dans tel sens ; les éléments de la matrice sont donc seulement -1, 0 ou 1. En outre, un bon ordonnancement des faces rend ces matrices quasiment triangulaires. De même, les seuls éléments, hors diagonale, des **Laplaciens**, $A_i A_i^t$ et $A_i^t A_i$, valent aussi -1, 0 et 1. La diagonale de ces Laplaciens est formée, respectivement, seulement de k ou seulement de l. D'autre part, ces Laplaciens ont plus que deux fois le nombre d'éléments non nuls de A_i. Pour calculer des produits matrice-vecteur avec eux, nous n'avons donc pas effectué la multiplication des matrices, nous avons plutôt effectué deux produits matrice-vecteur : $(A_i A_i^t)v = A_i(A_i^t v)$. Pour plus de détails sur ces complexes, voir [6 - Babson et al. (1999), 17 - Björner et Welker (1999)] pour les complexes de graphes non-i-connectés, [16 - Björner et al. (1994), 142 - Reiner et Roberts (2000)] pour les complexes de couplages et d'échiquier et, enfin, [64 - Friedman (1996)] pour l'utilisation des Laplaciens.

La figure 7.1, page 147 montre une de ces matrices, ch4-4.b2, la matrice des limites en dimension 2 pour l'échiquier 4×4. Elle est de dimension 96×72 avec 288 éléments non nuls, 3 par ligne et 4 par colonne. Les Laplaciens AA^T et $A^T A$ sont aussi présents, ils sont de dimensions respectives 96×96 avec 960 éléments non nuls et 72×72 avec 648 éléments non nuls.

7.1.3 BASES DE GRÖBNER

Ces matrices sont issues du calcul de bases de Gröbner. Elles nous ont été données par Jean-Charles Faugère du Laboratoire d'Informatique de Paris 6 (LIP6), à l'université Pierre et Marie Curie. On peut associer à un ensemble de polynômes à plusieurs variables une matrice dans laquelle les lignes correspondent à un polynôme et les colonnes correspondent à un monôme. Les éléments de la matrice sont les coefficients des monômes pour chacun des polynômes. La méthode de Faugère de calcul de bases de Gröbner d'un système d'équations polynomiales fait intervenir le calcul du rang et d'une forme triangulaire de telles matrices sur des sous-ensembles de polynômes intermédiaires [62 - Faugère (1999)]. De même que les matrices d'homologie, ces matrices sont quasiment triangulaires avec un bon ordonnancement*. Par exemple, la figure 5.8, page 107 montre la matrice

*Ces matrices sont aussi appelées *girafes* du fait de la forme évoquée par leurs entrées, voir figures 5.8, page 107 et 7.2, page 149

numéro 5 issue du système initial suivant [62 - Faugère (1999)] :

$$\begin{aligned}
-x_1 + 2x_8^2 + 2x_7^2 + 2x_6^2 + 2x_5^2 + 2x_4^2 + 2x_3^2 + 2x_2^2 + x_1^2 &= 0 \\
-x_2 + 2x_8x_7 + 2x_7x_6 + 2x_6x_5 + 2x_5x_4 + 2x_4x_3 + 2x_3x_2 + 2x_2x_1 &= 0 \\
-x_3 + 2x_8x_6 + 2x_7x_5 + 2x_6x_4 + 2x_5x_3 + 2x_4x_2 + 2x_3x_1 + x_2^2 &= 0 \\
-x_4 + 2x_8x_5 + 2x_7x_4 + 2x_6x_3 + 2x_5x_2 + 2x_4x_1 + 2x_3x_2 &= 0 \\
-x_5 + 2x_8x_4 + 2x_7x_3 + 2x_6x_2 + 2x_5x_1 + 2x_4x_2 + x_3^2 &= 0 \\
-x_6 + 2x_8x_3 + 2x_7x_2 + 2x_6x_1 + 2x_5x_2 + 2x_4x_3 &= 0 \\
-x_7 + 2x_8x_2 + 2x_7x_1 + 2x_6x_2 + 2x_5x_3 + x_4^2 &= 0 \\
-1 + 2x_8 + 2x_7 + 2x_6 + 2x_5 + 2x_4 + 2x_3 + 2x_2 + x_1 &= 0
\end{aligned}$$

Un autre exemple est la matrice de la figure 7.2, page 149 issue d'un problème de robotique [61 - Faugère (1994), Section 2.3].

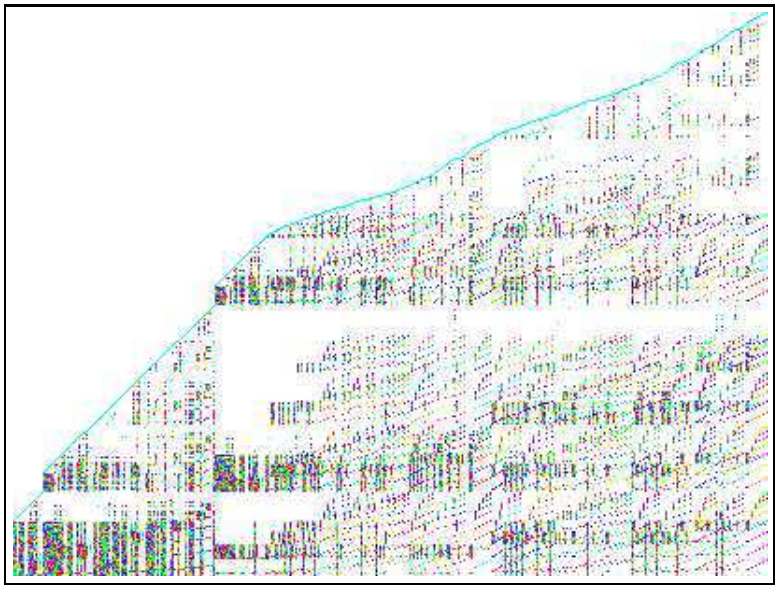

FIGURE 7.2 – Matrice robot24c1_mat5

7.1.4 BALANCED INCOMPLETE BLOCK DESIGN

Ces matrices nous ont été données par Mark Giesbrecht du Department of Computer Science à l'University of Western Ontario, Canada. Un *Balanced Incomplete Block Design* ou $BIBD(v, b, r, k, \lambda)$ est une manière de sélectionner b sous-ensembles, appelés blocs, d'un ensemble de v objets distincts de sorte que

chaque bloc contienne exactement k objets distincts, chaque objet apparaissant dans exactement r blocs et chaque paire de deux objets distincts apparaissant dans exactement λ blocs. En comptant les objets, on s'aperçoit que $r(k-1) = \lambda(v-1)$ et que $bk = vr$; une configuration pourra donc être désignée par seulement trois de ses paramètres : $BIBD(v, k, \lambda)$ [172 - Wallis et al. (1972)]. La question est de savoir s'il est possible de construire telle ou telle configuration. Pour répondre à cette question, on construit une matrice $A \in \mathbb{Z}^{m \times n}$ avec $m = C_v^2$, le nombre de paires possibles, $n = C_v^k$, le nombre de sous-ensembles de taille k. Ensuite, $a_{ij} = 1$ si la paire associée à la ligne i est contenue dans le sous-ensemble j, et $a_{ij} = 0$ sinon. La figure 7.3, page 150 montre la matrice 55×462 pour $v = 11$ et $k = 5$.

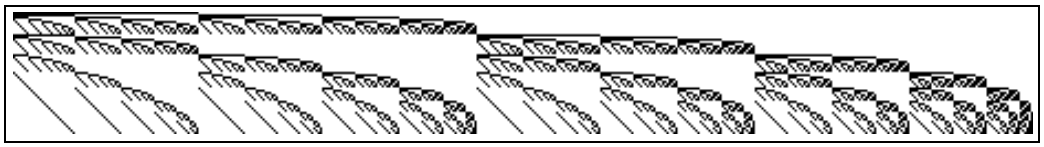

FIGURE 7.3 – Matrice BIBD pour 11 objets dans des blocs de taille 5

Ensuite il faut trouver b colonnes dans cette matrice telles que le nombre de fois qu'une paire apparaît dans ces colonnes est λ. C'est-à-dire, il faut trouver $X \in \mathbb{Z}_+^n$, avec seulement b éléments non nuls, tel que $AX = (\lambda, \ldots, \lambda)^T$. Une réponse partielle peut être obtenue en calculant simplement le rang de A et le rang de $[A, (\lambda, \ldots, \lambda)^T]$; une réponse plus complète peut nécessiter le calcul de formes normales et de matrices de passage. Les matrices sont assez allongées et contiennent seulement des 0 et des 1.

Par exemple, savoir si la configuration $BIBD(22, 33, 12, 8, 4)$ existe, est un problème ouvert. Nous avons pu montrer facilement, par des calculs de rang, que le système admet des solutions entières, nous ne savons pas s'il existe des solutions avec seulement b éléments non nuls et positifs.

7.2 QUEL ALGORITHME POUR LE RANG ?

Nous voulons déterminer les domaines privilégiés de calcul du rang sur matrices creuses pour l'algorithme de Wiedemann avec terminaison anticipée 6.6.2, page 137 et l'algorithme de Gauß avec renumérotation 5.1.3, page 84. Ce sont les deux algorithmes séquentiels les plus rapides. Nous comparons ces méthodes pour le calcul dans un corps fini de taille proche du mot machine avec l'implémentation

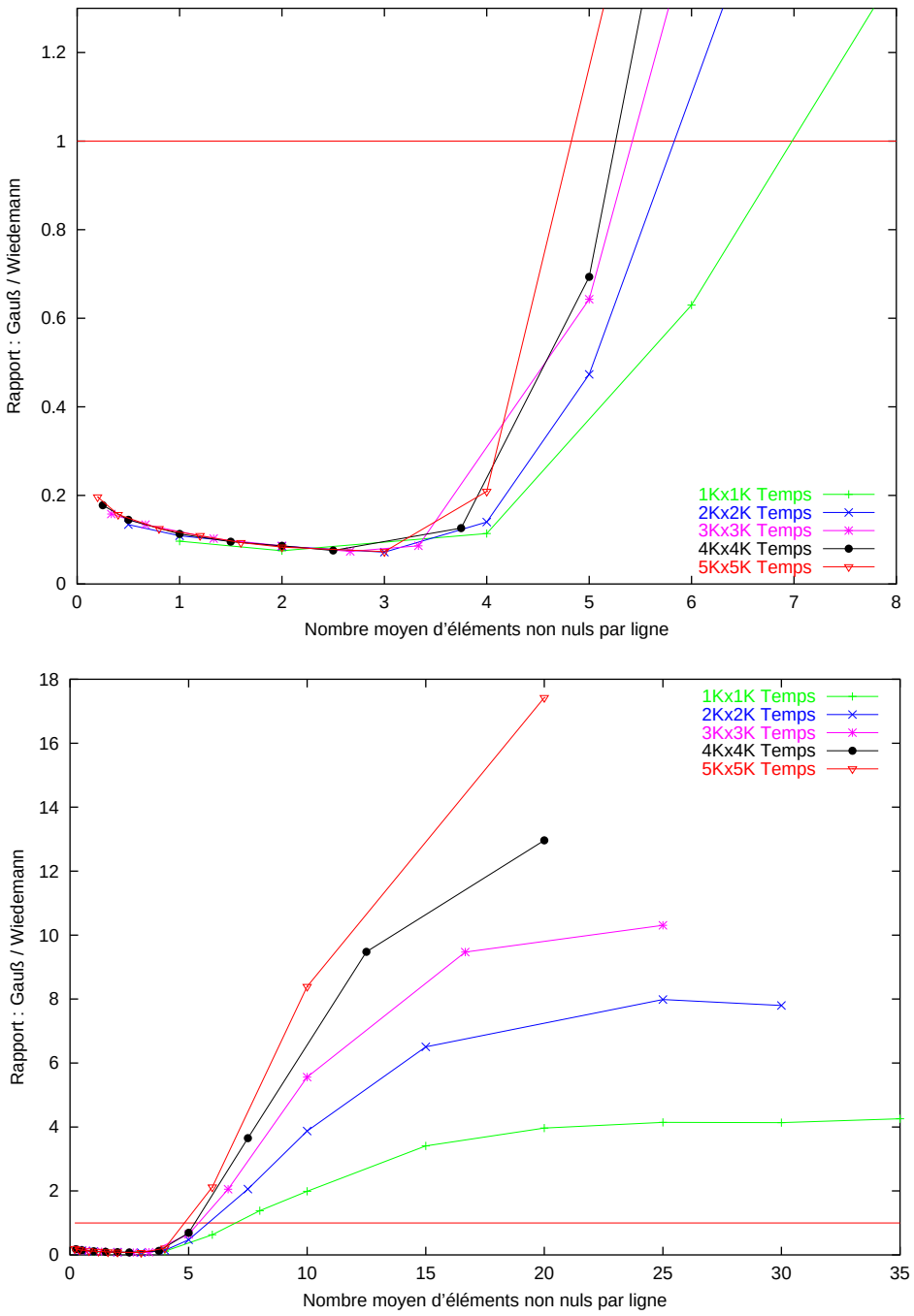

FIGURE 7.4 – Temps comparés, modulo 65521, des méthodes directe et itérative pour des matrices aléatoires. Chaque courbe correspond à une taille de matrices.

GFq de la section 4.1.2, page 61. Les temps ont été obtenus sur un ultrasparc II à 250 MHz.

Sur la figure 7.4, page 151, nous comparons les deux algorithmes sur les matrices aléatoires. Pour les matrices très creuses, l'élimination est bien plus performante car il y a très peu d'opérations dans ce cas. D'autre part, dès que le nombre d'éléments moyen par ligne dépasse 5, le remplissage n'est plus négligeable et l'algorithme de Wiedemann devient plus intéressant. En outre, ce comportement est valide jusqu'à ce que les matrices soient quasiment denses (l'étude des complexités arithmétiques, théorèmes 5.1.2, page 81, 6.3.3, page 122 et 6.3.4, page 123, pour une matrice $m \times n$ avec Ω éléments non nuls, montre que l'élimination redevient plus intéressante quand $\Omega > \frac{mn}{4}$ pour une matrice quelconque et quand $\Omega > \frac{n^2}{2}$ pour une matrice symétrique).

Au contraire, les matrices d'homologie, de Gröbner ou BIBD sont déjà quasiment triangulaires. Nous voyons sur les tableaux 7.1, 7.2 et 7.3 que l'élimination de Gauß est alors très performante puisque très peu d'opérations sont à effectuer. En outre, pour les matrices BIBD, par exemple, un grand déséquilibre entre le nombre de lignes et le nombre de colonnes est à l'avantage de l'élimination puisque dans ce cas, le remplissage n'a pas le temps de se faire. Toutefois,

Matrice	Ω, $n \times m$, r	Gauß	Wiedemann
robot24_m5	15118, 404x302, 262	0.52	1.52
rkat7_m5	38114, 694x738, 611	1.84	8.72
f855_m9	171214, 2456x2511, 2331	10.54	176.61
cyclic8_m11	2462970, 4562x5761, 3903	671.33	4138.77

TABLEAU 7.1 – Temps comparés, modulo 65521, pour des matrices de Gröbner

pour de très grandes matrices, l'algorithme de Wiedemann est à même de compléter l'exécution alors que l'élimination provoque un dépassement de capacité mémoire. Quand la mémoire disponible est le facteur limitant, même le plus léger remplissage peut ainsi anéantir l'élimination. En outre, nous voyons, sur la matrice ch7-7.b5 par exemple, qu'à la limite des processeurs, l'algorithme de Wiedemann commence à être compétitif puisque moins consommateur de mémoire (théorèmes 5.1.2, page 81 et 6.3.3, page 122).

En conclusion, nous avons testé deux méthodes pour calculer le rang de grandes matrices creuses. Nous avons validé une heuristique efficace pour la réduction du remplissage et constaté que, pour les matrices fortement clairsemées, l'élimination de Gauß avec renumérotation reste étonnamment efficace. D'autre part, nous avons montré que l'algorithme de Wiedemann peut être efficace en pratique.

Matrice	Ω, $n \times m$, r	Gauß	Wiedemann
bibd_12_5	7920, 66x792, 66	0.05	0.24
bibd_17_4	14280, 136x2380, 136	0.13	0.98
bibd_13_6	25740, 78x1716, 78	0.18	0.88
bibd_49_3	55272, 1176x18424, 1176	1.24	35.05
bibd_15_7	135135, 105x6435, 105	2.27	6.92
bibd_81_3	255960, 3240x85320	10.53	528.77
bibd_16_8	360360, 120x12870, 120	6.18	21.63
bibd_17_8	680680, 136x24310, 136	12.66	45.30
bibd_18_9	1750320, 153x48620, 153	33.59	132.20
bibd_19_9	3325608, 171x92378, 171	72.43	289.22
bibd_20_10	8314020, 190x184756, 190	180.97	834.59
bibd_22_8	8953560, 231x319770, 231	173.33	995.41

TABLEAU 7.2 – Temps comparés, modulo 65521, pour des matrices BIBD

Matrice	Ω, $n \times m$, r	Gauß	Wiedemann
n4c5.b6	33145, 4735x4340, 2474	2.73	47.17
n2c6.b7	31920, 3990x5715, 2772	3.64	49.46
n2c6.b6	40005, 5715x4945, 2943	6.44	64.66
n4c6.b13	88200, 6300x25605, 5440	8.92	288.57
n4c6.b12	1721226, 25605x69235, 20165	231.34	4131.06
mk9.b3	3780, 945x1260, 875	0.26	2.11
mk13.b5	810810, 135135x270270, 134211	Dépassement	23 heures
ch7-7.b6	35280, 5040x35280, 5040	4.67	119.53
ch7-6.b4	75600, 15120x12600, 8989	49.32	412.42
ch7-7.b5	211680, 35280x52920, 29448	2179.62	4141.32
ch8-8.b4	1881600, 376320x117600, 100289	19 heures	33 heures
En parallèle avec 4 processeurs			
ch7-9.b5	2540160, 423360x317520, 227870	Dépassement	34 heures
ch8-8.b5	3386880, 564480x376320, 279237	Dépassement	55 heures

TABLEAU 7.3 – Temps comparés, modulo 65521, pour des matrices d'Homologie

Il a un bon comportement général, même pour de petites matrices, et est la seule solution quand l'espace mémoire fait défaut. Enfin, avec ces deux algorithmes, nous sommes capables de traiter les plus grands systèmes sur des corps finis. À titre de comparaison, A. Lobo [109 - Lobo (1995), Table 2.1] résout un système 20000×20000 de 1.3 millions d'éléments non nuls sur $\mathbb{Z}/_{32749}\mathbb{Z}$ en 28 heures sur un quadri-processeurs 4×100 MHz avec l'algorithme de Wiedemann par blocs. Nous obtenons des temps de calcul comparables, ou meilleurs. Pour la matrice ch8-8.b4, par exemple, qui a 1.8 millions d'éléments, le temps de calcul du rang par l'algorithme de Wiedemann scalaire est de 33 heures en séquentiel et de 10 heures sur un quadri-processeurs 4×250 MHz.

Troisième partie

Forme normale de Smith entière

Dans cette partie, nous nous intéressons au calcul de la forme normale de Smith entière sur matrices creuses. Comme dans la définition 5.3.2, page 93, la forme de Smith d'une matrice est diagonale et ses éléments diagonaux sont les facteurs invariants. Le théorème de Smith précise que toute matrice entière est équivalente à une matrice diagonale [156 - Smith (1861), 66 - Gantmacher (1959)] :

Théorème (Forme normale de Smith)

Soit un anneau principal $\mathcal{A}$. Pour toute matrice $A \in \mathcal{A}^{m\times n}$, il existe deux matrices unimodulaires U et V telles que A est équivalente à une matrice diagonale $S \in \mathcal{A}^{m\times n}$. $A = USV = USmith(A)V = Udiag(s_1, \ldots, s_r, 0, \ldots, 0)V$, où r est le rang de A, $\forall i$, $s_i \neq 0$ et $\forall 1 \leq i < r$, $s_i | s_{i+1}$.

Nous nous intéressons plus particulièrement à la forme de Smith de matrices entières. Par exemple, si l'on considère la matrice suivante,

$$A = \begin{bmatrix} 0 & 3 & -3 & 0 & 0 \\ -1 & 0 & -2 & 2 & 1 \\ 3 & 6 & 0 & -6 & 3 \\ 0 & 6 & -6 & 0 & 6 \end{bmatrix}$$

ses mineurs d'ordre 1 sont $\{ -6, -3, -2, -1, 0, 1, 2, 3, 6 \}$ et donc $s_1(A) = d_1(A) = 1$. Ses mineurs d'ordre 2 sont $\{ -9, 9, -12, 12, -18, 18, -36, 36, 0, -3, 3, -6, 6 \}$ donc $s_2(A) = \frac{d_2(A)}{d_1(A)} = 3$. Ses mineurs d'ordre 3 sont $\{ -108, 108, -18, 18, -36, 36, -54, 54, 0 \}$ et donc $s_2(A) = \frac{d_3(A)}{d_2(A)} = \frac{18}{3} = 6$. Enfin, tous ses mineurs d'ordre 4 sont nuls puisque A est de rang 3. Ce qui nous donne la forme de Smith de A :

$$\begin{bmatrix} -2 & 2 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ -4 & 3 & 1 & 0 \\ 0 & -3 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 3 & -3 & 0 & 0 \\ -1 & 0 & -2 & 2 & 1 \\ 3 & 6 & 0 & -6 & 3 \\ 0 & 6 & -6 & 0 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & -5 & -5 & -2 & 2 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Après un bref aperçu des applications et des algorithmes de calcul de la forme normale de Smith, nous proposons un nouvel algorithme, s'appuyant sur le calcul du rang modulo des petites puissances de nombres premiers, et différentes

variantes d'implémentation. Cet algorithme est spécialement adapté au cas des matrices creuses. En particulier, il nous a permis de faire le premier calcul de la forme normale de Smith pour de très grandes matrices d'Homologie.

8

ÉTAT DE L'ART

« Pure mathematics, may it never be of any use to anyone. »

Henry John Stephen Smith

Sommaire

8.1 MOTIVATIONS

La forme normale de Smith joue un rôle très important dans l'étude des matrices rationnelles et polynomiales. Pour les matrices entières, elle permet le calcul actuel le plus rapide du déterminant [171 - Villard (2000), 58 - Eberly et al. (2000)]. En analyse Diophantienne, la forme de Smith permet de résoudre des systèmes linéaires en nombres entiers [124 - Newman (1972), 37 - Chou et Collins (1982)] ou des problèmes de combinatoire [172 - Wallis et al. (1972)], voir section 7.1.4, page 149. La forme de Smith intervient aussi en théorie des groupes : un groupe fini abélien est représenté par un ensemble de relations associé à une matrice entière ; la structure canonique du groupe ainsi que la caractérisation de tous ses sous-groupes peut être déduite de la forme de Smith de cette matrice [124 - Newman (1972), 82 - Iliopoulos (1989), 25 - Buchmann et al. (1997)]. De même, les invariants du groupe des classes d'équivalence de formes quadratiques binaires définies positives permettent de calculer son ordre et ainsi de résoudre le problème 304 des *Disquisitiones Arithmeticæ* de Gauß [69 - Gauß (1801), 77 - Hafner et McCurley (1989)].

Enfin, comme nous l'avons vu section 7.1.2, page 146, les invariants de Smith interviennent dans le calcul des groupes d'homologie de complexes simpliciaux [119 - Munkres (1994)].

8.2 MÉTHODES DIRECTES

L'algorithme classique de calcul de la forme de Smith est une variante de l'élimination de Gauß avec des calculs de pgcd [119 - Munkres (1994)]. Une ligne et une colonne sont construites de sorte qu'elles ne contiennent qu'un seul même élément non nul à leur intersection, le pivot. Cela est réalisé par des éliminations classiques successives de lignes et de colonnes. Ensuite, si un autre élément de la matrice n'est pas divisible par ce pivot, la colonne de cet élément est ajoutée à la colonne du pivot et des éliminations sont effectuées pour que le pgcd de ces deux éléments devienne le nouveau pivot. Quand le pivot obtenu divise tous les coefficients de la matrice, on passe à l'étape suivante. De nombreuses variantes sont possibles pour améliorer le comportement de l'algorithme (voir par exemple [140 - Rayward-Smith (1979)]). Néanmoins, sa complexité n'est pas polynomiale car la taille des éléments croît plus vite que n'importe quel polynôme en la taille de la matrice.

8.2.1 KANNAN ET BACHEM

Notations 8.2.1

- *Pour une matrice $A = (a_{ij})$, on note $||A|| = max_{ij}(|a_{ij}|)$.*
- *La notation $\widetilde{\mathcal{O}}(n)$ désigne $\mathcal{O}(nH(n))$ où $H(n)$ est une fonction polylogarithmique de n.*

Par exemple, $n.\log_2(n).\log_2(\log_2(n)) = \widetilde{\mathcal{O}}(n)$.

Le premier algorithme polynomial pour le calcul de la forme de Smith a été donné par Kannan et Bachem [7 - Bachem et Kannan (1979)] et amélioré par Chou et Collins [37 - Chou et Collins (1982)]. Iliopoulos [82 - Iliopoulos (1989)], enfin, donne un algorithme similaire calculant la forme de Smith d'une matrice $A \in \mathcal{A}^{m \times n}$ modulo le déterminant d'une sous-matrice inversible en

$$\mathcal{O}(s^5 I(s^2))$$

opérations binaires, où $I(l)$ est le coût de la multiplication de deux entiers de l bits et $s = m + n + \log_2 ||A||$.

8.2.2 A. STORJOHANN

Le meilleur algorithme déterministe est dû à A. Storjohann [157 - Storjohann (1996), 158 - Storjohann (2000)]. Si ω est l'exposant de la multiplication de matrices dans un anneau (c'est-à-dire 3 pour la multiplication classique ou $\log_2(7) \approx 2.81$ pour la multiplication de Strassen [159 - Strassen (1969)], le record actuel étant inférieur à 2.375477 [42 - Coppersmith et Winograd (1990)]), A. Storjohann calcule alors la forme de Smith en réduisant le calcul à des matrices bandes avec

$$\widetilde{\mathcal{O}}(m^{\omega-1} n I(m \log_2 ||A||))$$

opérations binaires.

8.2.3 PARALLÉLISME

E. Kaltofen, M. Krishnamoorthy et D. Saunders [85 - Kaltofen et al. (1989)], puis G. Villard [168 - Villard (1997a)] pour les matrices de passage, ont proposé un algorithme probabiliste pour calculer la forme de Smith et en

$$\mathcal{O}(\log_2{}^2 n)$$

opérations avec un nombre polynomial de processeurs. D'autre part, Neumann et Wilhemi [123 - Neumann et Wilhemi (1996)] ont étudié le comportement du calcul parallèle de la forme de Smith en pratique. Seulement, ils semblent n'en connaître que la version classique, de complexité non polynomiale.

8.3 MATRICES CREUSES ET MÉTHODES ITÉRATIVES

8.3.1 M. GIESBRECHT

Si la matrice considérée est creuse, les algorithmes probabilistes de M. Giesbrecht [72 - Giesbrecht (1995), 73 - Giesbrecht (1996)] sont meilleurs que ceux de A. Storjohann, puisqu'ils ont une complexité de $\mathcal{O}(m^2 \log_2 ||A||)$ produits matrice-vecteur et $\mathcal{O}(m^2 n \log_2 ||A|| + m^3 \log_2{}^2 ||A||)$ opérations additionnelles binaires. M. Giesbrecht plonge la matrice dans des grandes extensions disparates (*rough extensions*) de nombres composés puis calcule le polynôme caractéristique de la matrice préconditionnée dans l'extension. Après deux de ces calculs, les pgcd des coefficients des polynômes révèlent les invariants de Smith avec une forte probabilité. Si $\Omega \leq n^2$ est le coût de calcul d'un produit matrice-vecteur, et que le résultat est désiré correct avec une probabilité supérieure à $1 - \epsilon$, la complexité de l'algorithme de M. Giesbrecht est :

$$\tilde{\mathcal{O}}(\Omega m^2 \log_2 ||A|| \log_2 \epsilon^{-1} + m^2 n \log_2 ||A|| \log_2 \epsilon^{-1} + m^3 \log_2{}^2 ||A|| \log_2 \epsilon^{-1})$$

Toutefois, l'algorithme n'est pas très efficace en pratique du fait de l'utilisation des extensions disparates.

8.3.2 G. VILLARD

Le meilleur algorithme connu pour le calcul des invariants de Smith est celui de G. Villard [171 - Villard (2000), 58 - Eberly et al. (2000)]. Par remontée p-adique, il peut déterminer le $k^{\text{ième}}$ invariant en $\mathcal{O}(n(\log_2 n + \log_2 ||A||)^2 \log_2 n)$ produits matrice-vecteur. Ensuite, comme il n'y a que $\sqrt{\log_2 |detA|}$ invariants distincts, une recherche dichotomique permet alors le calcul de l'ensemble des invariants en seulement $\mathcal{O}(\log_2 n \sqrt{\log_2 |detA|})$ étapes. D'où une complexité globale en

$$\mathcal{O}(\Omega n^{1.5}(\log_2 n + \log_2 ||A||)^{2.5} {\log_2}^3 n)$$

pour une probabilité de succès de $1 - \frac{1}{2n}$. Cela induit le meilleur algorithme connu pour le calcul du déterminant d'une matrice entière creuse et même dense, car on obtient alors une complexité en $\tilde{\mathcal{O}}(n^{2+\frac{\omega}{2}})$! Toutefois, en pratique, il reste à éviter des préconditionnements assez coûteux (en $\mathcal{O}(\ln n)$, avec des constantes assez importantes, comme les préconditionnements Toeplitz, par exemple).

8.3.3 VALENCE

Nous proposons un nouvel algorithme [54 - Dumas et al. (2000), 55 - Dumas et al. (2001)], efficace en pratique pour certaines matrices creuses, facilement parallélisable, évitant les extensions disparates de M. Giesbrecht par des extensions de nombres premiers. Nous verrons que sa complexité n'est pas meilleure que celle de M. Giesbrecht, par exemple, mais, en pratique, c'est le seul algorithme actuel pour calculer certaines formes de Smith.

Le prochain chapitre étudie le calcul du polynôme minimal entier, brique essentielle de notre algorithme pour le calcul de la forme de Smith.

9

CALCUL PARALLÈLE DU POLYNÔME MINIMAL ENTIER

« Vive y cae como todos los frutos, no sólo tiene luz, no sólo tiene sombra, se apaga, se deshoja, se pierde entre las calles, se desploma en la tierra. »

Pablo Neruda

Sommaire

Nous étudions dans ce chapitre le calcul du polynôme minimal d'une matrice entière, par l'algorithme de Wiedemann. Comme cet algorithme est valide sur un corps, nous calculons le polynôme minimal modulo des nombres premiers, puis nous reconstruisons les coefficients entiers grâce au théorème Chinois (voir, par exemple, [68 - Gathen et Gerhard (1999), Théorème 5.8]). Nous commençons par la définition de la valence.

Définitions 9.0.1

- *La* **valence** *d'un polynôme est son dernier coefficient non nul. Par extension, la* **valence caractéristique** *d'une matrice est la valence de son polynôme caractéristique. La* **valence minimale**, *ou simplement* **valence** *d'une matrice, est la valence de son polynôme minimal.*
- *La* **valuation** *est le degré du monôme correspondant. La* **valuation caractéristique** *et la* **valuation minimale** *d'une matrice sont définies de même.*

Pour notre algorithme de calcul de la forme de Smith, par exemple, nous avons seulement besoin de la valence et non de la totalité du polynôme minimal. L'algorithme de calcul est le même, mais, pour calculer uniquement la valence, nous avons besoin de moins de mémoire.

9.1 RESTES CHINOIS

Nous calculons la valence minimale entière, v, d'une matrice B (la valence de son polynôme minimal sur les entiers) par reconstruction chinoise à partir des valences des polynômes minimaux modulo p, pour différents nombres premiers. L'algorithme a trois étapes. Nous calculons tout d'abord le degré du polynôme minimal par plusieurs essais modulo différents nombres premiers. Nous calculons alors une borne fine sur la taille de la valence en utilisant ce degré. Nous finissons par la reconstruction chinoise de valences obtenues modulo des nombres premiers.

La première question est de savoir combien de nombres premiers doivent être utilisés. Il faut que leur produit soit plus grand que la valence ; il faut donc estimer la taille de ce coefficient. Nous pouvons utiliser l'inégalité de Hadamard [68 - Gathen et Gerhard (1999), Théorème 16.6], qui borne les mineurs par $(B\sqrt{n})^n$, si B borne les coefficients de la matrice. Toutefois, cette borne induirait donc une utilisation de $\mathcal{O}(n)$ nombres premiers. Il parait alors important de réduire ce nombre. Dans les deux prochaines sections, nous développons deux méthodes à cette fin.

La première consiste à utiliser une terminaison anticipée dans un algorithme de reconstruction itératif. D'autre part, pour un calcul déterministe, il est intéressant d'avoir une évaluation plus fine. Nous montrons alors l'emploi des ovales de Cassini pour borner le rayon spectral de la matrice et donc la valence. Nous terminons le chapitre par des considérations sur les probabilités de réussite de cet algorithme et des expérimentations parallèles.

9.2 TERMINAISON ANTICIPÉE

Notation 9.2.1

Soient a, b, $c \in \mathbb{Z}$. Par la suite, nous notons $a \equiv b[c]$, ou parfois $a = b \mod c$, la relation a est congru à b modulo c, c'est-à-dire qu'il existe $d \in \mathbb{Z}$ tel que $a - b = cd$.

Dans les calculs que nous reportons ici, nous avons calculé v_{p_i}, la valence minimale modulo p_i, pour différents nombres premiers choisis au hasard aux alentours de 2^{16}, ou, au pire, 2^{23}. Nous acceptons le résultat de la reconstruction chinoise dès que le produit de ces nombres premiers dépasse la borne de Hadamard ou la borne de Cassini, comme discuté section suivante. Toutefois, quand ces deux bornes sont grandes, nous utilisons une condition probabiliste de terminaison anticipée. Soit v la valence entière et soit v_k le reste de v modulo $M = \prod_{i=1}^{k} p_i$ pour plusieurs nombres premiers p_i, choisis au hasard. On a alors $0 \leq v_k < M$ et $v_k \equiv v[M]$. Supposons, maintenant, que nous croyons que $v_k = v$, c'est-à-dire que nous supposons avoir suffisamment de nombres premiers, même si M est plus petit que notre borne (on peut raisonnablement supposer cela, par exemple si v_k reste identique pour plusieurs k successifs). Dans ce cas, il est possible de faire un test probabiliste rapide de cette supposition de la manière suivante : choisir un autre nombre premier p^*, au hasard dans un ensemble suffisamment grand, et calculer v^*, la valence minimale modulo p^*. Alors $0 \leq v^* < p^*$ et $v^* \equiv v[p^*]$. Faire aussi la réduction $v_k \mod p^*$, et donc obtenir v_k^* telle que $0 \leq v_k^* < p^*$ et $v_k^* \equiv v_k[p^*]$. Alors, si on a bien l'égalité $v_k = v$, les deux valeurs, modulo p^* doivent aussi être égales. Il y a deux cas. Si ces deux valeurs modulo p^* sont différentes, clairement, notre calcul n'est pas terminé. D'un autre côté, si les deux valeurs modulo p^* sont égales, alors v_k est v avec une bonne probabilité. Nous explicitons cette probabilité dans le lemme suivant.

Lemme 9.2.2

Soit $v \in \mathbb{Z}$ et une borne supérieure U telle que $v < U$. Soient P un ensemble de nombres premiers distincts et $P_k = \{p_1 \dots p_k, p^*\}$ un sous-ensemble de P choisi au hasard. Soient $M = \prod_{i=1}^{k} p_i$ et ℓ une borne inférieure telle que $\min(P_k) > \ell > 1$. Soient $v_k \equiv v[M]$, $v^* \equiv v[p^*]$ et $v_k^* \equiv v_k[p^*]$ comme ci-dessus. Si $v_k^* = v^*$, alors $v = v_k$ avec une probabilité d'au moins $1 - \frac{\log_\ell(\frac{U-v_k}{M})}{|P|}$.

Preuve de 9.2.2 : La seule manière de donner une réponse erronée est d'avoir en même temps $v \neq v_k$ et $v_k^* = v^*$. D'autre part, comme $v_k \equiv v[M]$ et $v_k \equiv v[p^*]$, alors, comme M et p^* sont premiers entre eux, p^* doit diviser $\frac{v-v_k}{M}$. Pour finir, on voit qu'il y a au plus $\log_\ell(\frac{v-v_k}{M})$ nombres premiers distincts qui divisent ce quotient. La probabilité d'erreur est donc au plus $\frac{\log_\ell(\frac{v-v_k}{M})}{|P|}$ puisque $|P|$ est le cardinal de P. On termine en constatant que $v - v_k < U - v_k$. □

Par exemple, le pire cas donné dans le tableau 9.1, page 178 est obtenu pour une matrice avec une valence bornée par $U = 117^{827}$. Nous choisissons des nombres premiers plus grands que $\ell = 2^{15}$. Forcément, M, le produit des nombres premiers, est plus grand que 2 ; ainsi $\log_\ell(\frac{U}{M}) \leq 379$. D'autre part, il y a exactement 3030 nombres premiers entre 2^{15} et 2^{16}. Donc, en choisissant un nombre premier dans cet intervalle, nous avons déjà 87% de chances d'être correct. En outre, en répétant ce test quatre fois, cette probabilité de réussite atteint au moins 99.97%. Enfin, en général, pour les matrices d'homologie classiques, la borne est plus proche de 10^{200} ; dans ce cas, un seul test donne une confiance de plus de 98.5%.

9.3 OVALES DE CASSINI

Pour un calcul déterministe, ou pour des calculs parallèles, une borne fine sur la taille est souvent utile. La borne de Hadamard peut être utilisée, mais elle est souvent beaucoup trop pessimiste pour de nombreuses matrices creuses. C'est pourquoi nous utilisons plutôt une borne déterminée par les disques de Gershgörin et les ovales de Cassini. Cette borne est de la forme β^d avec β une borne sur les valeurs propres et d le degré du polynôme minimal.

Le $i^{\text{ème}}$ disque de Gershgörin est centré sur a_{ii} et a pour rayon la somme des valeurs absolues des autres éléments de la ligne i ($R_i = \sum_{j \neq i} |a_{ij}|$), c'est-à-dire

que le disque i est la surface $\{\lambda, |\lambda - a_{ii}| < R_i\}$. Le théorème de Gershgörin précise que toutes les valeurs propres d'une matrice sont contenues dans l'union de ces disques [20 - Brauer (1946), 161 - Taussky (1948), 74 - Golub et Van Loan (1996)]. Il est possible de pousser plus loin et de considérer les ovales de Cassini qui produisent des bornes plus fines. Chacun de ces ovales est une région du plan complexe $\{\lambda, |\lambda - a_{ii}||\lambda - a_{jj}| < R_i R_j\}$. En outre, toutes les valeurs propres sont aussi contenues dans l'union des ovales, puisque chaque ovale est contenu dans l'union de ses deux disques de Gershgörin [21 - Brauer (1947), 165 - Varga (2000)]. Par exemple, la matrice complexe

$$A = \begin{bmatrix} -4-9i & i & 9 \\ -3 & 2+8i & 10 \\ 0 & 7i & -18 \end{bmatrix}$$

donne les disques de Gershgörin (surfaces pleines) et les ovales de Cassini (contours) de la figure 9.1, page 169 (les valeurs propres sont indiquées par des croix), obtenue par l'implémentation de Varga et Krautstengl [166 - Varga et Krautstengl (1999)]. Une borne sur le module des valeurs propres est donc obtenue par le

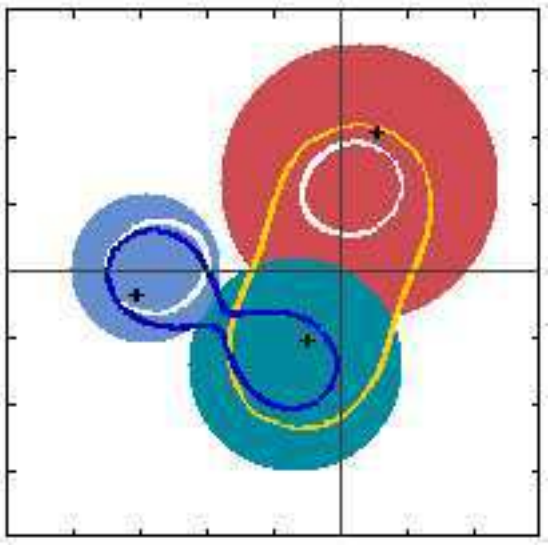

FIGURE 9.1 – Disques de Gershgörin (surfaces pleines) et ovales de Cassini (contours)

calcul du module maximal que peut prendre un point des ovales. Il est possible d'utiliser des bornes plus précises sur les valeurs propres, voir par exemple [24 - Brualdi et Mellendorf (1994), Théorème 6]. En pratique, sur des matrices très creuses, les disques ou les ovales sont assez faciles à calculer et donnent une borne relativement fine.

On obtient alors la borne suivante sur les coefficients du polynôme minimal.

Proposition 9.3.1

Soit $B \in \mathbb{C}^{n\times n}$ de rayon spectral borné par β. Soit $\pi_B(X) = \sum_{k=0}^{d} m_i X^i$. Alors, $|\text{valence}(B)| \leq \beta^d$, et $\forall i \in [\![0,d]\!]$, $|m_i| \leq \max(\sqrt{d}\beta, \beta)^d$.

Preuve de 9.3.1 : Il suffit de noter que la valence est le produit de d valeurs propres et que $|m_i| \leq \mathsf{C}_d^i \beta^i$ [113 - Mignotte (1989), Théorème IV.§4.1]. On borne ensuite chacun de ces coefficients par β^d quand $\beta \geq d$ ou $d^{\frac{d}{2}}\beta^{\frac{d}{2}}$ quand $\beta \leq d$. □

Pour des matrices avec des éléments de taille constante, β et d sont $\mathcal{O}(n)$. Cependant, quand d et/ou β est petit par rapport à n (en particulier si d est dans ce cas), cette borne peut être une amélioration importante par rapport à la borne de Hadamard, puisque cette dernière serait d'ordre $n\ln(n)$ au lieu de $d\ln(\beta)$. C'est le cas des matrices d'homologie. En effet, pour celles-ci, d'une part AA^t a un polynôme minimal de très petit degré et, de plus, nous allons voir que β est relativement faible car AA^t est à diagonale dominante.

Il reste à calculer β, la borne sur le rayon spectral. Nous remarquons qu'il est cher de calculer une limite quelconque parmi celles qui sont mentionnées ci-dessus tout en restant strictement dans le modèle de boîte noire. Il semble qu'un produit matrice-vecteur avec B soit nécessaire pour extraire chaque ligne ou colonne. De même, si $B = AA^T$, il faut alors deux produits avec A. Mais, si on a accès aux éléments de A, une limite pour le rayon spectral de AA^T est facilement obtenue avec très peu d'opérations en précision arbitraire :

Algorithme 9.3.2 *Borne-Ovales-de-Cassini*

Entrées : – une matrice $A \in \mathbb{C}^{m\times n}$.

Sorties : – $\beta \in \mathbb{R}_+$. Toute valeur propre λ de AA^t est de module borné par β

Centres

1 : **Pour** $i = 1$ **jusqu'à** m **Faire**

2 : $\quad q_i = \sum_{1\leq j\leq n} a_{ij}^2$

Rayons

3 : Former $|A|$, la matrice dont les éléments sont les valeurs absolues des éléments de A

4 : $v = |A||A|^T[1,1,\ldots,1]^T$

5 : **Pour** $i = 1$ **jusqu'à** m **Faire**

6 : $r_i = v_i - |q_i|$

Borne de Gershgörin

7 : $q = \max_{1 \leq i \leq m} |q_i|$

8 : i_1 est tel que $r_{i_1} = \max_{i \in [\![1,m]\!]} r_i$

Borne de Cassini

9 : i_2 est tel que $r_{i_2} = \max_{i \in [\![1,m]\!] \setminus \{i_1\}} r_i$

10 : $\beta = q + \sqrt{r_{i_1} r_{i_2}}$

Dans la suite, pour une matrice $A \in \mathbb{C}^{n \times m}$, nous notons $\Omega = \max(n, m,$ nombre d'éléments non nuls de $A)$. Ainsi, 2Ω borne le nombre d'opérations nécessaires au calcul d'un produit matrice-vecteur, Ax, et d'un produit scalaire, $x^T x$.

Théorème 9.3.3

Soit $A \in \mathbb{C}^{m \times n}$ avec Ω comme ci-dessus. L'algorithme Borne-Ovales-de-Cassini *calcule une borne supérieure sur les valeurs propres de AA^t, avec moins de 7Ω opérations et $3n$ comparaisons.*

Preuve de 9.3.3 : Nous utilisons le fait que les valeurs propres sont dans l'union des ovales. Maintenant, prenons q_1, q_2, r_1, r_2 les deux centres et rayons d'un tel ovale. Alors, tout point λ de cet ovale vérifie : $|\lambda - q_1||\lambda - q_2| \leq r_1 r_2$ [21 - Brauer (1947), Théorème 1]. Nous voulons connaître le module maximal d'un tel λ. Dans un premier cas, si $|\lambda - q_2| \leq |\lambda - q_1|$, alors $|\lambda - q_2| \leq \sqrt{r_1 r_2}$ et donc, comme $|\lambda| - |q_2| \leq |\lambda - q_2|$, on obtient $|\lambda| \leq |q_2| + \sqrt{r_1 r_2}$. Le deuxième cas est exactement symétrique, en remplaçant q_2 par q_1. Cela prouve donc que β, calculée par l'algorithme, est bien une borne sur les valeurs propres. Enfin, l'analyse de complexité est triviale. □

9.4 DEGRÉ DU POLYNÔME MINIMAL ET NOMBRE PREMIER TROMPEUR

Pour calculer le polynôme minimal d'une matrice modulo des nombres premiers, nous utilisons l'algorithme de Wiedemann ; mais, pour calculer la valence, nous devons connaître le degré du polynôme entier. Pour calculer ce degré, nous

choisissons des nombres premiers au hasard ; le degré du polynôme minimal entier sera le degré maximal des polynômes minimaux modulo p avec une forte probabilité.

En effet, certains nombres premiers peuvent donner un polynôme minimal de plus petit degré. Nous les appelons nombres premiers **trompeurs** (*bad prime*). Il faut donc borner la probabilité de tomber sur un de ces nombres. Pour cela, nous bornons la taille d'un mineur de la matrice qu'un tel trompeur doit diviser.

Soit δ, le degré du polynôme minimal entier d'une matrice B. Il existe un vecteur u tel que le sous-espace de Krylov associé, $Krylov(u, B)$, soit de rang δ. Ce fait est facilement prouvé en considérant, par exemple, la forme de Frobenius rationnelle de B. Donc, il existe un mineur non nul, de dimension $\delta \times \delta$ dans la matrice $[u, Bu, \ldots, B^{n-1}u] = M_\delta$. Un trompeur doit diviser ce mineur. Ainsi, toute borne sur M_δ induit une borne sur le nombre de trompeurs. Par exemple, soit β une borne supérieure sur la norme des lignes de B, en utilisant la borne de Hadamard [68 - Gathen et Gerhard (1999), Theorem 16.6], on peut affirmer que $|M_\delta| \leq ||u||.||Bu|| \ldots ||B^\delta u|| \leq \beta^{\frac{\delta^2}{2}}||u||$. En outre, Ozello a prouvé qu'il existait un vecteur u avec des entrées de petites valeurs absolues, i.e. plus petites que $\lceil \frac{\delta}{2} \rceil$, tel que son polynôme minimal *est* celui de B, $\pi_{u,B} = \pi_B$ [133 - Ozello (1987), Théorème III.4.a]. Il est alors possible de borner $||u||$ par $\lceil \frac{\delta}{2} \rceil \sqrt{n}$, puis $|M_\delta|$ par $\lceil \frac{\delta}{2} \rceil \sqrt{n} \beta^{\frac{\delta^2}{2}} = U$.

Une fois qu'une telle borne est calculée, il ne reste plus qu'à choisir des nombres premiers dans un ensemble suffisamment grand ; ainsi, il y aura peu de chances qu'ils divisent ce mineur. Supposons que l'on choisisse des nombres premiers au hasard, plus grands que ℓ. Alors il ne peut y avoir plus de $\log_\ell(U)$ d'entre eux divisant M_δ. La distribution des nombres premiers nous assure alors qu'un ensemble P suffisamment large peut être construit sans que les nombres premiers soient excessivement grands. Par exemple, ces bornes sur p_k, le $k^{\text{ième}}$ nombre premier, sont connues :

$$k\big(\ln(k) + \ln(\ln(k)) - 1\big) \leq p_k \qquad k \geq 2 \qquad (9.1)$$

$$p_k \leq k\left(\ln(k) + \ln(\ln(k)) - 1 + 1.8\frac{\ln(\ln(k))}{\ln(k)}\right) \qquad k \geq 13 \qquad (9.2)$$

$$p_k \leq k\big(\ln(k) + \ln(\ln(k)) - 0.9427\big) \qquad k \geq 15985 \qquad (9.3)$$

$$p_k \leq k\left(\ln(k) + \ln(\ln(k)) - 1 + \frac{\ln(\ln(k)) - 1.8}{\ln(k)}\right) \qquad k \geq 27076 \qquad (9.4)$$

Dans la suite on note $\eta(k)$ un nombre plus grand que le $k^{\text{ième}}$ nombre premier. Il est possible de calculer un tel η en utilisant une combinaison des inégalités

(9.2), (9.3), (9.4) de Massias et Robin [111 - Massias et Robin (1996), Théorème A]. Nous noterons aussi $\mu_l(x)$ une borne inférieure sur le nombre de nombres premiers distincts entre l et x ; cette borne est facilement calculée à partir de la réciproque de η, et de l'inégalité (9.1) de Dusart [57 - Dusart (1999)], ou encore à partir de bornes sur $\pi(x)$, le nombre exact de nombres premiers plus petits que x, [56 - Dusart (1998), Théorème 1.10] :

$$\pi(x) \leq \frac{x}{\ln(x)}\left(1 + \frac{1.2762}{\ln(x)}\right) \qquad x \geq 1 \tag{9.5}$$

$$\pi(x) \leq \frac{x}{\ln(x) - 1.1} \qquad x \geq 60184 \tag{9.6}$$

$$\pi(x) \leq \frac{x}{\ln(x)}\left(1 + \frac{1}{\ln(x)} + \frac{2.51}{\ln^2(x)}\right) \qquad x \geq 355391 \tag{9.7}$$

$$\pi(x) \leq \frac{x}{\ln(x)}\left(1 + \frac{1.0992}{\ln(x)}\right) \qquad x \geq 13320000000 \tag{9.8}$$

Enfin, il est très important de réduire U, de manière à prendre de petits nombres premiers. Cette borne dépend de la taille des éléments du vecteur u et de δ. Or, comme δ n'est pas connu a priori, on ne peut le borner que par n. La valeur de $\log_\ell(U)$ peut alors être assez grande en pratique ($O(n^2)$). Nous montrons dans la suite qu'à partir de calculs préliminaires, il est possible d'utiliser les degrés calculés de polynômes minimaux à la place de n, pour affiner U.

Pour prouver cela, nous avons besoin d'une généralisation du théorème d'Ozello [133 - Ozello (1987), Theorem III.4.a]. Il s'agit de borner la taille des éléments d'un vecteur ayant un polynôme minimal d'au moins un certain degré.

Lemme 9.4.1

Soit B une matrice symétrique de $\mathbb{Z}^{n\times n}$ avec un polynôme minimal sur $\mathbb{Z}$ de degré δ. Pour tout $d \leq \delta$, il existe un vecteur u avec des éléments entiers de valeur absolue plus petite que $\lceil \frac{d}{2} \rceil$ tel que $\pi_{u,B}$ est de degré au moins d.

Preuve de 9.4.1 : Il faut considérer $u = [U_1, U_2, \ldots, U_n]$ comme un vecteur d'indéterminées et former la matrice polynomiale $C_d(U_1, U_2, \ldots, U_n)$ avec comme colonnes $u, Bu, \ldots, B^{d-1}u$. Nous savons qu'il existe au moins un vecteur u_0 dans $\mathbb{Z}$ tel que son sous-espace de Krylov associé, $Krylov(u_0, B)$, est de rang δ. Donc, il existe un mineur de C d'ordre δ non identiquement nul. Donc, comme $d \leq \delta$, il existe un mineur de C d'ordre d non identiquement nul. Ce mineur est un polynôme homogène pour les U_i, de degré total d. Alors, ce mineur ne peut pas avoir plus de ds^{n-1} racines dans $\mathbb{Z}/_s\mathbb{Z} \subset \mathbb{Z}$, d'après le lemme de Zippel-Schwartz (voir [177 - Zippel (1993)] ou [68 - Gathen et Gerhard (1999), Lemme

6.44]). Tout d'abord, si nous choisissons $s = d + 1$, nous voyons qu'il y a au plus $d(d+1)^{n-1}$ racines de ce polynôme dans un espace à $(d+1)^n$ éléments. Donc il doit exister au moins un vecteur $u_{[d+1]}$ dans $\mathbb{Z}/(d+1)^n\mathbb{Z}$ pour lequel le mineur n'est pas nul dans $\mathbb{Z}$. Alors, ce vecteur, maintenant pris comme un vecteur de $\mathbb{Z}^n$, avec des éléments positifs et négatifs de valeur absolue plus petite que $\lceil \frac{d}{2} \rceil$, a donc un polynôme minimal de degré au moins d sans quoi, $C_d(u_{[d+1]})$ ne serait pas de rang d. □

Effectivement, en prenant $d = \delta$, on retrouve le théorème d'Ozello.

9.5 POLYNÔME MINIMAL ENTIER

Nous donnons maintenant l'algorithme complet pour le calcul du polynôme minimal entier en terminant la section par une analyse des probabilités de réussite. Nous utilisons plusieurs exécutions de l'algorithme de Wiedemann, avec calcul du plus petit commun multiple, comme dans [174 - Wiedemann (1986)], pour obtenir une probabilité de réussite modulo p, d'au moins $1 - \frac{1}{p}$. Aussi, nous avons pris 2^{15} comme taille minimale de nombre premier. Ce paramètre peut bien évidemment être augmenté si nécessaire ; nous le fixons ici pour plus de précisions sur les bornes de nombres premiers.

Algorithme 9.5.1 *Polynôme-Minimal-Entier*

Entrées : – une matrice A dans $\mathbb{Z}^{n\times n}$.
– une tolérance ϵ, telle que $0 < \epsilon < 1$.
– une borne supérieure m sur les nombres premiers pour lesquels les calculs sont rapides, $m > 2^{15}$.

Sorties : – le polynôme minimal de A, correct avec une probabilité d'au moins $1 - \epsilon$.

Ensemble initial de nombre premiers

1 : $\ell = 2^{15}$

2 : $d = 0$; $F = \emptyset$; $P = \emptyset$;

3 : $\beta =$ Borne de Cassini de A ;

4 : $M = m$ {Les calculs seront rapides}

Polynômes modulo p_i

5 : **Répéter**
6 : Choisir un nombre premier p_i, avec $\ell < p_i < M$. {ils sont au moins $\mu_\ell(M)$}
7 : Calculer w_{A,p_i} par Wiedemann. $\{ = \pi_{A,p_i}$ avec probabilité au moins $1 - \frac{1}{p_i}\}$
8 : **Si** $deg(w_{A,p_i}) > d$ **Alors**
9 : $d = deg(w_{A,p_i})$; $F = \{p_i\}$; $P = \{w_{A,p_i}\}$;
10 : $U = \sqrt{n}\lceil \frac{d}{2} \rceil \beta^{\frac{(d+1)^2}{2}}$;
11 : $bad = \log_\ell(U)$; { au plus $\log_\ell(U)$ trompeurs}
12 : $b_i = 2 \times bad(1 + \frac{2}{l-2}) + 3512$; { 3512 premiers $< 2^{15}$}
13 : $M_i =$ borne supérieure de $\pi(b_i)$; { au moins b_i premiers $< M_i$}
14 : **Si** $(M_i > M)$ **Alors** $M = M_i$; { Les calculs seront plus lents}
{ Les degrés seront corrects avec une probabilité au moins $\frac{1}{2}$}
15 : **Sinon, si** $deg(\pi_{A,p_i}) = d$ **Alors**
16 : $F = F \cup \{p_i\}$;
17 : $P = P \cup \{\pi_{A,p_i}\}$;
18 : **Jusqu'à ce que** $\prod_F p_i \geq \max\{\sqrt{d\beta}; \beta\}^d$ **et** $\prod_F (\frac{1}{p_i} + \frac{bad}{\mu_l(M)}) \leq \epsilon$
Restes Chinois
19 : Renvoyer $\pi_A = \sum_{j=1}^{d} \alpha_j X^j$, où chaque $\alpha_j \in \mathbb{Z}$ est reconstruit à partir de P et F.

THÉORÈME 9.1. *L'algorithme* Polynôme-Minimal-Entier *est correct.*
Soit $s = d\ max\{\log(\beta(A)); \log(d)\}$*, une borne sur la taille des coefficients du polynôme minimal. L'algorithme nécessite*

$$\mathcal{O}(d\Omega I(\log(s))(\log(s) + \log(\epsilon^{-1}))) \text{ c'est-à-dire } \tilde{\mathcal{O}}(n\Omega \log(\epsilon^{-1}))$$

opérations binaires en moyenne pour des entrées de taille constante. Il nécessite en outre $\mathcal{O}(n \log(s) + ds)$ *unités mémoire si tous les coefficients sont calculés et* $\mathcal{O}(n \log(s) + s)$*, c'est-à-dire* $\tilde{\mathcal{O}}(n)$*, si seule la valence est calculée.*

Preuve de 9.5.1 : Pour la correction, la première difficulté est de pouvoir choisir des petits nombres premiers, i.e. proches de la taille du mot machine, pour utiliser les calculs rapides. Le problème est que la borne de la section 9.4, page 171 peut être grande à cause de l'exposant en n^2. Cependant, il est possible de commencer avec de petits nombres premiers puis d'ajuster la borne au fur et à mesure que les degrés sont calculés. En effet, considérons de nouveau le vecteur u tel que le sous-espace de Krylov associé à B et u, $Krylov(u, B)$, soit de rang δ, le degré du polynôme minimal entier, et supposons qu'un nombre premier p produise un

polynôme de degré d. Alors $Krylov(u, B)$ est de rang d modulo p, et donc p doit diviser un mineur $(d+1) \times (d+1)$ des $d+1$ premières colonnes. Il y a au plus $\beta^{\frac{(d+1)^2}{2}}$ nombres premiers de ce type. Il est alors possible d'affiner la borne sur la taille des nombres premiers de $\frac{n^2}{2}\log(\beta)$ à $\frac{(d+1)^2}{2}\log(\beta)$.

La seconde difficulté est la terminaison de la boucle. Le premier membre de la condition d'arrêt assure un nombre suffisant de nombres premiers pour la reconstruction. Si l'on calcule seulement la valence, cela peut être réduit de $\max(\sqrt{d\beta}, \beta)^d$ à, simplement, β^d. Le deuxième membre assure une probabilité de réussite suffisamment grande. Si il y a $|F|$ polynômes du même degré, alors soit ils sont tous corrects, puisque qu'ils divisent forcément le vrai polynôme, soit ils sont tous faux. Comme la probabilité qu'un d'entre eux soit faux est la probabilité que l'algorithme de Wiedemann ait échoué ($\frac{1}{p_i}$) plus la probabilité que p_i ait été trompeur, cette dernière probabilité est bornée par le nombre de trompeurs sur le nombre total de nombre premiers dans notre ensemble, soit $\frac{bad}{\mu_l(M)}$.

Nous nous intéressons maintenant à la complexité. La valence est bornée par β^d. Pour la stocker et pour stocker les nombres premiers, où, de manière équivalente, pour stocker ses restes modulo p_i et les p_i, nous avons besoin de $\mathcal{O}(s)$ unités mémoire ; $\mathcal{O}(ds)$ pour le polynôme complet. L'algorithme de Wiedemann n'utilise qu'un nombre constant de polynômes et de vecteurs sur un corps premier. Ces nombres premiers sont bornés par M, et donc de taille $\log(M) = \log(\frac{b_i}{\log(b_i)})$, d'après l'inégalité de Dusart (9.5), ce qui vaut $\mathcal{O}(\log(d^2 \log_\ell(\beta))) = \mathcal{O}(\log(s))$. Enfin, chaque vecteur étant de taille n, le volume mémoire nécessaire pour les stocker est $\mathcal{O}(n\log(s))$.

Nous terminons la preuve par le temps en moyenne. À chaque itération, l'algorithme de Wiedemann nécessite $\mathcal{O}(d\Omega)$ opérations sur son corps de base, chacune de celles-ci nécessitant au plus $I(\log(s))$ opérations binaires. Or, $I(n) = 2n^2$ avec la multiplication classique, $I(n) = \mathcal{O}(n^{\log_2(3)})$ avec la méthode de Karatsuba et $I(n) = \mathcal{O}(n\ln(n)\ln(\ln(n)))$ avec la méthode de Schönhage et Strassen [68 - Gathen et Gerhard (1999)]. Ensuite, $\log(s)$ bons nombres premiers (**révélateurs**) sont nécessaires pour reconstruire la valence. Or,

$$\frac{1}{p_i} + \frac{bad}{\mu_\ell(M)} \leq \frac{1}{\ell} + \frac{bad}{2(1+\frac{2}{\ell-2})bad} = \frac{1}{2}$$

donc la probabilité de réussite de $1-\epsilon$ est obtenue avec au plus $\frac{-\log(\epsilon)}{-\log(0.5)} = \mathcal{O}(\log(\frac{1}{\epsilon}))$ bons nombres premiers. Maintenant, pour chaque itération, le polynôme de Wiedemann est correct avec une probabilité d'au moins $\frac{1}{2}$. Ainsi, le nombre moyen d'itérations ne dépasse pas le double du nombre de révélateurs

nécessaires et la complexité de cette partie est donc

$$\mathcal{O}\Big(d\Omega I(\log(s))max\{\log(s);\log(\epsilon^{-1})\}\Big).$$

Enfin, la reconstruction par restes chinois a un coût en $\mathcal{O}(I(s)\log(s))$ pour chaque coefficient du polynôme [68 - Gathen et Gerhard (1999), Théorème 10.25], donc en $\widetilde{\mathcal{O}(ds)}$ si tout le polynôme est calculé. □

En pratique, le nombre réel de nombres premiers, plus grands que 2^{15}, et divisant la valence des matrices d'homologie est très faible (pas plus de 50, en général). Nous avons donc souvent pris des nombres premiers entre 2^{15} et 2^{16}, où ils sont 3030. Cela induit, au pire, seulement 1.7% de trompeurs. Si l'on a 10 polynômes, cela réduit la probabilité d'échec à moins de 2×10^{-16} !

9.6 PARALLÉLISATION

La parallélisation de ce dernier algorithme, 9.5.1, page 174, est assez naturelle ; il s'agit de lancer le calcul de la borne sur les valeurs propres en parallèle avec un certain nombre de calculs de polynômes minimaux. Nous avons choisi d'ordonnancer cette première phase avec un algorithme de liste et, néanmoins, un seuil maximal de tâches en attente par nœud. Cela permet de répartir un peu plus les tâches dans la première phase. En effet, seul un petit nombre de tâches étant créées au début, cette astuce permet d'éviter un facteur 2 [52 - Dumas (2000)]. Ensuite, la borne sur les coefficients du polynôme entier, et donc le nombre total de polynômes nécessaires, peut être calculée. Ainsi, un deuxième groupe de polynômes minimaux est lancé avec un algorithme de liste classique. Enfin, un algorithme rapide de restes chinois est lancé en parallèle pour chacun des coefficients.

Nos expériences ont été menées sur une grappe de 20 Sun Microsystem Ultra Enterprise 450 avec, chacun, 4 processeurs à 250 MHz Ultra-II et 1024 ou 512 Mo. L'arithmétique utilisée est celle des corps finis *GFq*. Les algorithmes sont intégrés à LINBOX et utilisent ATHAPASCAN comme support d'exécution parallèle. Dans le tableau 9.1, page 178, nous présentons les calculs du polynôme minimal entier pour un échantillon de matrices de complexes simpliciaux de taille $m \times n$, avec ϖ éléments non nuls par ligne et de rang r sur les entiers. Pour celles-ci, le degré du polynôme minimal est très petit comparativement à la taille et au rang des matrices, ce qui permet des calculs de polynômes minimaux modulo p_i extrêmement

Matrice	$\varpi, m \times n, r$	CB	Rem	d_m	bad	M	v_m	Séquentiel	Parallèle	Gain
mk9.b3	4, 945x1260, 875	12	2	X^6	6	2^{16}	$2^6.3^4$	0.31	0.39	0.79
mk10.b3	4, 4725x3150, 2564,	24	3	X^7	10	2^{16}	$2.3^4.5^3.7.13$	1.16	0.81	1.43
mk11.b4	5, 10395x17325, 10143	15	2	X^8	11	2^{16}	$-2^4.3^2.5^2.7.11$	4.00	1.80	2.22
mk12.b3	4, 51975x13860, 12440	60	3	X^7	13	2^{16}	$2^{10}.3^5.5.7.11.13$	16.68	5.55	3.01
mk12.b4	5, 62370x51975, 39535	30	4	X^{11}	24	2^{16}	$-2^{13}.3^7.5.7.11$	34.49	11.39	3.03
mk13.b4	5, 270270x135135, 111463	50	4	X^{11}	28	2^{16}	$-23.2^{11}.3^7.5^2.7.11.13$	149.06	42.13	3.54
mk13.b5	6, 135135x270270, 134211	18	5	X^{12}	24	2^{16}	$2^{10}.3^5.5^2.11.13$	96.11	29.41	3.27
m133.b3	4, 200200x200200, 168310	16	4	X^{11}	20	2^{16}	$2^7.3^5.5.7.11.13$	94.05	33.31	2.82
ch4-4.b2	3, 96x72, 57	12	1	X^4	3	2^{16}	$2^7.3$	0.08	0.24	0.33
ch5-5.b3	4, 600x600, 424	16	3	X^8	11	2^{16}	$2^5.3^3.5^2.7$	0.35	0.39	0.90
ch6-6.b4	5, 4320x5400, 3390	20	3	X^{10}	18	2^{16}	$2^{10}.3^5.5.11$	1.83	1.43	1.28
ch7-7.b4	5, 52920x29400, 22884	45	6	X^{15}	48	2^{16}	$-19.17.2^{11}.3^7.5^3.7^3.11.13$	45.17	10.28	4.39
ch7-7.b5	6, 35280x52920, 29448	24	5	X^{12}	27	2^{16}	$2^7.3^4.5^2.7^2.11.13$	23.88	7.52	3.18
ch7-8.b5	6, 141120x141120, 92959	36	7	X^{21}	84	2^{16}	$19.17.2^{18}.3^9.5^4.7^3.11.13$	123.18	24.85	4.96
ch7-9.b3	4, 105840x17640, 16190	96	7	X^{16}	64	2^{16}	$29.19.2^{19}.3^{10}.5^5.7^4.11.13.31$	82.57	15.04	5.49
ch7-9.b4	5, 317520x105840, 89650	75	9	X^{22}	111	2^{16}	$-1433.C_4$	471.22	64.67	7.29
ch7-9.b5	6, 423360x317520, 227870	48	9	X^{24}	117	2^{16}	$23.19.17.2^{24}.3^{10}.5^4.7^4.11^2.13^2$	775.33	106.15	7.30
ch8-8.b4	5, 376320x117600, 100289	80	7	X^{16}	62	2^{16}	$29.19.17.2^{22}.3^8.5^4.7^2.11.13$	386.13	64.78	5.96
ch8-8.b5	6, 564480x376320, 276031	54	7	X^{19}	78	2^{16}	$-19.17.2^{21}.3^9.5^4.7^2.11^2.13$	732.44	127.57	5.74
ch8-8.b6	7, 322560x564480, 279237	28	5	X^{14}	37	2^{16}	$2^{15}.3^4.5^3.7.11.13$	337.72	77.52	4.36
ch8-8.b7	8, 40320x322560, 40320	8	1	X^1	1	2^{16}	-2^3	10.63	10.47	1.02
ch9-9.b3	4, 381024x42336, 39824	144	5	X^{10}	30	2^{16}	$-73.6287.C_2$	190.74	45.10	4.23
ch9-9.b4	5, 1905120x381024, ? ?	125	8	X^{17}	76	2^{16}	$-2^{13}.3^{12}.5^6.7^3.13^2.C_5$	2279.32	358.97	6.35
n2c6.b6	7, 5715x4945, 2943	63	25	X^{66}	895	2^{16}	$17.2^{26}.3^{21}.5^2.7.13^2.C_{10}$	55.43	3.44	16.11
n2c6.b7	8, 3990x5715, 2772	64	58	X^{152}	4684	2^{18}	$17^2.2^{83}.3^{55}.5^{50}.7^3.13^2.C_{11}$	206.18	7.51	27.45
n3c6.b9	10, 2511x4935, 1896	60	15	X^{40}	332	2^{16}	$17^2.2^9.3^{13}.5^8.13^4.7.C_4$	15.80	1.83	8.63
n4c5.b5	6, 4340x2852, 1866	60	39	X^{103}	2131	2^{17}	$-2^{16}.3^{10}.5^{16}.13^2.191^2.89^2.C_{21}$	82.76	4.04	20.49
n4c5.b6	7, 4735x4340, 2474	63	82	X^{216}	9385	2^{18}	$19.17.2^{45}.3^{10}.5^{26}.7.67^2.59^2.C_{34}$	401.05	11.49	34.90
n4c5.b7	8, 3635x4735, 2261	64	100	X^{263}	13944	2^{19}	$-17.2^{47}.3^{13}.5^{28}.11.13^2.59^2.C_{36}$	511.69	14.26	35.88
n4c5.b8	9, 1895x3635, 1374	63	62	X^{164}	5427	2^{18}	$17^2.2^{35}.3^{25}.5^{23}.11.13^2.379^2.C_{21}$	125.93	5.30	23.76
n4c6.b5	6, 51813x20058, 15228	96	44	X^{104}	2421	2^{17}	$-23.2.52009.C_3$	1161.39	35.35	32.85
n4c6.b12	13, 25605x69235, 20165	117	358	X^{827}	157050	2^{23}	$-2^{243}.3^{43}.5^{36}.7^7.11^4.13^2.31^2.C_{59+}$	66344.40	1181.61	**56.15**
n4c6.b13	14, 6300x25605, 5440	98	37	X^{87}	1709	2^{17}	$-2^{31}.3^8.5^8.7^2.11^2.13.31^2.C_{18}$	249.28	9.90	25.18

TABLEAU 9.1 – Calcul du polynôme minimal entier de AA^t

rapides. Nous indiquons la borne de Cassini, CB, le nombre de nombres premiers nécessaires pour reconstruire la valence, Rem, le degré du polynôme minimal, d_m, la borne supérieure sur le nombre de trompeurs, bad, une borne supérieure pour les nombres premiers choisis, M, et la valence minimale, v_m, indiquant le nombre exact de trompeurs. Comme certaines v_m comportent quelques dizaines de facteurs, nous n'écrivons que les plus petits nombres premiers et notons C_i un produit de i autres facteurs premiers. Dans un cas, C_{59+} définit un produit d'au moins 59 nombres premiers. En effet, ce nombre a 57 facteurs premiers connus et un autre facteur composé de 376 chiffres que nous avons été incapables de factoriser. Nous rappelons qu'il y a respectivement $3030, 8746, 19510, 39915$ nombres premiers entre 2^{15} et 2^{16}, 2^{15} et 2^{17}, 2^{15} et 2^{18}, 2^{15} et 2^{19}, et 560821 nombres premiers entre 2^{15} et 2^{23}. Ces derniers résultats impliquent les valeurs de M.

La meilleure accélération obtenue est de 56,1. En fait, même si il y a peu de tâches, ces accélérations paraissent un peu faibles, par rapport à une accélération optimale de 80. Une explication est un surcoût dû à la scrutation du réseau par un thread spécialisé dans le noyau d'exécution Athapascan-0. En effet, un processus par nœud s'occupe de scruter le réseau pour effectuer d'éventuelles communications. Il est alors possible d'augmenter le nombre de processus pour diminuer la part de celui spécialisé dans la scrutation. Toutefois, cette augmentation induit un surcoût de gestion d'un plus grand nombre de processus. Les résultats du tableau 9.1, page 178 ont été obtenus avec le meilleur compromis, de 7 processus sur 4 processeurs, les processeurs ne pouvant alors exploiter qu'un maximum de 86% des ressources pour le calcul, situant, dans ce cas, l'accélération théorique maximale à 68,6. Cela rend plus acceptable notre accélération de 56,1. Notre application, avec un faible volume de communication, n'a de fait pas besoin d'une scrutation constante du réseau. Différents moyens pour réduire ce surcoût de scrutation ont été étudiés à la suite de ce travail [29 - Carissimi (1999), §5.4] et seront bientôt disponibles dans Athapascan-0 (scrutation à la demande, scrutation périodique, ...). Ils doivent être expérimentés pour ce calcul et permettront sans doute d'encore améliorer les accélérations.

9.7 CONCLUSIONS

En conclusion, nous avons développé un algorithme probabiliste fortement parallèle pour calculer les coefficients du polynôme minimal d'une matrice à coefficients entiers. Par l'utilisation des ovales de Cassini et de l'algorithme de Wiedemann, nous pouvons utiliser un nombre de calculs modulaires proche de l'optimal

tout en garantissant la probabilité de réussite de l'algorithme.

En outre, même sur réseau hétérogène, le calcul parallèle du polynôme minimal entier a permis un gain de temps appréciable (d'une journée à 20 minutes dans le meilleur cas) sur petites tout autant que sur grandes matrices. Il a en outre mis en évidence un problème de scrutation dans le noyau exécutif tout en validant le support applicatif. Il reste à coupler la parallélisation de l'algorithme de Wiedemann avec celle-ci pour tirer le meilleur parti possible des deux méthodes suivant les tailles de matrice et l'architecture disponible. Une extension sera alors d'implémenter un choix automatique des ressources et des algorithmes par rapport à la taille des problèmes.

10

VALENCE

« El tiempo es el mejor antologista, o el unico, tal vez... »

Jorge Luis Borgès

Sommaire

Notre nouvel algorithme probabiliste ramène le calcul de la forme de Smith à des calculs modulo des puissances de petits nombres premiers. En conséquence, l'algorithme ne souffre pas de la croissance des coefficients. En outre, les calculs modulaires sont indépendants les uns des autres, permettant une paraléllisation aisée et effective. Selon les besoins et les ressources en temps et espace, nous pouvons choisir des méthodes itératives ou directes pour certaines étapes de notre algorithme. Les idées majeures de notre approche sont que (1) nous utilisons le rang modulo des puissances de nombres premiers pour déterminer les invariants de Smith, (2) nous employons les ovales de Cassini pour obtenir des bornes fines sur les valeurs propres et (3) nous commençons par le dernier coefficient non nul (la valence) du polynôme minimal de la matrice symétrique mais non préconditionnée AA^t. Notre méthode est particulièrement efficace quand ce polynôme est de faible degré et par conséquent rapide à calculer par la méthode de Wiedemann. Ceci s'est avéré être le cas, notamment, pour toutes les matrices d'homologie et les matrices $BIBD$. Nous présentons une vue d'ensemble de notre algorithme section 10.1, page 182. Nous discutons alors quelques détails d'implémentation sections 10.3, page 185, et 10.4, page 188. Une analyse asymptotique complète est donnée section 10.5, page 190. Enfin, des expériences comparatives sont menées section 10.6, page 193.

10.1 MR SMITH GOES TO VALENCE

Dans cette section, nous décrivons notre nouvel algorithme pour le calcul de la forme de Smith d'une matrice entière. La méthode est particulièrement pertinente quand le degré du polynôme minimal de AA^T est petit. Il s'agit de calculer l'ensemble des nombres premiers apparaissant dans la forme de Smith, ou un ensemble les contenant tous, puis de calculer le rang de la matrice modulo les puissances de ces nombres premiers. Reprenons l'exemple introductif, la matrice

$$A = \begin{bmatrix} 0 & 3 & -3 & 0 & 0 \\ -1 & 0 & -2 & 2 & 1 \\ 3 & 6 & 0 & -6 & 3 \\ 0 & 6 & -6 & 0 & 6 \end{bmatrix}.$$

Les seuls nombres premiers apparaissant dans sa forme de Smith sont 2 et 3. Le rang de A modulo 7, par exemple, est 3 et c'est aussi le rang sur les entiers puisque 7 n'est pas dans la forme de Smith : A possède donc trois invariants non nuls. Le rang de A modulo 2 est 2 et le rang de A modulo 4 est 3, donc le dernier facteur invariant est un multiple de 2. Enfin, le rang de A modulo 3 est 1, et 3 modulo

9, donc les deux derniers invariants sont des multiples de 3. On retrouve bien la forme de Smith de A : $Smith(A) = diag(1, 3, 6, 0) \in \mathbb{Z}^{4\times 5}$. Nous donnons maintenant cet algorithme plus formellement.

Algorithme 10.1.1 *Forme-de-Smith-via-Valence*

Entrées : – une matrice $A \in \mathbb{Z}^{m\times n}$. A peut être une "boîte noire", c'est-à-dire que le seul pré-requis est que les produits matrice-vecteur à gauche et à droite soient calculables : $x \longrightarrow Ax$ pour $x \in \mathbb{Z}^n$ et $y \longrightarrow yA$ pour $y^T \in \mathbb{Z}^m$.

Sorties : – $S = \mathrm{diag}(s_1, \ldots, s_{\min(n,m)})$, la forme de Smith entière de A.

Calcul de la Valence

1 : **Si** $n < m$ **Alors**

2 : $B = AA^T$

3 : **Sinon**

4 : $B = A^T A$

5 : $N = \min(n, m)$

6 : $v =$ valence de B. {chapitre 9}

Factorisation

7 : L est la liste des facteurs premiers de v.
{Exceptionnellement, un facteur peut être composé, voir section 10.5, page 190}

Rang et première estimation de la forme de Smith

8 : Choisir un nombre premier p n'appartenant pas à L (i.e. $p \nmid v$).

9 : $r = \mathrm{rang}_p(A)$. { C'est aussi le rang entier. Les r premiers invariants sont non nuls.}

10 : $S = \mathrm{diag}(s_1, \ldots, s_N)$, avec $s_i = 1$ pour $i \leq r$ et $s_i = 0$ pour $i > r$.

Invariants non triviaux

11 : **Pour tout** $p \in L$ **Faire**

12 : $S_p = \mathrm{diag}(s_{p,1}, \ldots, s_{p,N})$, la forme de Smith de A dans l'anneau local $\mathbb{Z}^{(p)}$.
{ Voir section 10.4, page 188}

13 : $S = S \otimes S_p$. { i.e., $s_i = s_i s_{p,i}$ pour les $s_{p,i}$ qui sont des puissances de p.}

Renvoyer la forme de Smith

14 : Renvoyer $S = \mathrm{diag}(s_1 \ldots s_N) \in \mathbb{Z}^{m\times n}$.

Pour prouver la validité de notre méthode, nous avons besoin du théorème suivant.

Théorème 10.1.2

Soit A une matrice de $\mathbb{Z}^{m\times n}$. Soient $(s_1, \ldots, s_r)$ ses invariants non nuls. Si $p \in \mathbb{Z}$ est un nombre premier divisant un des s_i, alors p^2 divise la valence caractéristique de AA^T et p divise la valence minimale de AA^T. La même chose est vraie pour les valences de A^TA.

Preuve de 10.1.2 : Soit $B = AA^T$. L'argument s'appliquera de même à $B = A^TA$. Soit $v = \text{valence}(\pi(B)) = \text{valence}(B)$. Soit $C(x)$, le polynôme caractéristique de B, et $F_1(x), \ldots F_k(x)$, les facteurs invariants de B. Il est bien connu que ces polynômes sont unitaires, entiers et que

$$F_1(x)|F_2(x)|\ldots|(F_k(x) = \pi(B)) \text{ et } C(x) = \prod F_i(x).$$

Soit $v_c = \text{valence}(C) = \text{valence caractéristique}(B)$ et $v_i = \text{valence}(F_i)$. Il en découle aisément que $v_1|v_2|\ldots|v_k = v$, et que $v_c = \prod v_i$. Ainsi, v et v_c ont les mêmes facteurs premiers. Donc la deuxième partie du théorème se déduit de la première et il est suffisant de montrer que tout nombre premier apparaissant dans A apparaît au carré dans v_c.

En utilisant la formule de Cauchy-Binet 5.1, page 95, on remarque que v_c, comme coefficient du polynôme caractéristique, vérifie cette relation sur les mineurs de A pour certains i :

$$\pm v_c = \sum_{I\in E_i^n} B_{II} = \sum_{I\in E_i^n}\sum_{K\in E_i^n} A_{IK}A_{KI}^T = \sum_{I\in E_i^n}\sum_{K\in E_i^n} A_{IK}^2.$$

Ainsi, comme somme de carrés, un tel coefficient est non nul si et seulement si un des A_{IK} est non nul. Donc v_c est le coefficient de $C(x)$ pour lequel $i = \text{rang}(A)$. En outre, si p apparaît dans la forme de Smith, alors $p|pgcd(A_{IK})$, le $r^{\text{ième}}$ diviseur déterminantiel de A. On en conclut, $p^2|v_c$. □

Il est trivial de montrer que ce théorème est toujours valide dans n'importe quel anneau Euclidien de caractéristique zéro (en général en ajoutant la conjugaison à la transposition).

Corollaire 10.1.3

L'algorithme Forme-de-Smith-via-Valence *calcule la forme de Smith.*

Preuve de 10.1.3 : Le théorème prouve que l'on considère les bons nombres premiers. Il est en outre clair que la forme de Smith entière est composée des différentes formes de Smith dans les anneaux locaux p-adiques $\mathbb{Z}^{(p)}$, puisque la

forme de Smith *est* la forme de Smith locale pour p à des multiples près et que ceux-ci sont inversibles modulo p. □

10.2 UN POINT SUR LA SYMÉTRISATION

Le choix entre A^TA et AA^T est trivial au vu de la proposition suivante.

Proposition 10.2.1

$\pi_{A^TA}(X)$ et $\pi_{AA^T}(X)$ sont égaux ou diffèrent d'un facteur X.

Preuve de 10.2.1 : Le théorème de Cayley-Hamilton [66 - Gantmacher (1959), Théorème IV.§4.2] prouve que $\pi_{A^TA}(A^TA) = 0$. Alors, en multipliant des deux côtés par A et A^T, on obtient $A\,\pi_{A^TA}(A^TA)A^T = 0$. Or $A\,\pi_{A^TA}(A^TA)A^T = AA^T\,\pi_{A^TA}(AA^T)$, ce qui veut dire que le polynôme $X\,\pi_{A^TA}$ appliqué à AA^T donne zéro. Or, π_{AA^T} est le polynôme minimal de AA^T. Il s'en suit que π_{AA^T} doit diviser tout annulateur de AA^T et donc, en particulier, il doit diviser $X\,\pi_{A^TA}$. De la même façon, on prouve que $\pi_{A^TA}\,|X\,\pi_{AA^T}$. Il n'y a alors que trois possibilités : soit $\pi_{AA^T} = X\,\pi_{A^TA}$, soit $\pi_{AA^T} = \pi_{A^TA}$, soit $X\,\pi_{AA^T} = \pi_{A^TA}$. □

Ainsi, la différence de degré a un effet négligeable sur l'algorithme. Il est donc avantageux de choisir la plus petite matrice entre AA^T et A^TA, pour réduire le coût des produits scalaires. En outre, toute borne calculée sur les coefficients de π_{A^TA} peut s'appliquer à ceux de π_{AA^T} et vice versa.

10.3 ÉVICTION DE NOMBRES PREMIERS : MÉTHODE DU NOYAU

Nous montrons maintenant qu'il est parfois possible de restreindre la liste des nombres premiers, puisque tous les diviseurs de la valence de AA^T ne sont pas forcément dans la forme de Smith de A.

Considérons un nombre premier p apparaissant dans la forme de Smith de A. Nous savons que p^2 divise la valence caractéristique de AA^T. Sur de nombreuses

matrices, nous nous sommes aperçu qu'en général p^2 divise aussi la valence minimale. Bien sûr, il est possible de construire un contre-exemple. Par exemple, soit $A = \left[\begin{smallmatrix} 1 & 1 \\ -1 & 1 \end{smallmatrix}\right]$; elle a pour forme de Smith $\left[\begin{smallmatrix} 1 & 0 \\ 0 & 2 \end{smallmatrix}\right]$. Pourtant, $AA^T = A^TA = \left[\begin{smallmatrix} 2 & 0 \\ 0 & 2 \end{smallmatrix}\right]$ et a pour polynôme minimal $x - 2$ et pour polynôme caractéristique $(x-2)^2$. Toutefois, pour toutes les matrices d'homologie que nous avons examinées, tous les nombres premiers de la forme de Smith de A apparaissent au carré dans la valence minimale de AA^T et A^TA.

Nous voulons donc, après avoir calculé la valence, déterminer le plus rapidement possible si un nombre premier apparaissant sans carré dans la valence est dans la forme de Smith. C'est-à-dire que nous voulons savoir si le rang modulo ce nombre premier est le même que le rang entier. Nous proposons, dans cette section, un algorithme dont le temps de calcul est simplement fonction du degré du polynôme minimal de $B = AA^T$. Cependant, cette méthode nécessite une arithmétique entière en précision arbitraire, au contraire de l'approche par le calcul du rang modulo p. Malgré tout, quand ce rang est grand par rapport au degré, cette méthode peut s'avérer plus rapide.

L'idée est d'utiliser un facteur R de $M = \pi(B)$ tel que $M = RN$. Nous voulons savoir si ce facteur est répété dans la forme de Frobenius de A, i.e. si la dimension du noyau de $R(B)$ est le degré de R ou un multiple de ce degré. Le prochain lemme prouve que c'est le cas quand R et N sont premiers entre eux, puisque la dimension de l'image de $N(B)$ est égale à la dimension du noyau de $R(B)$. Cela conduit à un algorithme probabiliste : si $d = \deg(R)$, nous essayons $d+1$ vecteurs u_i choisis au hasard et nous regardons si les $v_i = N(B)u_i$ sont dépendants. Dans ce cas, nous savons, avec une forte probabilité, que la dimension du noyau de $R(B)$ est d et que donc R ne peut pas être répété. Alors, tout nombre premier apparaissant sans carré dans la valence de R et dans la valence de M apparaîtra aussi sans carré dans la valence caractéristique de AA^T. Un tel nombre premier ne peut pas être dans la forme de Smith de A, nous pouvons donc l'écarter.

Nous donnons maintenant l'algorithme, puis le lemme des dimensions et terminons cette section par l'analyse de probabilités.

Algorithme 10.3.1 *Dimension-du-Noyau*

Entrées : – une matrice $A \in \mathbb{Z}^{n\times m}$, A peut être une boîte noire.
– le polynôme minimal M de AA^T, et un facteur R de M.
– Une tolérance $\epsilon \in \mathbb{R}$, telle que $0 < \epsilon < 1$.

Sorties : – une liste d'éviction $\overline{L}$ des nombres premiers, divisant la valence de R, mais n'apparaissent pas dans la forme de Smith de A. La liste est correcte avec une probabilité au moins $1 - \epsilon$.

Initialisations

1 : $\overline{L} = \emptyset$

2 : $d = \deg(R)$.

3 : $N = \frac{M}{R}$

4 : $g = \lceil \epsilon^{-\frac{1}{d}} \rceil$.

5 : p_0 un nombre premier choisi au hasard tel que $p_0 > g$ et $p_0 \nmid \text{valence}(M)$.

6 : Former la boîte noire $N(B)$.

Dimension Probabiliste du noyau

7 : Choisir $d+1$ vecteurs non nuls $u_i \in (\mathbb{Z}/p_0\mathbb{Z})^n$ au hasard.

8 : **Pour** $i = 0$ **jusqu'à** d **Faire**

9 : $\quad v_i = N(B)u_i$.

10 : **Si** $\text{rang}([v_0, v_1, \ldots, v_d]) < d+1$ modulo p_0 **Alors**

11 : $\quad$ **Pour tout** p premier et divisant valence(R) **Faire**

12 : $\quad\quad$ **Si** p^2 ne divise pas valence(M) **Alors**

13 : $\quad\quad\quad \overline{L} = \overline{L} \cup \{p\}$

Nombres premiers écartés

14 : Renvoyer $\overline{L}$.

Bien sûr, cet algorithme peut s'appliquer pour tout facteur de M pour écarter le plus possible de nombres premiers. Il faut alors factoriser le polynôme minimal sur les entiers ; cela peut être réalisé efficacement si le polynôme est petit par une factorisation modulaire suivie d'une remontée p-adique, voir [1 - Abbott et al. (2000)], par exemple, pour plus de détails.

Il nous reste à prouver la validité de notre algorithme ; nous avons besoin pour cela du lemme suivant.

Lemme 10.3.2

Soit $B \in \mathbb{Z}^{n\times n}$. Soit $N, R \in \mathbb{Z}[X]$ premiers entre eux tels que $N(B)R(B) = 0 \in \mathbb{Z}^{n\times n}$. Alors $\text{Im}(R(B)) = \ker(N(B))$ et $\text{Im}(N(B)) = \ker(R(B))$.

Preuve de 10.3.2 : D'abors, comme $R(B)N(B) = N(B)R(B) = 0$ l'image d'une des matrices polynomiales est incluse dans le noyau de l'autre. Ensuite, en utilisant [66 - Gantmacher (1959), Théorème VII.§2.1], on sait que $\ker(R(B))$ et $\ker(N(B))$ sont supplémentaires, puisque N et R sont premiers entre eux. La conclusion découle alors du théorème de la dimension : la somme des dimensions

de l'image et du noyau d'une application linéaire, $dim(\mathrm{Im}(X)) + dim(\ker(X))$, est égale à la dimension de l'espace. □

Théorème 10.3.3

L'algorithme Dimension-du-Noyau *est correct.*

Preuve de 10.3.3 : Soit $B = AA^T$. B est symétrique, donc son polynôme minimal M est sans carré. On suppose maintenant que ce polynôme n'est pas irréductible. Soient N et R deux cofacteurs de M. Comme M est sans carré, N et R sont premiers entre eux. En outre, d'après le lemme, l'image de $N(B)$ a même dimension que le noyau de $R(B)$. Donc le noyau de $R(B)$ a pour dimension kd, où k est la multiplicité de R dans le polynôme caractéristique de B. Alors, l'algorithme *Dimension-du-Noyau* ne peut donner une réponse erronée que si $d+1$ vecteurs au hasard (les u_i) sont les antécédents de $d+1$ vecteurs dépendants (les v_i) dans un espace de dimension kd, sur un corps avec plus de g éléments. Remarquons que les v_i sont distribués uniformément dans l'image de $N(B)$ si les u_i le sont dans $(\mathbb{Z}/_{p_0}\mathbb{Z})^n$.

Nous quantifions la probabilité d'une telle dépendance. Soit $P(j,n)$ la probabilité d'une dépendance parmi j vecteurs choisis au hasard dans un espace vectoriel de dimension n ($j \leq n$) sur un corps $\mathbb{Z}/_{p_0}\mathbb{Z}$ avec p_0 éléments. Alors $P(j,n) = P(j-1,n) + P($ les $j-1$ premiers sont indépendants mais le $j^{\text{ième}}$est dépendant) Cela donne :

$$P(j,n) \leq P(j-1,n) + \frac{p_0^{j-1}}{p_0^n} \leq \frac{p_0^j - 1}{(p_0-1)p_0^n} < (p_0-1)^{j-1-n}.$$

Or, comme $p_0 > g$, si $k \geq 2$ alors $P(d+1, kd) \leq g^{d-kd} \leq \frac{1}{g^d} \leq \epsilon$, comme annoncé dans l'algorithme. □

10.4 FORME DE SMITH LOCALE

La question suivante est le calcul de la forme de Smith locale dans $\mathbb{Z}^{(p)}$, l'espace p-adique. Ce calcul est équivalent au calcul du rang modulo p^k pour k suffisamment grand. Rappelons que l'on définit le rang modulo p^k comme le nombre de facteurs invariants non nuls modulo p^k (voir section 5.3, page 93).

Dans beaucoup de cas, nous réussissons à calculer ce rang grâce à l'algorithme d'élimination modulo p^k, 5.3.5, page 94, malgré le remplissage de la matrice. En raison de l'explosion de la taille des calculs intermédiaires, il n'est pas pertinent de calculer directement dans $\mathbb{Z}^{(p)}$. Aussi, nous calculons modulo p^e et déterminons ainsi une suite de rangs modulo p^k pour $k \leq e$: $(r_{p,1}; \ldots; r_{p,e})$. Si le rang modulo p^e est plus petit que r, le rang de A sur les entiers, nous pouvons répéter le calcul avec un plus grand exposant e jusqu'à ce que le rang modulo p^e égale le rang sur les entiers. Dans ce cas, les exposants de p dans la forme de Smith, c'est-à-dire les exposants de p dans les s_i, sont totalement déterminés, ils valent 1 pour i de 1 à $r_{p,1}$ inclus, puis 2 pour i de $r_{p,1}$ à $r_{p,2}$, etc.

Pour certaines matrices, cette approche par élimination échoue à cause d'une demande excessive en mémoire. Il est donc important d'avoir une méthode efficace pour ces cas. Une première idée est d'utiliser l'algorithme de Wiedemann avec un préconditionnement diagonal, pour calculer le rang modulo p (voir section 6.6, page 137). Si ce rang est égal au rang entier, nous savons que p n'apparaît pas dans la forme de Smith. Au contraire, si ce rang est plus petit, nous savons que des puissances de p apparaissent dans les derniers invariants mais nous ne connaissons pas la valeur de leurs exposants. La forme de Smith n'est alors pas complète. Il faudrait avoir un algorithme de Wiedemann modulo p^k, qui permette de calculer le rang. La méthode de Reeds et Sloan [141 - Reeds et Sloane (1985)] calcule un ensemble de e polynômes annulateurs modulo $\mathbb{Z}/p^e\mathbb{Z}$. Tout polynôme annulateur doit être formé à partir de cette "base" [128 - Norton (1999b)]. Norton et Kurakin, en particulier, ont étudié les méthodes pour calculer des polynômes minimaux dans $\mathbb{Z}/p^e\mathbb{Z}$ [99 - Kurakin (1993), 100 - Kurakin (1994), 63 - Fitzpatrick et Norton (1989), 126 - Norton (1995), 127 - Norton (1999a)] ; toutefois le calcul du rang modulo p^k à partir de cette information ne semble pas aisé. Après plusieurs expériences, il nous est apparu que la suite des degrés de ces polynômes est représentative de la forme de Smith. Ainsi, si le nombre premier p apparaît dans les k derniers facteurs invariants, à toute série de puissances de p dans la forme de Smith, il semble correspondre une série particulière des degrés des polynômes annulateurs dans $\mathbb{Z}/p^k\mathbb{Z}$. Nous n'avons pas été capable de prouver ce fait. En outre, cela induirait de calculer modulo p puissance le nombre de fois que p apparaît dans la forme de Smith et donc de stocker au moins k polynômes denses, ce qui peut dépasser les capacités mémoires.

G. Villard propose alors une alternative, la méthode p-adique pour calculer le dernier facteur invariant [54 - Dumas et al. (2000), Section 4.2.3] et obtenir ainsi toutes les puissances intervenant dans la forme de Smith. Informellement, il s'agit de calculer une solution rationnelle du système linéaire ; avec une forte probabilité, le plus grand dénominateur de cette solution *sera* le dernier facteur invariant.

Cela complète l'algorithme d'un point de vue théorique, mais, en pratique, cette méthode ne fonctionne pas avec des préconditionnements diagonaux mais nécessite des préconditionnements coûteux (souvent multipliant par 100 le temps de calcul d'un produit matrice-vecteur).

Un axe de recherche actuellement important est de trouver des préconditionnements moins coûteux.

10.5 ANALYSE ASYMPTOTIQUE

Dans cette section, nous résumons les complexités binaires et spatiale ainsi que les probabilités de réussite des différentes parties de notre algorithme. Nous rappelons que pour une matrice A, $\Omega = max(n, m,$ nombre d'éléments non nuls de $A)$, r est le rang sur les entiers, $d = d_{AA^T}$ est le degré du polynôme minimal de AA^T, $s \leq d \ln(\beta)$ est le nombre de nombres premiers divisant la valence de ce polynôme, π_{AA^T}. La figure 10.1, page 191 rappelle l'organisation générale de l'algorithme.

Dans la table 10.1, page 192, nous supposons que la valence est de petite taille, comme c'est le cas pour les matrices pour lesquelles l'algorithme est efficace. Nous négligeons ainsi deux conséquences d'une valence de grande taille.

(1) Même après l'utilisation de la méthode du noyau (voir section 10.3, page 185), il peut subsister de grands nombres premiers potentiels, de taille $\mathcal{O}(s)$. Le coût d'une opération arithmétique n'est alors plus unitaire mais en $I(s) = \widetilde{\mathcal{O}}(s)$. Dans la table, les coûts de calcul du rang et des invariants sont alors multipliés par $I(s)$ de même que la complexité globale.

(2) Nous ne montrons pas, dans cette table, les complexités pour la factorisation : en fait, la factorisation *totale* de la valence n'est pas nécessaire. En effet, il est possible de commencer les calculs de rang avec un facteur composé. Ensuite, dans les deux cas, élimination et boîte noire, les problèmes résultant de la non primalité révèlent les facteurs premiers de ce nombre ! Tout d'abord, dans le cas de l'élimination, les problèmes surgissent quand un pivot non nul et non inversible est trouvé. Le pgcd de ce pivot et de la valence indique alors deux facteurs non triviaux. En outre, l'élimination peut reprendre modulo ces deux facteurs. De même, dans le deuxième cas, la méthode de Wiedemann, les problèmes surgissent quand un produit scalaire non nul et non inversible est calculé. Là encore, le pgcd indique deux facteurs non triviaux et les calculs peuvent reprendre modulo ces

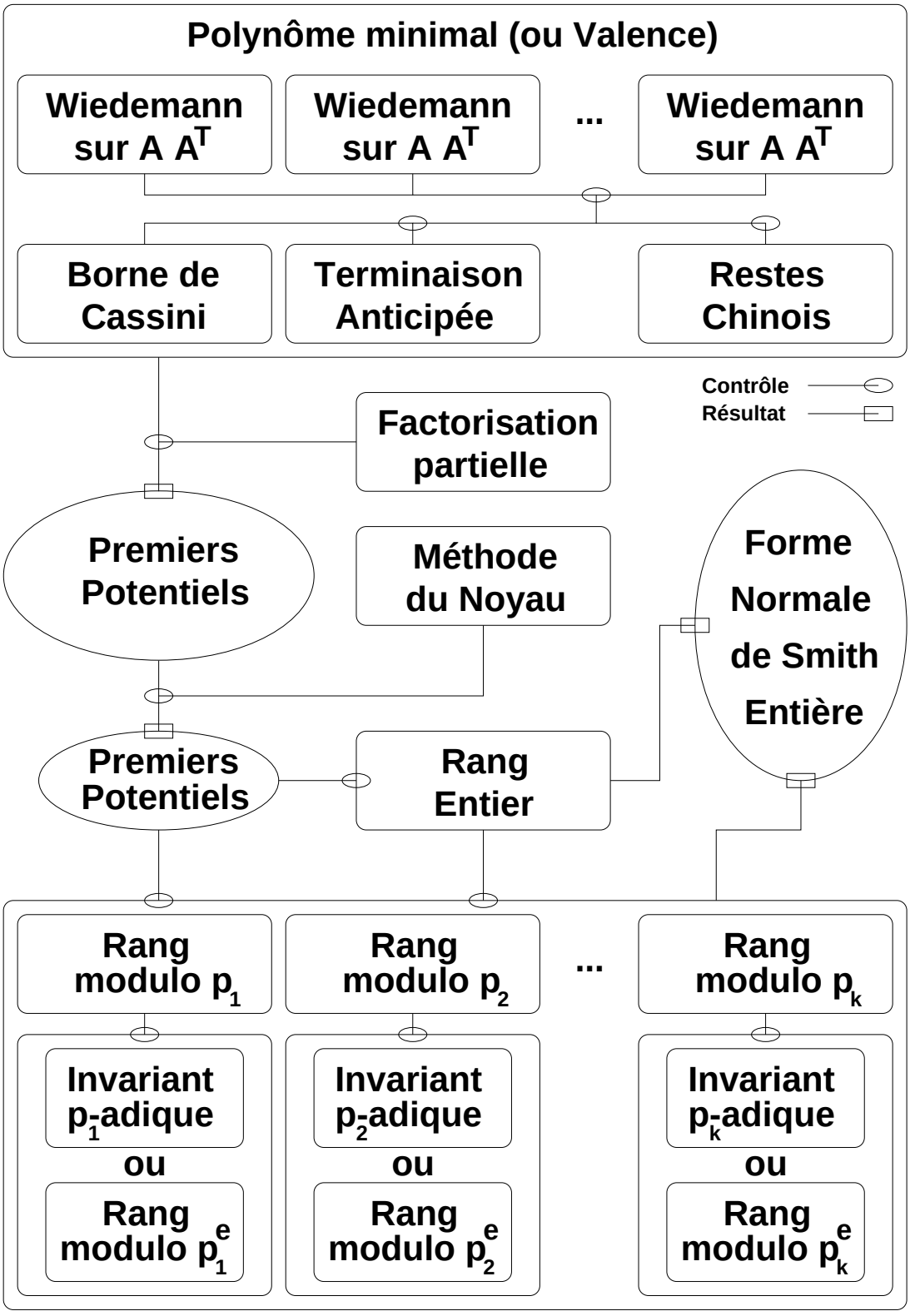

FIGURE 10.1 – Algorithme Valence

Étape	Temps	Mémoire	Réussite	Section
Wiedemann	$\tilde{\mathcal{O}}(d\Omega \log(\epsilon^{-1}))$	$\tilde{\mathcal{O}}(n)$	$1-\epsilon$	6.3
borne de Cassini	$\mathcal{O}(\Omega)$	$\mathcal{O}(n)$	1	9.3
Restes Chinois	$\mathcal{O}(s)$	$\tilde{\mathcal{O}}(n)$	1	9.5
Total				
Valence	$\tilde{\mathcal{O}}(d\Omega \log(\epsilon^{-1}))$	$\tilde{\mathcal{O}}(n)$	$1-\epsilon$	9.5
Élimination				
Rang entier	$\mathcal{O}(rmn)$	$\mathcal{O}(n^2)$	$1-\epsilon$	5.1
Rang premier	$\tilde{\mathcal{O}}(rmn)$	$\tilde{\mathcal{O}}(n^2)$	1	5.1
Rang puissances	$\tilde{\mathcal{O}}(rmn)$	$\tilde{\mathcal{O}}(n^2)$	1	5.3.2
Boîtes noires				
Rang entier	$\tilde{\mathcal{O}}(r\Omega \log(\epsilon^{-1}))$	$\tilde{\mathcal{O}}(n)$	$1-\epsilon$	6.6
Rang premier	$\tilde{\mathcal{O}}(r\Omega \log(\epsilon^{-1}))$	$\tilde{\mathcal{O}}(n)$	$1-\epsilon$	6.6
Invariant	$\tilde{\mathcal{O}}(r\Omega \log(\epsilon^{-1}))$	$\tilde{\mathcal{O}}(n)$	$1-\epsilon$	10.4
Algorithme complet avec				
Élimination	$\tilde{\mathcal{O}}(srmn + d\Omega \log(\epsilon^{-1}))$	$\tilde{\mathcal{O}}(n^2)$	$1-\epsilon$	
Boîtes noires	$\tilde{\mathcal{O}}(sr\Omega \log(\epsilon^{-1}))$	$\tilde{\mathcal{O}}(n)$	$1-\epsilon$	

TABLEAU 10.1 – Complexités arithmétiques asymptotiques de l'algorithme Valence

deux facteurs.

De plus, en général, pour les matrices d'homologie, la valence est facile à factoriser (voir le tableau 9.1, page 178), en utilisant les courbes elliptiques, par exemple. Nous factorisons autant que possible, ce qui signifie complètement pour la plupart des matrices, pour isoler les petits nombres premiers. Il est conjecturé que l'algorithme de factorisation par courbes elliptiques détermine un diviseur non trivial d'un nombre composé t en un temps moyen

$$\ln(t)^2 e^{\sqrt{\ln(p)\ln\ln(p)(2+o(1))}}$$

où p est le plus petit nombre premier divisant t [106 - Lenstra (1987)]. En outre, comme nous le voyons dans le tableau 9.1, page 178, pour les matrices que nous avons considérées, s est habituellement très petit avec de très petits diviseurs premiers, permettant une efficacité pratique de l'algorithme.

10.6 EXPÉRIENCES AVEC MATRICES D'HOMOLOGIE

Les matrices d'homologie sont décrites section 7.1.2, page 146. Nous vérifions dans le tableau 9.1, page 178 que leurs Laplaciens (AA^T et A^TA) ont bien un polynôme minimal de très petit degré (jusqu'à 25 pour les couplages et les jeux d'échec, 200 pour les non-connectés).

Dans le tableau 10.2, page 193, nous comparons les temps d'exécution de notre algorithme et d'autres méthodes. Nous comparons les résultats obtenus avec la version séquentielle de l'algorithme Valence où nous utilisons l'algorithme de Wiedemann pour calculer la valence *puis des éliminations* modulo de petits nombres premiers p pour calculer les facteurs invariants localement à p. Simplicial Homology [53 - Dumas et al. (2000)] est un module proposé pour le logiciel GAP [67 - GAP (1999)]. Il calcule les groupes d'homologie de complexes simpliciaux via la forme de Smith de leurs matrices des limites. Il comporte une version séquentielle de notre algorithme ainsi qu'une méthode par élimination spécialisée pour ces matrices écrite par Frank Heckenbach. Cette dernière est une variante de la méthode classique en précision arbitraire, mais comme elle tire parti de la structure particulière des matrices d'homologie, elle est souvent la plus efficace pour ces matrices. L'entrée "Hom-Elim-GMP" dans cette table se rapporte à cette méthode implémentée avec les entiers de la bibliothèque "Gnu Multi Precision". Fermat [107 - Lewis (1997)] est un système de calcul formel pour Mac et Windows. Sa routine pour la forme de Smith est une implémentation de l'algorithme de Kannan & Bachem [7 - Bachem et Kannan (1979)]. "Hom-Elim-GMP"

Matrice	Fermat	Hom-Elim-GMP	Valence (Éliminations)
ch6-6.b4	49.4	2.15	27.42 (6)
mk9.b3	2.03	0.21	0.95 (4)
mk10.b3	8.4	0.94	13.97 (7)
mk11.b4	98937.27	2789.71	384.51 (7)
mk12.b3	189.9	26.11	304.22 (7)
mk12.b4	Dépassement	Dépassement	13161.01 (7)

TABLEAU 10.2 – Fermat vs. Hom-Elim-GMP vs. Valence-Elim

et "Valence-Elim" ont été exécutées sur un processeur sparc SUNW, Ultra-4 à 400 MHz et 512 Mo, mais Fermat n'est disponible que sur Mac ou Windows. Nous présentons ici des expériences avec Fermat sur un processeur Intel i860 à 400 MHz et 512 Mo. Nous voyons tout d'abord que "Fermat" ne peut jamais rivaliser

avec “Hom-Elim-GMP”. L’explication principale est que la stratégie de pivot utilisée par “Hom-Elim-GMP” est spécialisée pour les matrices d’homologie. Nous pouvons voir également que, aussi longtemps que les coefficients restent petits, “Hom-Elim-GMP” est souvent meilleure que “Valence-Elim”. En effet, quand “Hom-Elim-GMP” ne fait qu’une seule élimination, “Valence-Elim” effectue une élimination pour *chaque* nombre premier (le nombre de ces éliminations est indiqué entre parenthèses dans la colonne Valence (Eliminations) de la table) - bien sûr, en parallèle, cette différence s’estompe. Enfin, dès que la taille des coefficients devient importante, comme par exemple pour les matrices mk11.b4 et mk12.b4, “Valence-Elim” devient meilleure.

D’autre part, la version de l’algorithme “Valence” n’utilisant que les méthodes itératives peut donner des résultats partiels là où des dépassements de capacité mémoire empêchent les méthodes par élimination de terminer. Dans la table 7.3, page 153 nous avons déjà vu de tels effets pour le calcul du rang. Nous présentons certains résultats partiels dans la table 10.3, page 194 : pour certaines matrices, nous sommes capables de calculer le rang modulo des nombres premiers et, ainsi, l’occurrence de ces nombres premiers dans la forme de Smith, mais pas leur exposant. Les temps indiqués sont pour le calcul séquentiel. Ces informations sont toutefois les seuls résultats connus actuellement pour ces matrices.

Matrice	Temps			Forme de Smith
mk13.b5	98 heures	Partielle	:	133991 uns & 220 puissances de 3
m133.b3	201 heures	Partielle	:	168309 uns & 1 puissance de 2
ch7-8.b5	180 heures	Partielle	:	92916 uns & 35 puissances de 3 & 8 puissances de 2 et 3
ch8-8.b4	264 heures	Complète	:	100289 uns

TABLEAU 10.3 – Forme de Smith via Valence avec techniques itératives

Les précédentes comparaisons des méthodes d’élimination et de la méthode par la valence fournissent une base pour des remarques récapitulatives :

(1) L’élimination peut être très efficace sur ces matrices creuses et structurées. Cependant cela n’est vrai que tant que la stratégie de pivot est spécialisée et/ou qu’une renumérotation est mise en œuvre.

(2) Pour des matrices suffisamment grandes, le remplissage ajouté au grossissement des coefficients rend l’élimination plus lente que notre méthode. Pour les plus grandes matrices, l’élimination échoue même, à cause d’une trop grande demande en mémoire. Avec l’approche par la valence, combinée à l’utilisation

de méthodes itératives, nous sommes capables de calculer le rang modulo des nombres premiers, et donc la forme de Smith, de matrices de 500000 lignes, alors que l'élimination échoue quand le nombre de lignes est aux alentours de 50000.

(3) Un problème ouvert est de déterminer efficacement le rang modulo des puissances de nombres premiers (> 1), tout en utilisant des méthodes itératives peu gourmandes en mémoire.

Quatrième partie

Implémentation

11

STRUCTURES DE DONNÉES

« I would like to be the air that inhabits you for a moment only. I would like to be that unnoticed and that necessary. »

Margaret Atwood

Sommaire

Le problème majeur après la définition de l'algorithmique est la définition des structures de données les plus adaptées. Il s'agit de faire correspondre des objets mathématiques à des représentations informatiques dans un langage de programmation. En particulier, les ensembles mathématiques définissent des opérations sur leurs éléments alors qu'en informatique, les opérations sont construites en premier lieu (procédures, méthodes) et définissent ainsi leur ensemble (leurs paramètres admissibles). À ces problèmes s'ajoutent les différences de performances liées au matériel. Dans la première partie, par exemple, nous avons vu l'influence de la mémoire cache sur l'arithmétique.

Nous nous intéressons tout d'abord, dans ce chapitre, à l'implémentation pratique des ensembles mathématiques et principalement à celle des entiers modulaires. Nous étudions ensuite différents formats de matrices creuses afin de déterminer le plus efficace. Enfin, nous montrons l'utilité du concept de boîte noire, combiné à une utilisation de la généricité et la spécialisation offerte par les « template » C++, pour la réutilisation efficace de code.

11.1 DOMAINES

Le problème est donc d'associer un ensemble (par exemple un sous-groupe de $\mathbb{Z}$) et ses éléments en tant que concept mathématique à des concepts informatiques (type, classe, objet, instance). Cette association doit être principalement guidée par le besoin de performance.

Nous commençons par étudier le cas d'un sous-groupe de $\mathbb{Z}$, $\mathbb{Z}/_{m}\mathbb{Z}$. Il s'agit de manipuler des éléments de ces groupes, les entiers modulaires. Lorsqu'une opération doit être effectuée sur ces éléments, le modulo doit être connu. La représentation d'un élément, par exemple un entier de type `int`, doit alors avoir une connaissance de l'ensemble sous-jacent, le groupe des entiers modulo m.

Une première approche, pour implémenter les opérations, consiste à stocker le modulo avec chaque élément. Évidemment, l'espace mémoire nécessaire est alors doublé. Pour réduire cet espace, le modulo peut être déclaré comme une variable globale (membre `static` en C++). Dans ce cas, le modulo est unique dans un espace de noms donné et nécessite une gestion lourde en parallèle.

Une réponse consiste à spécialiser *chacune* des structures de données (matrice creuse, matrice dense, polynômes creux et denses, vecteurs, etc.) en leur ajoutant le modulo. La généricité est alors perdue.

Une autre solution a donc été privilégiée dans Linbox. Il s'agit de considérer un sous-groupe comme un véritable objet C++, contenant le modulo. Les éléments

de ce groupe ont une certaine représentation machine qui n'est pas connue de l'utilisateur. Celui-ci doit passer par l'objet groupe pour effectuer toutes les opérations sur les éléments. Par exemple, les éléments peuvent toujours être représentés par des entiers machine, mais les opérations arithmétiques sur ces entiers sont interdites (en tout cas, elles ne sont pas définies a priori, du point de vue de l'utilisateur). Ainsi, même dans un contexte informatique, c'est l'ensemble qui définit ses éléments et leurs opérations. Nous nous rapprochons du modèle mathématique. Ces définitions sont éclaircies sur un exemple :

Supposons que l'on veuille additionner deux éléments de $\mathbb{Z}/_{7}\mathbb{Z}$. Il faut tout d'abord créer un objet représentant ce sous-groupe et contenant le modulo (il sera d'une certaine classe C++ permettant de construire de tels groupes). Il faut ensuite créer des éléments de ce groupe : la classe de l'objet groupe nous donne un type C++ pour pouvoir les déclarer, ainsi, les éléments sont communs à tous les groupes issus d'une même classe. Nous différons sur ce point du modèle mathématique, mais les conséquences n'en sont pas fondamentales. Puis ces éléments doivent être initialisés : il s'agit de calculer le reste modulo m, et seul l'objet groupe peut s'en charger. Enfin, le groupe effectue l'addition sur ses éléments :

```
Zpz F7(7); // 7 est stocké
Zpz::element a, b, c;
F7.init(a,10); F7.init(b,-2);  // a reçoit 3 et b, 5
F7.add(c, a, b); // c reçoit 1 = (3 + 5) % 7
```

Dans un premier temps, cette utilisation peut sembler déconcertante, mais les mêmes fonctionnalités sont fournies par la syntaxe de fonctions membres que par la syntaxe d'opérateurs (du type a = b+c) ; souvent, cette syntaxe peut même clarifier les opérations effectuées, en particulier l'utilisation ou non de variables temporaires. Par exemple, un suffixe `in` est ajouté aux noms de fonctions pour spécifier qu'une opération s'effectue en place :

```
F7.addin(a, b); // équivalent à  a += b
```

Les avantages de cette démarche sont multiples. Tout d'abord, nous limitons l'utilisation mémoire tout en conservant la généricité et l'efficacité, par des spécialisations. En effet, le sous-groupe peut alors implémenter de manière plus efficace les opérations sur des conteneurs ou des itérateurs ; par exemple, il peut implémenter le produit scalaire dans $\mathbb{Z}/_{7}\mathbb{Z}$. L'objet, connaissant le modulo et la représentation des éléments sur 32 bits, peut ainsi ne faire qu'une division toutes les $\frac{32}{\log_2 7}$ multiplications ! Ensuite, nous nous rapprochons des concepts mathématiques et simplifions ainsi la hiérarchie des différentes entités que nous manipulons. Enfin, l'extension de cette démarche à toutes les structures de données de Linbox en induit une utilisation uniforme par tous les algorithmes et ainsi une généricité intrinsèque.

Un objet d'une classe C++ proposant les opérations arithmétiques sous cette forme est appelé un *domaine*. Dans Linbox, l'utilisation des domaines est générale, du corps fini à l'ensemble des matrices, en passant par les espaces de polynômes.

11.2 MATRICES ET VECTEURS CREUX

Dans un deuxième temps, nous nous intéressons au format de stockage des matrices creuses non structurées. Ce format doit d'une part minimiser l'espace mémoire nécessaire au stockage de la matrice et d'autre part maximiser la vitesse de calcul des opérations sur la matrice. Les opérations que nous faisons sur les matrices creuses sont de deux types : des produits matrice-vecteur pour les méthodes itératives et des éliminations pour les méthodes directes.

11.2.1 ELLPACK-ITPACK ET LIGNE-COMPRESSÉ

D'après [98 - Kumar et al. (1994), Section 11.1], le format le plus rapide pour calculer des produits matrice-vecteur avec une matrice creuse non structurée est le format *Ellpack-Itpack* : pour une matrice $m \times n$ avec un maximum de χ éléments non nuls par ligne, celui-ci consiste en deux matrices de taille $m \times \chi$, l'une contenant les valeurs des éléments, VAL, l'autre contenant les indices de colonne des éléments, J ; chaque ligne de ces deux matrices correspond à une ligne de la matrice creuse.

$$\underset{\text{A}}{\begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 3 & 0 & 4 & 5 & 6 \\ 0 & 7 & 0 & 0 & 0 \\ 0 & 8 & 0 & 9 & 10 \end{bmatrix}} \longrightarrow \underset{\text{VAL}}{\begin{bmatrix} 1 & 2 & - & - \\ 3 & 4 & 5 & 6 \\ 7 & - & - & - \\ 8 & 9 & 10 & - \end{bmatrix}} \quad \underset{\text{J}}{\begin{bmatrix} 0 & 1 & -1 & - \\ 0 & 2 & 3 & 4 \\ 1 & -1 & - & - \\ 1 & 3 & 4 & -1 \end{bmatrix}}$$

FIGURE 11.1 – Une matrice creuse stockée au format Ellpack-Itpack

La figure 11.1, page 202 montre ce format sur une matrice A. Toutes les lignes de VAL et J ayant moins de χ éléments non nuls ont donc des espaces vides ; ces espaces vides stockent une valeur *sentinelle*, ici -1, qui annonce la fin de la ligne.

Le défaut majeur de ce format est que si les lignes sont déséquilibrées, de nombreux espaces inutiles sont stockés dans les tableaux. Nous avons donc préféré un autre format classique, le format ligne-compressé, qui ne laisse pas d'espace inutilisé dans ses tableaux. Si Ω est le nombre d'éléments non nuls de la matrice, ce format nécessite deux tableaux de taille $\Omega \times 1$, VAL et J, et un tableau de taille m, I. Les éléments (respectivement les indices de colonne) sont stockés à la suite dans VAL (respectivement J). L'élément i du tableau I pointe sur le premier élément de la ligne i dans VAL et J. La figure 11.2, page 203 montre ce format sur A.

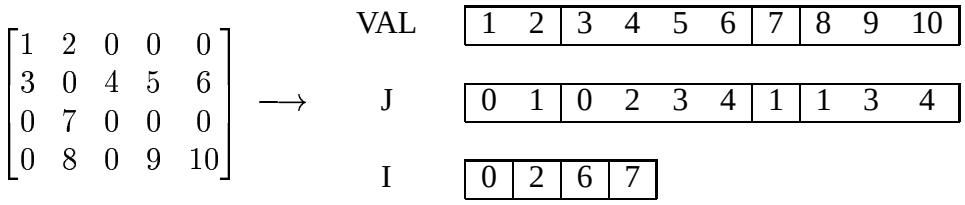

FIGURE 11.2 – Une matrice creuse stockée au format ligne-compressé

Ce format a, en outre, l'avantage de stocker les éléments dans des espaces mémoire contigus et, ainsi, d'accélérer les accès mémoire, puisque plusieurs éléments contigus sont souvent rapatriés en même temps. Quant à la programmation du produit matrice-vecteur avec ce format, elle est assez simple : il suffit de parcourir *une* fois les tableaux VAL et J guidé en cela par les valeurs de I. Le code 11.2.1 est un exemple d'implémentation.

Code 11.2.1 Produit matrice vecteur au format ligne-compressé

```
1   for (i = 0; i < I.size(); ++i) {
      jmax = (I[i+1] - I[i]) ;
      if (jmax > 0) {
        const T *pt_val = & ( VAL[ I[i] ]) ;
5       const int *pt_ind = & ( J[ I[i] ] ) ;
        T auxt;
        _domain.mul(auxt,
                    Vecteur_in[ *(pt_colind) ],
                    *(pt_val) ) ;
10       for (; --jmax ;)
           _domain.axpyin(auxt,
                          Vecteur_in[*(++pt_colind)],
                          *(++pt_val)) ;
         Vecteur_out[i] = auxt;
15     }
    }
```

11.2.2 FORMAT HYBRIDE

Dans le cas de l'élimination, le format ligne-compressé est difficile à manipuler lors des éliminations par lignes, par exemple, du fait des permutations de colonnes et des insertions de nouveaux éléments non nuls. Nous avons donc choisi

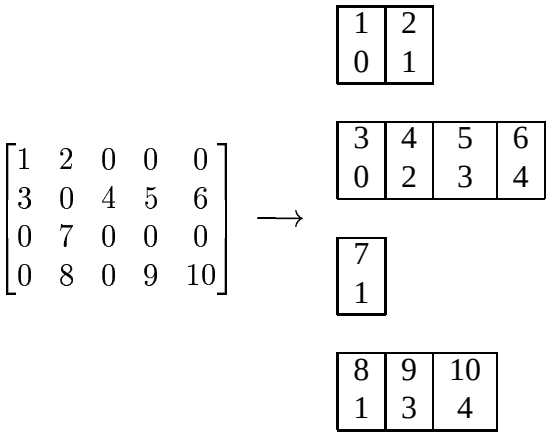

FIGURE 11.3 – Une matrice creuse stockée au format hybride

un format inspiré du format Ellpack-Itpack et du format ligne-compressé. Il s'agit de m vecteurs, un par ligne de A, chacun de la taille du nombre d'éléments non nuls dans sa ligne associée. Chaque élément de ces vecteurs est une paire (valeur, indice de colonne), ces éléments étant ordonnés par indices de colonne croissants. La figure 11.3 montre ce format pour A.

Ainsi, la permutation est aisée et les éléments restent contigus par ligne. L'algorithme d'élimination est notablement favorisé par ce format, par rapport aux deux précédents.

Nous comparons alors ce format avec le ligne-compressé, dans le cas du produit matrice-vecteur. Comme implémentation du format ligne-compressé, nous utilisons celle de SparseLib++ [137 - Pozo et al. (1996)]. Le produit est implémenté comme présenté dans le code 11.2.1, page 203. Pour le format hybride, l'implémentation du produit matrice-vecteur est similaire. Pour ce type d'opérations, la différence de performances entre les formats est en faveur du ligne-compressé pour les matrices assez denses. Nous pouvons le voir pour le cas de l'algorithme de Wiedemann scalaire dans le tableau 11.1, page 205. Toutefois, nous voyons sur ce tableau que cette différence reste relativement faible (elle s'inverse même pour certaines matrices très creuses) et donc, si un format unique

Matrice	$\Omega, n \times m, r$	ligne-compressé	hybride
robot24_m5	15118, 404x302, 262	1.52	1.84
rkat7_m5	38114, 694x738, 611	8.72	10.51
f855_m9	171214, 2456x2511, 2331	176.61	202.17
cyclic8_m11	2462970, 4562x5761, 3903	4138.77	4834.41
bibd_22_8	8953560, 231x319770, 231	995.41	1118.38
n4c5.b6	33145, 4735x4340, 2474	47.17	51.75
n2c6.b7	31920, 3990x5715, 2772	49.46	57.10
n2c6.b6	40005, 5715x4945, 2943	64.66	72.96
n4c6.b13	88200, 6300x25605, 5440	351.42	288.57
n4c6.b12	1721226, 25605x69235, 20165	4546.71	4131.06
mk9.b3	3780, 945x1260, 875	2.36	2.11
ch7-7.b6	35280, 5040x35280, 5040	228.50	119.53
ch7-6.b4	75600, 15120x12600, 8989	412.42	416.97
ch7-7.b5	211680, 35280x52920, 29448	4141.32	4283.4

TABLEAU 11.1 – Temps comparés, en secondes, de l'algorithme de Wiedemann scalaire pour différents formats creux

entre produits matrice-vecteur et élimination est nécessaire, le format hybride est le meilleur compromis.

11.3 BOÎTES NOIRES

Le concept de boîte noire [84 - Kaltofen (2000), Problème 3] est utilisé dans le cas de structures creuses. Habituellement, les algorithmes supposent qu'il existe des méthodes permettant d'accéder aux éléments de la matrice (avec $A[i][j]$, par exemple) ; dans le cas d'une boîte noire, l'algorithme suppose qu'il ne dispose pas de l'accès aux éléments (ou que celui-ci est très coûteux), mais seulement d'une fonction calculant le produit matrice-vecteur, par exemple. Pour une matrice boîte noire, ce sera l'existence d'une fonction, nommée $Apply$, effectuant le produit de la matrice par un vecteur ($v \leftarrow Au$, par l'appel $A.Apply(v, u)$).

Ainsi, pour des matrices très spéciales, le stockage est simplifié par le fait que seule cette fonction a besoin d'être implémentée. Supposons que l'on veuille construire une matrice comme le produit de deux autres matrices, mais que la seule chose dont on ait besoin par la suite soit le produit de cette matrice par des vecteurs. Au lieu d'effectuer le produit réel de ces deux matrices (en général

coûteux et souvent plus rempli que les deux matrices), il suffit de construire une fonction $Apply$, à l'aide des fonctions $Apply$ des deux matrices ; c'est-à-dire que le produit matrice-vecteur $(AB)v$ est réalisé par $A(Bv)$. Cela est illustré par le code suivant :

Code 11.3.1 Composition de boîtes noires

```
template <class BBA, class BBB,
class TmpVect = GoodChoice< BBA::InVector,
                            BBB::OutVector >::Vector >
class BB_Composition {
private:   // Composition is  BBA x BBB
  BBA   * _amat;
  BBB   * _bmat;
  TmpVect _inter;

public:

  BB_Composition(BBA* a, BBB* b)
      : _amat(a), _bmat(b), _inter(0) {}

  typedef BBB::InVector   InVector;
  typedef BBA::OutVector OutVector;
  typedef TmpVect   InternalVector;


  long n_row() const { return _amat->n_row();}
  long n_col() const { return _bmat->n_col();}

  template<class OutVect, class InVect>
  OutVect& Apply(OutVect& outM,  const InVect& inM ) {
    _inter.resize (_bmat->n_row());
    return _amat->Apply (outM,
                         _bmat->Apply ( _inter, inM) );
  }
};
```

Ce concept est utilisé section 6.6, page 137 pour calculer le polynôme minimal de AA^T. En sus d'éviter le produit de matrices, la matrice produit de A par sa transposée contenant souvent au moins deux fois plus d'éléments non nuls que A, le coût de calcul d'un produit matrice-vecteur est réduit d'autant ! Par exemple, la matrice ch4-4.b2 de la figure 7.1, page 147 contient 288 éléments non nuls,

alors que la matrice produit AA^T contient 648 éléments non nuls et le produit A^TA, en possède 960. Le coût d'un produit matrice-vecteur par AA^T est donc de 648 additions et multiplications lorsque le produit des matrices est effectué et de seulement 576 additions et multiplications si l'on utilise la composition des boîtes noires. Un exemple plus conséquent est la matrice rkat7_mat5, figure 5.8, page 107 qui contient seulement 38114 éléments non nuls alors que AA^T et A^TA en contiennent respectivement 303376 et 170028. Ainsi, l'utilisation des boîtes noires permet de tirer parti de ce fait sans même réécrire l'algorithme de Wiedemann ! Il suffit d'implémenter l'algorithme de résolution de récurrence linéaire de Berlekamp/Massey (voir section 6.3, page 119) avec comme entrée un itérateur sur la séquence linéairement générée. La *même* implémentation du programme C++ peut alors servir, d'une part pour résoudre une récurrence linéaire classique, si l'itérateur représente un tableau, et, d'autre part, elle peut faire office d'algorithme de Wiedemann, si l'itérateur représente la suite des valeurs prises par v^TA^iu, et ce pour n'importe quelle classe représentant une matrice en boîte noire. Ainsi, notre programme Massey suppose qu'on lui fournit en entrée une classe C++ avec une méthode "$++$", qui permet de déplacer le pointeur vers l'élément suivant de la séquence et une méthode "$*$" qui permet d'accéder à l'élément courant dans la séquence. Nous avons implémenté un tel itérateur pour les boîtes noires :

Code 11.3.2 Itérateur pour boîtes noires

```
template<class BB,
class Vect = GoodChoice< BB::InVector,
                         BB::OutVector >::Vector >
class BB_Iterator {
public:
  typedef typename Vect::value_type  value_type;
private:
  BB * _mat; // matrix A
  Vect v0, v, u;

   value_type _value;

 public:
 ...

   void operator ++() {
       // v <- Au
     _mat->Apply(v, u);
       // _value <- v0^T . v
     _domain.dotproduct(_value, v0, v);
       // exchange, u <- A^i . u0
```

```
      swap(v, u);
    }

  value_type operator *() const {
    return _value;
  }
};
```

L'utilisation est alors très simple comme le montre l'exemple 11.3.3.

Code 11.3.3 Réutilisation de code avec Linbox

```
// argv[1] : Fichier format creux
// argv[2] : Fichier format creux
int main(int argc, char ** argv) {

// Un corps de taille 5^4
GFq F625(5,4);

// Deux matrices creuses
SparseLibBB< GFq > slBB(&F625, argv[1]);
 HybrideBB< GFq >   hyBB(&F625, argv[2]);

 // La composition slBB x hyBB est réalisée
 typedef  BB_Composition< SparseLibBB< GFq >,
                          HybrideBB< GFq > > Composed;
 Composed coBB( &slBB, &hyBB );

 // Un itérateur v^T itBB^i u
 // avec v et u vecteurs aléatoires
 BB_Iterator< Composed > itBB( &coBB );

 // Calcul du degré du polynôme minimal
 // de la matrice slBB x hyBB
 Massey< BB_Iterator<Composed> > MassMat( &F625, itBB);
 long degMat = MassMat.degree();

 // L'algorithme fonctionne de même sur une séquence
 vector< GFq::element > sequence;
 ...
 // La séquence est générée dans ce vecteur, par exemple
 // via un fichier ou un algorithme...
```

```
    // Calcul du degré du polynôme minimal
    // générateur de la séquence
    Massey< vector< GFq::element >::const_iterator > >
35      MassSeq( &F625, sequence.begin() );
    long degSeq = MassSeq.degree();

    return 0;
    }
```

12

LOGICIELS

« Je soutiens que la seule morale à la portée du présent siècle est la morale du bilboquet »

Jean-Jacques Rousseau

Sommaire

Ce chapitre présente l'organisation des implémentations que nous avons réalisées. Le premier module concerne principalement les algorithmes de la première partie sur les corps finis. Ces algorithmes sont implémentés comme un module interne de Givaro et utilisent l'arithmétique entière en précision arbitraire de GMP. La deuxième partie détaille l'organisation du prototype de bibliothèque pour Linbox. Les algorithmes implémentés sont principalement ceux de la deuxième partie. La bibliothèque comporte en outre plusieurs « enveloppes » destinées à l'utilisation de Linbox avec diverses bibliothèques spécialisées. Enfin, les algorithmes pour la forme de Smith sont contenus dans un module que nous avons réalisé pour GAP. Les fonctionnalités et caractéristiques principales de ce module sont ici brièvement exposées.

12.1 CORPS FINIS DANS GIVARO

Givaro contenait déjà des classes d'entiers à précision infinie construites au-dessus de GMP, ainsi qu'un module de polynômes construit à partir de vecteurs.

Nous avons donc, d'une part, implémenté des algorithmes probabilistes de test de primalité (algorithme de Rabin [139], algorithme de Lehmann [95] et tabulation des premiers nombres premiers), de factorisation d'entiers (algorithmes de Pollard [136], et Lenstra [106] — dont une version parallèle avec Athapascan-1 —) et de calcul de racines primitives (algorithme 3.2.7, page 49). D'autre part, nous avons implémenté des routines polynomiales de test d'irréductibilité, de factorisation sur des corps finis (algorithmes de Berlekamp [11, 12] et variantes de Cantor, Zassenhaus et Ben-Or, voir section 3.3, page 50). Ce module représente plus de 3500 lignes de code.

À partir de l'arithmétique des polynômes, il est alors possible de construire un module d'extensions algébriques par l'implémentation d'opérations modulo un polynôme irréductible. Ensuite, nous avons implémenté une conversion entre polynôme d'entiers modulo p et nombre p-adique. Par exemple, $X^2 + 2X + 3$ est représenté de manière unique en notation 7-adique par $7^2 + 2.7 + 3 = 66$. De là, on peut associer un nombre p-adique à chaque polynôme définissant un élément d'un corps fini $\mathtt{GF}(p^k)$. Deux tables de correspondances en entiers machine sont ensuite créées entre ces nombres et les indices correspondant à leur exposant par rapport à une racine primitive (voir section 3.1, page 43). Cela permet alors de passer rapidement d'une représentation à une autre.

Le schéma 12.1, page 213 résume l'organisation de ces modules à l'intérieur de Givaro, chacun étant implémenté comme un domaine Linbox.

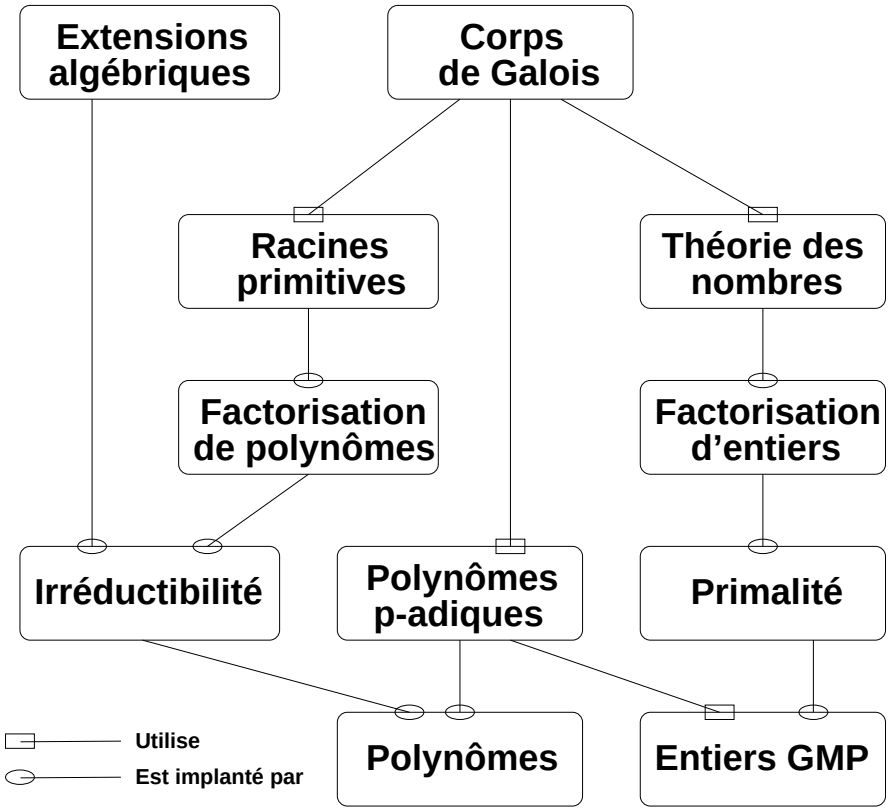

FIGURE 12.1 – Organisation des modules pour les corps finis dans Givaro

12.2 ORGANISATION DE LA BIBLIOTHÈQUE LINBOX

La bibliothèque Linbox est l'objet logiciel du projet Linbox, commun aux universités de Calgary, du Maryland, de l'Ontario de l'ouest, du Delaware et de la Caroline du Nord et aux laboratoires LMC et LID à Grenoble.

Avec Will Turner, de l'université de Caroline du Nord, nous avons réalisé le premier prototype de la bibliothèque. Deux aspects prédominent dans Linbox : un aspect hiérarchie de classes et généricité et un aspect algorithmique. Du côté logiciel, le travail de W. Turner se concentre sur la réalisation d'un cadre de programmation alliant efficacité et maîtrise de l'explosion de code (*code bloat*). Cela se concrétise par la mise au point d'« archétypes » permettant de choisir le bon compromis. D'autre part, nous avons développé le côté générique par l'implémentation et la définition de l'interface des structures de données (corps finis et domaines, boîtes noires, itérateurs, etc.) et des algorithmes (détermination des pré-

requis). En particulier, les algorithmes d'élimination, les algorithmes de Lanczos, les algorithmes de Wiedemann, décrits dans la deuxième partie de ce document et écrits de manière totalement générique, sont disponibles. Les algorithmes par blocs nécessitent encore quelques expérimentations et optimisations et seront disponibles dans une prochaine version.

La bibliothèque est divisée en deux parties principales : d'une part, un noyau (*library*) comportant les algorithmes écrits avec des entrées génériques et les objets internes de base ; d'autre part, un ensemble de classes permettant d'envelopper de manière automatique les bibliothèques existantes (*wrappers*) afin que leurs objets soient conformes aux pré-conditions supposées par les algorithmes*. Ainsi elle peut concilier efficacité (provenant de celle des bibliothèques spécialisées), généricité (pour la réutilisation du code) et maîtrise de la quantité de code généré (par l'utilisation des archétypes). La partie dont je me suis chargée représente actuellement près de 10000 lignes de code.

12.3 SIMPHOM : UN PACKAGE POUR GAP

Le calcul d'Homologie de complexes simpliciaux ainsi que de calcul de forme normale de Smith de matrices creuses a été assez peu traité jusqu'ici, c'est pourquoi Volkmar Welker, Frank Heckenbach, David Saunders et moi-même avons développé un module pour GAP répondant à ces deux attentes. GAP [67 - GAP (1999)], acronyme pour *Groups, Algorithms and Programming*, est un système de calcul pour l'algèbre discrète, et en particulier pour la théorie des groupes ; les objets combinatoires et algébriques, leurs constructeurs et les invariants associés sont déjà implémentés dans GAP, de même que des routines de calcul de la forme de Smith de matrices denses (par l'algorithme de Storjohann, en particulier). Ce système nous est donc apparu comme un bon point de départ pour fournir des routines construisant des complexes simpliciaux et leurs matrices aux limites associées ainsi que des routines calculant la forme de Smith, éventuellement incomplète, de matrices creuses.

12.3.1 LA BIBLIOTHÈQUE

Les fonctionnalités suivantes sont offertes :

*Un descriptif plus détaillé est disponible à l'adresse suivante : http ://www.cis.udel.edu/~caviness/linbox/library.html

– Créer des complexes, et en construire par adjonction.
  • Triangulations de variétés topologiques particulières.
  • Classes de complexes issus des théories des groupes et des graphes.
  • Classes de complexes d'ensembles partiellement ordonnés.
– Calculer les groupes d'Homologie à partir des facteurs invariants des matrices aux limites.
– Nous offrons quatre algorithmes pour le calcul des facteurs invariants (la forme de Smith) des matrices aux limites. En effet, le meilleur algorithme dépend de la matrice.
  • Calcul des facteurs invariants par élimination utilisant des entiers machine.
  • Calcul des facteurs invariants par élimination utilisant des entiers en précision arbitraire (bibliothèque GMP).
  • Calcul des facteurs invariants par l'algorithme Valence utilisant l'élimination pour calculer le rang modulo de petits nombres premiers.
  • Calcul des facteurs invariants par l'algorithme Valence utilisant uniquement les méthodes itératives, mais pouvant donner des résultats incomplets.

Une description plus détaillée des différentes fonctions disponibles est donnée dans [53 - Dumas et al. (2000)]. Nous décrivons dans la suite les quatre algorithmes implémentés.

12.3.2 LES ALGORITHMES

L'option `HomologyAlgorithms` du module permet de choisir l'un des quatre algorithmes de base pour calculer les invariants. Cette option peut donc prendre les valeurs suivantes : EliminateAlgorithm, EliminateGMPAlgorithm, ValenceElimAlgorithm, et ValenceBBAlgorithm. Brièvement, le choix de l'algorithme est le suivant :

EliminateAlgorithm : c'est l'algorithme classique sur les entiers (voir section 8.2, page 160), mais il est particulièrement adapté aux complexes simpliciaux. En effet, connaissant la structure particulière des matrices aux limites, la stratégie de pivot recherche principalement un pivot unitaire. De plus, les lignes de ces matrices contiennent peu d'éléments non nuls et ceux-ci sont très localisés. Il est alors possible d'effectuer un test rapide d'appartenance à l'espace généré par les précédentes lignes. Une telle ligne est alors simplement écartée, puisque n'influant pas sur le rang. Cette technique améliore la vitesse de résolution dans certains cas, mais elle augmente la capacité mémoire nécessaire. Les entiers sont des mots machine. Les dépassements éventuels de capacité du mot machine sont détectés et la méthode

renvoie “Échec” dans ce cas.

EliminateGMPAlgorithm : il est identique à ‘EliminateAlgorithm’ mais il utilise les entiers en précision arbitraire de GMP. Cet algorithme est plus lent d’un facteur 2 quand la taille des mots aurait suffi.

ValenceElimAlgorithm : il utilise les méthodes itératives (la Valence, voir chapitre 9) pour déterminer quels nombres premiers peuvent apparaître dans la forme de Smith. Il utilise l’élimination modulo p^k (voir section 5.3, page 93) pour déterminer les exposants des nombres premiers apparaissant dans la forme de Smith. Il peut être efficace quand ‘EliminateGMPAlgorithm’ dépasse la capacité mémoire puisqu’il n’y a pas de grossissement des coefficients modulo p. Toutefois, les problèmes de remplissage peuvent subsister. Enfin, l’algorithme est probabiliste. Il y a un risque que les nombres premiers déterminés par la Valence soient faux. L’option `UncertaintyTolerance` permet de régler la probabilité de succès désirée, en augmentant le nombre de nombres premiers utilisés pour calculer la Valence (voir section 9.1, page 166).

ValenceBBAlgorithm : il utilise uniquement les méthodes itératives. Dans certains cas, seule cette méthode évite les dépassements mémoire et fournit ainsi des éléments de réponse. Toutefois, cette méthode reste, pour l’instant, plus lente, en général, que les précédentes quand la mémoire n’est pas un facteur limitant. En outre, les résultats peuvent être incomplets puisque nous ne savons pas toujours calculer les exposants exacts des nombres premiers apparaissant dans la forme de Smith (voir section 10.4, page 188).

Pour un utilisateur désirant calculer des groupes d’Homologie, ou une forme normale de matrices creuses, une méthodologie de calcul est donc d’essayer, dans l’ordre, chacun de ces 4 algorithmes.

12.3.3 LES LANGAGES

Le module est écrit en GAP, Pascal et C++ et dépend de la bibliothèque GMP. Les fonctions de création de complexes sont écrites en GAP. Par souci d’efficacité, les algorithmes de calcul de la forme de Smith sont écrits en Pascal pour les deux premiers et en C++ pour les algorithmes Valence, un gestionnaire général écrit en Pascal permet l’utilisation des routines C++ et de la bibliothèque GMP. Les algorithmes Valence sont réalisés à partir de sous-ensembles de Givaro pour les corps finis et Linbox pour les algorithmes de rang et de formes normales. Le module SIMPHOM représente plus de 2200 lignes de documentation, 1700 lignes de code GAP, près de 3000 lignes de Pascal et plus de 5600 lignes de C++ consacrées aux algorithmes Valence, auxquelles s’ajoutent près de 8000 lignes issues

de différents modules de Givaro.

12.3.4 LES UTILISATEURS

Nous avons développé ce module pour trois types principaux d'utilisateurs. Le premier est un chercheur intéressé dans le calcul de groupes d'Homologie d'une série particulière de complexes simpliciaux pour lesquels l'Homologie est difficile à calculer. Dans ce cas, les calculs effectués par notre module peuvent conduire à des conjectures sur le comportement général des groupes d'homologie de la série de complexes. Ce schéma a été appliqué avec succès dans [6 - Babson et al. (1999), 17 - Björner et Welker (1999)] sur les matrices de couplages et de graphes non i-connectés, tout comme il a permis de prouver l'inexactitude de certaines conjectures théoriques (notamment par la résolution partielle de la forme de Smith de la matrice ch7-8.b5, voir tableau 10.3, page 194). Un deuxième type d'utilisateur est un chercheur intéressé par le calcul de formes normales de Smith sur matrices creuses. Par exemple, Matthias Franz et Gottfried Barthel à l'université de Konstanz, Michelle Wachs et John Shareshian à l'université de Miami, ou encore Abdul Salam Jarrah à l'université d'état du Nouveau Mexique, ont utilisé le module pour calculer des formes normales de Smith de matrices creuses. Enfin, le troisième type d'utilisateur est l'enseignant en topologie algébrique et combinatoire géométrique. Volkmar Welker a utilisé le module pour introduire ces notions dans son cours d'analyse combinatoire géométrique à l'université de Marburg.

CONCLUSIONS ET PERSPECTIVES

« Pour que tout soit consommé, pour que je me sente moins seul, il me restait à souhaiter qu'il y ait beaucoup de spectateurs le jour de mon exécution et qu'ils m'accueillent avec des cris de haine. »

Albert Camus

Nous avons présenté dans ce document un nouvel algorithme parallèle et efficace pour le calcul de la forme normale de Smith entière de grandes matrices creuses. Il a permis le premier calcul de cette forme pour plusieurs très grandes matrices. Pour arriver à ce résultat, nous avons étudié les différentes arithmétiques de corps finis, les heuristiques de renumérotations pour l'élimination de Gauß et les méthodes itératives de Krylov afin de déterminer leurs domaines privilégiés respectifs. Ensuite, deux points principaux sont à retenir dans notre nouvelle approche de calcul de la forme de Smith : d'une part, le nombre de nombres premiers nécessaires au calcul du polynôme minimal entier est réduit par l'utilisation de la terminaison anticipée et des ovales de Cassini et, d'autre part, la Valence contenant tous les nombres premiers de la forme de Smith, cela permet des calculs rapides dans des petits corps finis. De plus, le parallélisme général de la méthode est important et facilement exploitable. Enfin, nous avons mis au point des structures de données efficaces pour les corps finis, les matrices creuses et les méthodes itératives. Il ressort de ces travaux de recherche que :

(1) L'arithmétique la plus efficace sur les corps finis est une arithmétique tabulée utilisant des racines primitives.

(2) L'algorithme de Wiedemann et l'algorithme de Coppersmith nécessitent moins d'opérations que les algorithmes de Lanczos pour le calcul du rang.

(3) Les algorithmes d'élimination avec renumérotation sont très rapides pour calculer le rang quand le nombre d'éléments non nuls par ligne ne dépasse pas 5 ou 6, quand la matrice est quasiment diagonale ou quand il y a peu de lignes.

(4) L'algorithme de Wiedemann est un peu moins rapide dans les cas précédents, mais permet de calculer sur de plus grandes matrices.

(5) L'utilisation de la Valence permet le calcul de la forme de Smith lorsque les méthodes classiques échouent.

Ces conclusions ne constituent certes pas un aboutissement. En effet, de nombreuses questions demeurent pour l'instant en suspens :

(1) Comment implémenter efficacement des corps finis plus grands ? Une première piste peut être de combiner les approches polynomiale et tabulée pour rester assez efficace tout en diminuant l'espace mémoire nécessaire.

(2a) Comment améliorer l'efficacité pratique, en séquentiel et en parallèle, de

l'algorithme récursif d'élimination de Gauß par blocs sur matrices creuses ? Une première étape consiste à mettre en œuvre une stratégie de renumérotation adaptée au cas par blocs.

(2b) Par ailleurs, cet algorithme possède la meilleure complexité séquentielle pour le calcul du rang de matrices quelconques ; pour quelles tailles de matrices l'utilisation de la méthode de multiplication de Strassen peut elle rendre cet algorithme plus efficace ? D'après Von zur Gathen et Gerhard [68 - Gathen et Gerhard (1999), section 12.1], cette méthode doit pouvoir accélérer notre algorithme pour d'assez petites matrices. En effet, ils reportent qu'en pratique, à partir d'une taille d'environ 100×100, la méthode de Strassen permet de multiplier deux matrices plus rapidement qu'avec l'algorithme classique.

(3) Des tests doivent être menés pour les algorithmes parallèles itératifs par blocs, afin de mieux cerner leurs performances pratiques. Après l'étude théorique, ils semblent intéressants seulement sur des matrices moins creuses. Est-ce correct et comment peut-on améliorer les performances de l'algorithme scalaire parallèle sur machine distribuée ?

(4) Comment calculer complètement la forme de Smith de mk13.b5, m133.b3 et ch7-8.b5 ? Pour répondre, il suffit (!) sans doute de calculer le dernier invariant de Smith. Le nouvel algorithme de G. Villard (voir section 10.4, page 188) doit pouvoir être rendu plus pratique et répondre à cette question.

(5) Comment trouver des solutions entières positives à BIBD(22,8) ? La forme de Smith de cette matrice est composée de deux cent neuf 1, vingt et un 7 et un 28. Il doit être possible de trouver des matrices de passage simples entre la matrice et cette forme diagonale pour permettre de calculer des solutions entières.

(6) La borne d'Ozello sur la taille des vecteurs minimaux, et plus généralement la borne de Zippel-Schwartz, semble trop pessimiste dans de nombreux cas : préconditionnements diagonaux pour le rang, taille des éléments des vecteurs nécessaires au calcul du polynôme minimal. Comment améliorer cette borne, au moins pour des cas particuliers ?

(7) Comment prouver que l'algorithme de Reeds et Sloane permet bien de calculer le rang modulo p^e ?

(8) Quels préconditionnements peuvent être plus rapides que les préconditionnements Toeplitz et garantir l'obtention du rang par les méthodes itératives ?

(9) Comment certifier rapidement qu'un rang obtenu par une méthode probabiliste est correct ?

Enfin, si le premier prototype de la bibliothèque Linbox est efficace, il reste à implémenter de nombreuses enveloppes pour mettre en commun plus de bibliothèques spécialisées ainsi qu'à écrire les modules permettant l'utilisation de Linbox par d'autres systèmes de calcul scientifique. Si le package pour GAP commence à être employé, une interface avec Maple et Mathematica, par exemple, simplifierait grandement l'utilisation.

BIBLIOGRAPHIE

[1] John Abbott, Victor Shoup et Paul Zimmermann. – Factorisation in $\mathbb{Z}[x]$: the searching phase. Dans : *ISSAC'2000* [163 - Traverso (2000)], pages 1–7.

[2] Alfred V. Aho, John E. Hopcroft et Jeffrey D. Ullman. – *The Design and Analysis of Computer Algorithms*. – Addison-Wesley, 1974.

[3] Patrick Amestoy, François Pellegrini et Jean Roman. – Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. Dans : *Proceeding of IRREGULAR'99*, avril 1999. pages 986–995. – Puerto Rico.

[4] Patrick R. Amestoy, Timothy A. Davis et Iain S. Duff. – An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, volume 17, n° 4, octobre 1996, pages 886–905.

[5] Margaret Atwood. – *Variations on the word Sleep*.

[6] Eric Babson, Anders Björner, Svante Linusson, John Shareshian et Volkmar Welker. – Complexes of not i-connected graphs. *Topology*, volume 38, n° 2, 1999, pages 271–299.

[7] Achim Bachem et Ravindran Kannan. – Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM Journal on Computing*, volume 8, n° 4, novembre 1979, pages 499–507.

[8] Erwin H. Bareiss. – Sylvester's identity and multistep integer-preserving Gaussian elimination. *Mathematics of Computation*, volume 22, n° 103, juillet 1968, pages 565–578.

[9] Bernhard Beckermann et George Labahn. – A uniform approach for the fast computation of matrix-type Padé approximants . *SIAM Journal on Matrix Analysis and Applications*, volume 15, n° 3, juillet 1994, pages 804–823.

[10] Michael Ben-Or. – Probabilistic algorithms in finite fields. Dans : *22th Annual Symposium on Foundations of Computer Science*, octobre 1981. pages 394–398. – Los Alamitos, California, USA.

[11] Elwyn R. Berlekamp. – Factoring polynomials over finite fields. *Bell System Technical Journal*, volume 46, 1967, pages 1853–1859.

[12] Elwyn R. Berlekamp. – Factoring polynomials over large finite fields. *Mathematics of Computation*, volume 24, n° 11, 1970.

[13] Piotr Berman et Georg Schnitger. – On the performance of the minimum degree ordering for Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, volume 11, n° 1, janvier 1990, pages 83–88.

[14] Laurent Bernardin, Bruce Char et Erich Kaltofen. – Symbolic computation in java : An appraisement. Dans : *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, Vancouver, Canada*, édité par Sam Dooley, 29–31 juillet 1999. – ACM Press, New York.

[15] Robert R. Bitmead et Brian D. O. Anderson. – Asymptotically fast solution of Toeplitz and related systems of linear equations. *Linear Algebra and its Applications*, volume 34, 1980, pages 103–116.

[16] Anders Björner, László Lovász, Sinisa T. Vrećica et Rade T. Živaljević. – Chessboard complexes and matching complexes. *Journal of the London Mathematical Society*, volume 49, n° 1, 1994, pages 25–39.

[17] Anders Björner et Volkmar Welker. – Complexes of directed graphs. *SIAM Journal on Discrete Mathematics*, volume 12, n° 4, novembre 1999, pages 413–424.

[18] Guy E. Blelloch. – *NESL : A Nested Data-Parallel Language (Version 3.1)*. – Rapport technique n° CMU-CS-95-170, Computer Science Department, Carnegie Mellon University, septembre 1995.

[19] Robert D. Blumofe, Christopher F. Joerg, Bradley. C. Kuszmaul, Charles E. Leiserson, Keith H. Randall et Yuli C. E. Zhou. – Cilk : an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, volume 30, n° 8, août 1995, pages 207–216.

[20] Alfred Brauer. – Limits for the characteristic roots of a matrix. I. *Duke Mathematical Journal*, volume 13, 1946, pages 387–395.

[21] Alfred Brauer. – Limits for the characteristic roots of a matrix. II. *Duke Mathematical Journal*, volume 14, 1947, pages 21–26.

[22] Claude Brezinski, Michela Redivo Zaglia et Hassane Sadok. – Avoiding breakdown and near-breakdown in Lanczos type algorithms. *Numerical Algorithms*, volume 1, n° 3, 1991, pages 261–284.

[23] Jacques Briat, Ilan Ginzburg, Marcelo Pasin et Brigitte Plateau. – Athapascan runtime : efficiency for irregular problems. Dans : *EURO-PAR'97 Parallel Processing, Passau, Allemagne*, édité par Christian Lengauer et al., août 1997. – *Lecture Notes in Computer Science*, volume 1300, pages 591–600. – Springer.

[24] Richard A. Brualdi et Stephen Mellendorf. – Regions in the complex plane containing the eigenvalues of a matrix. *American Mathematical Monthly*, volume 101, n° 10, décembre 1994, pages 975–985.

[25] Johannes Buchmann, Michael J. Jacobson, Jr. et Edlyn Teske. – On some computational problems in finite abelian groups. *Mathematics of Computation*, volume 66, n° 220, octobre 1997, pages 1663–1687.

[26] David M. Burton. – *Elementary number theory*. – McGraw-Hill, 1998, quatrième édition, *International series in Pure and Applied Mathematics*.

[27] Albert Camus. – *L'étranger*. – 1942.

[28] David G. Cantor et Hans Zassenhaus. – A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, volume 36, n° 154, avril 1981, pages 587–592.

[29] Alexandre Carissimi. – *Le noyau exécutif Athapascan-0 et l'exploitation de la multiprogrammation légère sur les grappes de stations multiprocesseurs*. – Thèse de Doctorat en informatique, Institut National Polytechnique de Grenoble, France*, novembre 1999.

[30] Nicholas Carriero et David Gelernter. – How to write parallel programs : A guide to the perplexed. *ACM Computing Surveys*, volume 21, n° 3, septembre 1989, pages 323–357.

[31] Bertrand Carton et Fabien Giquel. – *Data processing using Athapascan-1*. – Rapport technique, Apache, juin 1999. http ://www-apache.imag.fr/software/ath1/athavista/Matrix.html.

[32] Gerson Cavalheiro, François Galilée et Jean-Louis Roch. – Athapascan-1 : Parallel programming with asynchronous tasks. Dans : *Yale Multithreaded Programming Workshop*, juin 1998. – USA.

[33] Gerson-Geraldo-Homrich Cavalheiro. – *Athapascan-1 : Interface générique pour l'ordonnancement dans un environnement d'exécution parallèle*. – Thèse de Doctorat en informatique, Institut National Polytechnique de Grenoble, France*, novembre 1999.

*Les thèses de l'INPG sont accessibles sur le serveur de la Médiathèque IMAG : ftp ://ftp.imag.fr/pub/Mediatheque.IMAG/theses

[34] Stefania Cavallar. – *Strategies in filtering in the number field sieve*. – Rapport technique, Centrum voor Wiskunde en Informatica, 31 mai 2000. http ://www.cwi.nl/ftp/CWIreports/MAS/MAS-R0012.ps.Z.

[35] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton Leong, Michael B. Monagan et Stephen M. Watt. – *Maple V Library Reference Manual*. – Springer-Verlag, 1991.

[36] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton Leong, Michael B. Monagan et Stephen M. Watt. – *First Leaves : A Tutorial Introduction to Maple V*. – New York, Springer Verlag, 1992.

[37] Tsu Wu J. Chou et George E. Collins. – Algorithms for the solution of systems of linear Diophantine equations. *SIAM Journal on Computing*, volume 11, n° 4, novembre 1982, pages 687–708.

[38] Jean Cocteau. – *Les mariés de la tour Eiffel*. – 1921.

[39] Don Coppersmith. – Fast evaluation of logarithms in fields of characterstic two. *IEEE Transactions on Information Theory*, volume IT-30, 1984, pages 587–594.

[40] Don Coppersmith. – Solving linear equations over $\mathtt{GF}(2)$: block Lanczos algorithm. *Linear Algebra and its Applications*, volume 192, octobre 1993, pages 33–60.

[41] Don Coppersmith. – Solving homogeneous linear equations over $\mathtt{GF}(2)$ via block Wiedemann algorithm. *Mathematics of Computation*, volume 62, n° 205, janvier 1994, pages 333–350.

[42] Don Coppersmith et Shmuel Winograd. – Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, volume 9, n° 3, 1990, pages 251–280.

[43] Leonardo Dagum et Ramesh Menon. – OpenMP : An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, volume 5, n° 1, janvier–mars 1998, pages 46–55.

[44] Michel Demazure. – *Cours d'algèbre. Primalité, Divisibilité, Codes*. – Paris, Cassini, 1997, *Nouvelle biliothèque Mathématique*, volume XIII.

[45] Leonard Eugene Dickson. – *Linear Groups with an Exposition of the Galois Field Theory*. – New York, Reprinted (1958) by Dover, 1901.

[46] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling et Iain Duff. – A set of level 3 Basic Linear Algebra Subprograms. *Transactions on Mathematical Software*, volume 16, n° 1, mars 1990, pages 1–17.

[47] Jack J. Dongarra, Steve W. Otto, Marc Snir et David Walker. – *An Introduction to the MPI Standard*. – Rapport technique n° UT-CS-95-274, Department of Computer Science, University of Tennessee, janvier 1995.

[48] Mathias Doreille. – *Calcul Formel et Parallélisme : résolution de systèmes linéaires denses, structurés et creux*. – Diplôme d'Études Approfondies en Mathématiques appliquées, Université Joseph Fourier, 15 juin 1995. http ://www-apache.imag.fr/RR/dea_doreille.ps.gz.

[49] Mathias Doreille. – *Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. – Thèse de Doctorat en mathématiques appliquées, Institut National Polytechnique de Grenoble, France*, décembre 1999.

[50] Jean-Louis Dornstetter. – On the equivalence between Berlekamp's and Euclid's algorithms. *IEEE transactions on information theory*, volume 33, mai 1987, pages 428–431.

[51] Iain S. Duff, Albert M. Erisman et John K. Reid. – *Direct Methods for Sparse Matrices*. – Oxford, UK, Clarendon Press, 1986.

[52] Jean-Guillaume Dumas. – Calcul parallèle du polynôme minimal entier en Athapascan-1 et Linbox. Dans : *RenPar'2000. Actes des douzièmes rencontres francophones du parallélisme, Besançon, France*, 19-22 juin 2000, pages 119–124.

[53] Jean-Guillaume Dumas, Frank Heckenbach, B. David Saunders et Volkmar Welker. – *Simplicial Homology, a (proposed) share package for GAP*, mars 2000. Manual (http ://www.cis.udel.edu/~dumas/Homology).

[54] Jean-Guillaume Dumas, B. David Saunders et Gilles Villard. – Integer Smith form via the Valence : experience with large sparse matrices from Homology. Dans : *ISSAC'2000* [163 - Traverso (2000)], pages 95–105.

[55] Jean-Guillaume Dumas, B. David Saunders et Gilles Villard. – On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computations*, volume 32, n° 1/2, juillet–août 2001, pages 71–99.

[56] Pierre Dusart. – *Autour de la fonction qui compte le nombre de nombres premiers*. – Thèse de Doctorat en Mathématiques Appliquées, Théorie des nombres, Université de Limoges, 1998.

[57] Pierre Dusart. – The k^{th} prime is greater than $k(\ln k + \ln\ln k - 1)$ for $k \geq 2$. *Mathematics of Computation*, volume 68, n° 225, janvier 1999, pages 411–415.

[58] Wayne Eberly, Mark Giesbrecht et Gilles Villard. – 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*, 12–14 novembre 2000.

[59] Wayne Eberly et Erich Kaltofen. – On randomized Lanczos algorithms. Dans : *ISSAC'97* [97 - Küchlin (1997)], pages 176–183.

[60] Peter D. T. A. Elliott et Leo Murata. – On the average of the least primitive root modulo p. *Journal of The london Mathematical Society*, volume 56, n° 2, 1997, pages 435–454.

[61] Jean-Charles Faugère. – Parallelization of Gröbner basis. Dans : *First International Symposium on Parallel Symbolic Computation, PASCO '94, Hagenberg/Linz, Austria*, édité par Hoon Hong, 26–28 septembre 1994. – *Lecture notes series in computing*, volume 5, pages 124–132.

[62] Jean-Charles Faugère. – *A new efficient algorithm for computing Gröbner bases (F_4)*. – Rapport technique, Laboratoire d'Informatique de Paris 6, janvier 1999. http ://www-calfor.lip6.fr/~jcf.

[63] Patrick Fitzpatrick et Graham H. Norton. – Linear recurrence relations and an extended subresultant algorithm. Dans : *Proceedings of the 3rd International Colloquium on Coding Theory and Applications*, édité par G. Cohen et J. Wolfmann, novembre 1989. – *Lecture Notes in Computer Science*, volume 388, pages 232–246. – New York.

[64] Joel Friedman. – Computing Betti numbers via combinatorial Laplacians. Dans : *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, 22–24 mai 1996, pages 386–391. – Philadelphia, Pennsylvania.

[65] François Galilée. – *Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle*. – Thèse de Doctorat en informatique, Institut National Polytechnique de Grenoble, France*, septembre 1999.

[66] Feliks Rudimovich Gantmacher. – *The Theory of Matrices*. – New York, Chelsea, 1959.

[67] GAP. – *Groups, Algorithms, and Programming, Version 4.2*. – The GAP Group, Aachen, St Andrews, 1999. http ://www-gap.dcs.st-and.ac.uk/~gap.

[68] Joachim von zur Gathen et Jürgen Gerhard. – *Modern Computer Algebra*. – New York, NY, USA, Cambridge University Press, 1999.

[69] Carl Friedrich Gauß. – Disquisitiones arithmeticæ, 1801.

[70] Thierry Gautier. – *Calcul Formel et Parallélisme : Conception du Système* GIVARO *et Applications au Calcul dans les Extensions Algébriques*. – Thèse de Doctorat en Mathématiques Appliquées, Institut National Polytechnique de Grenoble, 1996. http ://www-apache.imag.fr/software/givaro.

[71] David Gelernter. – Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, volume 7, n° 1, janvier 1985, pages 80–112.

[72] Mark W. Giesbrecht. – Fast computation of the Smith Normal Form of an integer matrix. Dans : *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation : July 10–12, 1995, Montreal, Canada*, édité par A. H. M. Levelt, 1995. pages 110–118. – ACM Press.

[73] Mark W. Giesbrecht. – Probabilistic computation of the Smith Normal Form of a sparse integer matrix. *Lecture Notes in Computer Science*, volume 1122, 1996, pages 173–186.

[74] Gene H. Golub et Charles F. Van Loan. – *Matrix computations*. – Baltimore, MD, USA, The Johns Hopkins University Press, 1996, troisième édition, *Johns Hopkins Studies in the Mathematical Sciences*.

[75] Torbjörn Granlund. – *The GNU multiple precision arithmetic library*, 2000. Version 3.0, http ://www.gnu.org/gnulist/production/gnump.html.

[76] Roger G. Grimes, John G. Lewis et Horst D. Simon. – A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM Journal on Matrix Analysis and Applications*, volume 15, n° 1, 1994, pages 228–272.

[77] James Lee Hafner et Kevin S. McCurley. – A rigorous subexponential algorithm for computation of class groups. *Journal of the American Mathematical Society*, volume 2, 1989, pages 837–850.

[78] Bruce Hendrickson et Edward Rothberg. – Improving the run time and quality of nested dissection ordering. *SIAM Journal on Scientific Computing*, volume 20, n° 2, mars 1999, pages 468–489.

[79] High Performance Fortran Forum. – High Performance Fortran language specification. *Scientific Programming*, volume 2, n° 1-2, 1993, pages 1–170.

[80] Oscar H. Ibarra, Shlomo Moran et Roger Hui. – A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, volume 3, n° 1, mars 1982, pages 45–56.

[81] IEEE. – *The Institute of Electrical and Electronics Engineers, Inc 1003.1c-1995 : Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2 : Threads Extension (C Language)*. – IEEE Computer Society Press, 1995.

[82] Costas S. Iliopoulos. – Worst-case complexity bounds on algorithms for computing the canonical structure of finite Abelian groups and the Hermite and Smith normal forms of an integer matrix. *SIAM Journal on Computing*, volume 18, n° 4, 1989, pages 658–669.

[83] Erich Kaltofen. – Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, volume 64, n° 210, avril 1995, pages 777–806.

[84] Erich Kaltofen. – Challenges of symbolic computation : My favorite open problems. *Journal of Symbolic Computation*, volume 29, n° 6, juin 2000, pages 891–919.

[85] Erich Kaltofen, Mukkai S. Krishnamoorthy et B. David Saunders. – Mr Smith goes to Las Vegas : Randomized parallel computation of the Smith normal form of polynomial matrices. Dans : *Proceedings of the 1987 European Conference on Computer Algebra*, édité par James H. Davenport, juin 1989. – *Lecture Notes in Computer Science*, volume 378, pages 317–322. – Berlin.

[86] Erich Kaltofen, Mukkai S. Krishnamoorthy et B. David Saunders. – Parallel algorithms for matrix normal forms. *Linear Algebra and its Applications*, volume 136, 1990, pages 189–208.

[87] Erich Kaltofen, Wen-Shin Lee et Austin A. Lobo. – Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. Dans : *ISSAC'2000* [163 - Traverso (2000)], pages 192–201.

[88] Erich Kaltofen et Austin Lobo. – Distributed matrix-free solution of large sparse linear systems over finite fields. Dans : *Proceedings of High Performance Computing 1996, San Diego, California*, édité par A.M. Tentner, avril 1996. – Society for Computer Simulation, Simulation Councils, Inc.

[89] Erich Kaltofen et Viktor Pan. – Processor-efficient parallel solution of linear systems II : The positive characteristic and singular cases. Dans : *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, édité par IEEE, octobre 1992. pages 714–723. – IEEE Computer Society Press.

[90] Erich Kaltofen et Viktor Pan. – Parallel solution of Toeplitz and Toeplitz-like linear systems over fields of small positive characteristic. Dans : *First International Symposium on Parallel Symbolic Computation, PASCO '94, Hagenberg/Linz, Austria*, édité par Hoon Hong, 26–28 septembre 1994. – *Lecture Notes in Computer Science*, volume 5, pages 225–233. – World Scientific Publishing Co.

[91] Erich Kaltofen et B. David Saunders. – On Wiedemann's method of solving sparse linear systems. Dans : *Applied Algebra, Algebraic Algorithms and Error–Correcting Codes (AAECC '91)*, octobre 1991. – *Lecture Notes in Computer Science*, volume 539, pages 29–38.

[92] Erich Kaltofen et Victor Shoup. – Fast polynomial factorization over high algebraic extensions of finite fields. Dans : *ISSAC'97* [97 - Küchlin (1997)], pages 184–188.

[93] Richard K. Karp, Michael Luby et Friedhelm Meyer auf der Heide. – Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, volume 16, n° 4/5, octobre–novembre 1996, pages 517–542.

[94] George Karypis et Vipin Kumar. – A fast and high quality mutilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, volume 20, n° 1, janvier 1999, pages 359–392.

[95] Donald E. Knuth. – *Seminumerical Algorithms*. – Reading, MA, USA, Addison-Wesley, 1997, troisième édition, *The Art of Computer Programming*, volume 2.

[96] Neal Koblitz. – *A course in number theory and cryptography*. – Berlin, Germany / Heidelberg, Germany / London, UK / etc., Springer-Verlag, 1987, *Graduate texts in mathematics*, volume 114.

[97] Wolfgang W. Küchlin (éditeur). – *ISSAC'97. Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii*, 21–23 juillet 1997. – ACM Press, New York.

[98] Vipin Kumar, Ananth Grama, Anshul Gupta et George Karypis. – *Introduction to parallel computing. Design and analysis of algorithms*. – The Benjamin/Cummings Publishing Company, Inc., 1994.

[99] V. L. Kurakin. – Representations over $\mathbb{Z}/p^n\mathbb{Z}$ of linear recurring sequence of maximal period over $\mathtt{GF}(p)$. *Discrete Mathematics and Applications*, volume 3, n° 3, 1993, pages 275–296.

[100] V. L. Kurakin. – The first coordinate sequence of a linear recurrence of maximal period over a galois ring. *Discrete Mathematics and Applications*, volume 4, n° 2, 1994, pages 129–141.

[101] Dan Laksov et Anders Thorup. – Counting matrices with coordinates in finite fields and of fixed rank. *Mathematica Scandinavia*, volume 74, n° 1, 1994, pages 19–33.

[102] Brian A. LaMacchia et Andrew M. Odlyzko. – Solving large sparse linear systems over finite fields. *Lecture Notes in Computer Science*, volume 537, 1991, pages 109–133. – http ://www.research.att.com/~amo/doc/arch/sparse.linear.eqs.ps.

[103] Rob Lambert. – *Computational aspects of discrete logarithms*. – Doctor of Philosophy Thesis in Electrical engineering, University of Waterloo, Ontario, Canada, 1996. http ://www.cacr.math.uwaterloo.ca/techreports/2000/lambert-thesis.ps.

[104] Patrick Lascaux et Raymond Théodor. – *Analyse numérique matricielle appliquée à l'art de l'ingénieur, Méthodes itératives*. – Masson, 1994, seconde édition.

[105] Hong R. Lee et B. David Saunders. – Fraction free Gaussian elimination for sparse matrices. *Journal of Symbolic Computation*, volume 19, n° 5, avril 1995, pages 393–402.

[106] Hendrik Lenstra. – Factoring integers with elliptic curves. *The Annals of Mathematics*, volume 126, n° 3, novembre 1987, pages 649–673.

[107] Robert H. Lewis. – *Fermat, a computer algebra system for polynomial and matrix computations*, 1997. http ://www.bway.net/~lewis.

[108] The Linbox Group. – Linear algebra with black boxes, 2000. http ://www.cis.udel.edu/~caviness/linbox.

[109] Austin Lobo. – *Matrix-free linear system solving and applications to symbolic computation*. – Doctor of Philosophy Thesis in Computer Science, Faculty of Rensselaer Polytechnic Institute, décembre 1995.

[110] James L. Massey. – Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, volume IT-15, 1969, pages 122–127.

[111] Jean-Pierre Massias et Guy Robin. – Bornes effectives pour certaines fonctions concernant les nombres premiers. *Journal de Théorie des Nombres de Bordeaux*, volume 8, 1996, pages 213–238.

[112] Kurt Mehlhorn et Stefan Näher. – *LEDA : A Platform of Combinatorial and Geometric Computing*. – Cambridge, England, Cambridge University Press, janvier 1999. http ://www.mpi-sb.mpg.de/LEDA.

[113] Maurice Mignotte. – *Mathématiques pour le calcul formel*. – Presses Universitaires Françaises, 1989.

[114] Michael B. Monagan, Keith O. Geddes, K. M. Heal, George Labahn et S. M. Vorkoetter. – *Maple V Programming Guide*. – Berlin, Germany / Heidelberg, Germany / London, UK / etc., Springer-Verlag, 1998. With the assistance of J. S. Devitt, M. L. Hansen, D. Redfern, and K. M. Rickard.

[115] Peter L. Montgomery. – A block Lanczos algorithm for finding dependencies over $\mathtt{GF}(2)$. Dans : *Proceedings of the 1995 International Conference on the Theory and Application of Cryptographic Techniques, Saint-Malo, France*, édité par Louis C. Guillou et Jean-Jacques Quisquater, mai 1995. – *Lecture Notes in Computer Science*, volume 921, pages 106–120.

[116] Bernard Mourrain et Hélène Prieto. – *The ALP reference manual : A framework for symbolic and numeric computations*, 2000. http ://www-sop.inria.fr/saga/logiciels/ALP.

[117] MPI Forum. – MPI : A Message Passing Interface. Dans : *Proceedings, Supercomputing '93 : Portland, Oregon, November 15–19, 1993*, édité par IEEE, 1993. pages 878–883. – IEEE Computer Society Press.

[118] Ketan Mulmuley. – A fast parallel algorithm to compute the rank of a matrix. *Combinatorica*, volume 7, n° 1, 1987, pages 101–104.

[119] James R. Munkres. – *Elements of algebraic topology*, chapitre The computability of homology groups, pages 53–61. – The Benjamin/Cummings Publishing Company, Inc., 1994, *Advanced Book Program*.

[120] Leo Murata. – On the magnitude of the least prime primitive root. *Journal of Number Theory*, volume 37, n° 1, 1991, pages 47–66.

[121] Raymond Namyst. – *PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. – Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, France, septembre 1997.

[122] Pablo Neruda. – Oda al libro (II). Dans : *Odas Elementales*. – 1954.

[123] Ingmar Neumann et Wolfgang Wilhemi. – A parallel algorithm for achieving the Smith normal form of an integer matrix. *Parallel Computing*, volume 22, 1996, pages 1399–1412.

[124] Morris Newman. – *Integral Matrices*. – Academic Press, 1972, *Pure and Applied Mathematics, a Series of Monographs and Textbooks*, volume 45.

[125] Noir Désir. – Les persiennes. Dans : *666.667 Club*. – 1996.

[126] Graham H. Norton. – On the minimal realizations of a finite sequence. *Journal of Symbolic Computation*, volume 20, n° 1, juillet 1995, pages 93–115.

[127] Graham H. Norton. – On minimal realization over a finite chain ring. *Designs, Codes, and Cryptography*, volume 16, n° 2, février 1999, pages 161–178.

[128] Graham H. Norton. – On shortest linear recurrences. *Journal of Symbolic Computation*, volume 27, n° 3, mars 1999, pages 325–349.

[129] Scott Oaks et Henry Wong. – *Java Threads*. – 981 Chestnut Street, Newton, MA 02164, USA, O'Reilly & Associates, Inc., 1999, seconde édition.

[130] Andrew M. Odlyzko. – Discrete logarithms : The past and the future. *Designs, Codes, and Cryptography*, volume 19, 2000, pages 129–145.

[131] Dianne P. O'Leary. – The block conjugate gradient algorithm and related methods. *Linear Algebra and its Applications*, volume 29, 1980, pages 293–322.

[132] Tomás Oliveira e Silva. – Least primitive root of prime numbers, 3 janvier 2000. http ://www.ieeta.pt/~tos/p-roots.html.

[133] Patrick Ozello. – *Calcul exact des formes de Jordan et de Frobenius d'une matrice*. – Thèse de Doctorat en Informatique et Mathématiques Appliquées, Université scientifique technologique et médicale de Grenoble, janvier 1987.

[134] Beresford N. Parlett, Derek R. Taylor et Zhishun A. Liu. – A look-ahead Lanczos algorithm for unsymmetric matrices. *Mathematics of Computation*, volume 44, n° 169, janvier 1985, pages 105–124.

[135] Olaf Penninga. – *Finding column dependencies in sparse matrices over $\mathbb{F}_2$ by block Wiedemann*. – Master Thesis in Mathematics, Leiden University, The Netherlands, 21 août 1998. http ://www.cwi.nl/ftp/CWIreports/MAS/MAS-R9819.ps.Z.

[136] John M. Pollard. – A Monte Carlo method for factorization. *BIT Numerical Mathematics*, volume 15, n° 3, 1975, pages 331–334.

[137] Roldan Pozo, Karin Remington et Andrew Lumsdaine. – *Sparse-Lib++ v.1.5 : Sparse Matrix Class Library, reference guide*. – Rapport technique, National Institute of Standards and Technology, 1996. http ://math.nist.gov/sparselib++.

[138] Jacques Prévert. – Intermède. Dans : *Spectacle*. – 1951.

[139] Michaël O. Rabin. – A probabilistic algorithm for testing primality. *Journal of Number Theory*, volume 12, 1980.

[140] Victor J. Rayward-Smith. – On computing the Smith normal form of an integer matrix. *ACM Transactions on Mathematical Software*, volume 5, n° 4, décembre 1979, pages 451–456.

[141] James A. Reeds et Neil J. A. Sloane. – Shift-register synthesis (modulo m). *SIAM Journal on Computing*, volume 14, n° 3, août 1985, pages 505–513.

[142] Victor Reiner et Joel Roberts. – Minimal resolutions and the homology of matching and chessboard complexes. *Journal of Algebraic Combinatorics*, volume 11, n° 2, mars 2000, pages 135–154.

[143] Martin C. Rinard et Monica S. Lam. – The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, volume 20, n° 3, mai 1998, pages 483–545.

[144] Jean-Louis Roch et Gilles Villard. – Parallel computer algebra, a survey. Dans : *ISSAC'97* [97 - Küchlin (1997)]. Tutorial.

[145] J. Barkley Rosser et Lowell Schoenfeld. – Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, volume 6, 1962, pages 64–94.

[146] Edward Rothberg et Stanley C. Eisenstat. – Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, volume 19, n° 3, juillet 1998, pages 682–695.

[147] Jean-Jacques Rousseau. – *Les confessions*. – 1766.

[148] Yousef Saad. – On the Lanczos method for solving symmetric systems with several right-hand sides. *Mathematics of Computation*, volume 48, n° 178, avril 1987, pages 651–662.

[149] Yousef Saad. – *Iterative Methods for Sparse Linear Systems*, chapitre Krylov Subspace Methods, pages 144–229. – Boston, PWS Publishing, 1996.

[150] Yousef Saad. – *Iterative Methods for Sparse Linear Systems*, chapitre Storage schemes, pages 84–86. – Boston, PWS Publishing, 1996.

[151] Victor Shoup. – Searching for primitive roots in finite fields. *Mathematics of Computation*, volume 58, n° 197, janvier 1992, pages 369–380.

[152] Victor Shoup. – Fast construction of irreducible polynomials over finite fields. *Journal of Symbolic Computation*, volume 17, n° 5, mai 1994, pages 371–391.

[153] Victor Shoup. – NTL3.7a : A library for doing number theory, 2000. http ://www.shoup.net/ntl.

[154] Ernest Sibert, Harold F. Mattson et Paul Jackson. – Finite Field Arithmetic Using the Connection Machine. Dans : *Proceedings of the second International Workshop on Parallel Algebraic Computation, Ithaca, USA*, édité par Richard Zippel, mai 1990. – *Lecture Notes in Computer Science*, volume 584, pages 51–61. – Springer Verlag.

[155] Horst D. Simon. – The Lanczos algorithm with partial reorthogonalization. *Mathematics of Computation*, volume 42, 1984, pages 115–142.

[156] Henry John Stephen Smith. – On systems of linear indeterminate equations and congruences. *Philosophical Transactions of the Royal Society*, volume 151, 1861, pages 293–326.

[157] Arne Storjohann. – Near optimal algorithms for computing Smith normal forms of integer matrices. Dans : *ISSAC '96 : Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, Zurich, Switzerland*, édité par Yagati N. Lakshman, 24–26 juillet 1996, pages 267–274.

[158] Arne Storjohann. – *Algorithms for Matrix Canonical Forms*. – Doctor of Philosophy Thesis in Computer Science, Institut für Wissenschaftliches Rechnen, ETH-Zentrum, Zürich, Switzerland, novembre 2000.

[159] Volker Strassen. – Gaussian elimination is not optimal. *Numerische Mathematik*, volume 13, 1969, pages 354–356.

[160] Vaidy S. Sunderam. – PVM : a framework for parallel distributed computing. *Concurrency, practice and experience*, volume 2, n° 4, décembre 1990, pages 315–339.

[161] Olga Taussky. – Bounds for characteristic roots of matrices. *Duke Mathematical Journal*, volume 15, 1948, pages 1043–1044.

[162] Jeremy Teitelbaum. – Euclid's algorithm and the Lanczos method over finite fields. *Mathematics of Computation*, volume 67, n° 224, octobre 1998, pages 1665–1678.

[163] Carlo Traverso (éditeur). – *ISSAC'2000. Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation, Saint Andrews, Scotland*, 6–9 août 2000. – ACM Press, New York.

[164] Alan M. Turing. – Rounding-off errors in matrix processes. Dans : *Pure Mathematics*, édité par J. L. Britton, pages 287–308. – 1948. Cité par Felipe Cucker à *ISSAC'2000* [163 - Traverso (2000)].

[165] Richard S. Varga. – *Matrix iterative analysis*. – Springer-Verlag, 2000, seconde édition, *Springer series in Computational Mathematics*.

[166] Richard S. Varga et Alan Krautstengl. – On gersgörin-type problems and ovals of cassini. *Electronic Transactions on Numerical Analysis*, volume 8, 1999, pages 15–20. – http ://etna.mcs.kent.edu.

[167] Boris Vian. – *L'herbe rouge*. – 1950.

[168] Gilles Villard. – Fast parallel algorithms for matrix reduction to normal forms. *Applicable Algebra in Engineering, Communication and Computing*, volume 8, n° 6, 1997, pages 511–538.

[169] Gilles Villard. – Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. Dans : *ISSAC'97* [97 - Küchlin (1997)], pages 32–39.

[170] Gilles Villard. – *A study of Coppersmith's block Wiedemann algorithm using matrix polynomials*. – Rapport technique n° 975–IM, LMC/IMAG, avril 1997.

[171] Gilles Villard. – Computing the Frobenius normal form of a sparse matrix. Dans : *Proceedings of the Third International Workshop on Computer Algebra in Scientific Computing, Samarkand, Uzbekistan*, édité par Victor G. Ganzha, Ernst W. Mayr et Evgenii V. Vorozhtsov, 5–9 octobre 2000.

[172] Wal D. Wallis, Anne Penfold Street et Jennifer Seberry Wallis. – *Combinatorics : Room Squares, Sum-Free Sets, Hadamard Matrices*. – Berlin, Springer-Verlag, 1972, *Lecture Notes in Mathematics*, volume 292.

[173] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach et Robert S. Sutor. – *AXIOM Library Compiler User Guide*. – Rapport technique, The Numerical Algorithms Group, Ltd., Oxford, 1994.

[174] Douglas H. Wiedemann. – Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, volume 32, n° 1, janvier 1986, pages 54–62.

[175] Stephen Wolfram. – *The Mathematica book*. – New York, NY, USA and 100 Trade Center Drive, Champaign, IL 61820-7237, USA, Cambridge University Press and Wolfram Research, Inc., 1999, quatrième édition.

[176] Mihali Yannakakis. – Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, volume 2, n° 1, mars 1981, pages 77–79.

[177] Richard Zippel. – *Effective Polynomial Computation*, chapitre Zero Equivalence Testing, pages 189–206. – Kluwer Academic Publishers, 1993.

[178] Zahari Zlatev. – *Computational Methods for General Sparse Matrices*, chapitre Pivotal Strategies for Gaussian Elimination, pages 67–86. – Norwell, MA, Kluwer Academic Publishers, 1992.

RÉSUMÉ : Depuis quelques années, l'extension de l'utilisation de l'informatique dans tous les domaines de recherche scientifique et technique se traduit par un besoin croissant de puissance de calcul. Il est donc vital d'employer les microprocesseurs en *parallèle*. Le problème principal que nous cherchons à résoudre dans cette thèse est le calcul d'une forme canonique de très grandes matrices creuses à coefficients entiers, la forme normale de Smith. Par «très grandes», nous entendons un million d'inconnues et un million d'équations, c'est-à-dire mille milliards de coefficients. De tels systèmes sont même, en général, impossibles à stocker actuellement. Cependant, nous nous intéressons à des systèmes dans lesquels beaucoup de ces coefficients sont identiques et valent zéro ; on parle dans ce cas de système *creux*. Enfin, nous voulons résoudre ces systèmes de manière exacte, c'est-à-dire que nous travaillons avec des nombres entiers ou dans une structure algébrique plus petite et autorisant toutes les opérations classiques, un *corps fini*. La reconstruction de la solution entière à partir des solutions plus petites est ensuite relativement aisée.

MOTS CLÉS : Corps finis. Matrice creuse. Renumérotation et élimination de Gauß. Méthodes itératives de Krylov. Matrice en boîte noire. Algorithme de Wiedemann. Forme normale de Smith entière. Algorithme Valence.

TITLE : EFFICIENT PARALLEL ALGORITHMS FOR COMPUTER ALGEBRA: SPARSE LINEAR ALGEBRA AND ALGEBRAIC EXTENSIONS

ABSTRACT: In every field of scientific and industrial research, the extension of the use of computer science has resulted in an increasing need for computing power. It is thus vital to use these computing resources in *parallel* . In this thesis we seek to compute the canonical form of very large sparse matrices with integer coefficients, namely the integer Smith normal form. By "very large", we mean a million indeterminates and a million equations, i.e. thousand billion of coefficients. Nowadays, such systems are usually not even storable. However, we are interested in systems for which many of these coefficients are identical ; in this case we talk about *sparse* systems. We want to solve these systems in an exact way, i.e. we work with integers or in smaller algebraic structures where all the basic arithmetic operations are still valid, namely *finite fields*. The rebuilding of the whole solution from the smaller solutions is then relatively easy.

KEYWORDS: Finite fields. Large sparse matrix. Reordering Gaussian elimination. Krylov iterative methods. Black Box. Wiedemann Algorithm. Integer Smith normal form. Valence Algorithm.

LABORATOIRE INFORMATIQUE ET DISTRIBUTION – ENSIMAG, antenne de Montbonnot. ZIRST – 51, av. Jean Kuntzmann, 38330 Montbonnot Saint Martin.